conference
proceedings

20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)

Boston, MA, USA
April 17–19, 2023

Sponsored by
usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
In cooperation with
ACM SIGCOMM and ACM SIGOPS

ISBN 978-1-939133-33-5

Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23) Boston, MA, USA April 17–19, 2023

NSDI '23 Sponsors

Platinum Sponsor

Gold Sponsors

Silver Sponsors

Bronze Sponsors

Open Access Sponsor

USENIX Supporters

USENIX Patrons

Amazon • Futurewei • Google • Meta

USENIX Benefactors

Bloomberg • NetApp

USENIX Partner

Thinkst Canary • Two Sigma

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)

April 17–19, 2023
Boston, MA, USA

© 2023 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-33-5

Conference Organizers

Program Committee

Sangeetha Abdu-Jyothi, *University of California, Irvine, and VMware Research*
Fadel Adib, *Massachusetts Institute of Technology*
Rachit Agarwal, *Cornell University*
Aditya Akella, *The University of Texas at Austin*
Deniz Altinbuken, *Google*
Ganesh Ananthanarayanan, *Microsoft Research*
Maria Apostolaki, *Princeton University*
Katerina Argyraki, *EPFL*
Behnaz Arzani, *Microsoft Research*
Adam Belay, *Massachusetts Institute of Technology*
Ken Birman, *Cornell University*
Matthew Caesar, *University of Illinois at Urbana–Champaign*
Marco Canini, *KAUST*
Ranveer Chandra, *Microsoft Research*
Ang Chen, *Rice University*
Paolo Costa, *Microsoft Research*
Murat Demirbas, *Amazon*
Nandita Dukkipati, *Google*
Ramakrishnan Durairajan, *University of Oregon*
Giuila Fanti, *Carnegie Mellon University*
Anja Feldmann, *Max Planck Institute for Informatics*
Bryan Ford, *EPFL*
Yashar Ganjali, *University of Toronto, Huawei Canada*
Mojgan Ghasemi, *Google*
Yasaman Ghasempour, *Princeton University*
Soudeh Ghorbani, *Johns Hopkins University*
Shyam Gollakota, *University of Washington*
Prateesh Goyal, *Microsoft Research*
Arpit Gupta, *University of California, Santa Barbara*
Indranil Gupta, *University of Illinois at Urbana–Champaign*
Hamed Haddadi, *Imperial College London*
Andreas Haeberlen, *University of Pennsylvania*
Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST)*
Haitham Hassanieh, *University of Illinois at Urbana–Champaign*
Michio Honda, *University of Edinburgh*
Jon Howell, *VMware Research*
Wenjun Hu, *Yale University*
Rebecca Isaacs
Anand Iyer, *Microsoft Research*
Vikram Iyer, *University of Washington*
Zhihao Jia, *Carnegie Mellon University*
Junchen Jiang, *University of Chicago*
Xin Jin, *Peking University*
Srikanth Kandula, *Microsoft Research*
Sachin Katti, *Stanford University*
Anurag Khandelwal, *Yale University*
Song Min Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Marios Kogias, *Imperial College London*
Dejan Kostic, *KTH Royal Institute of Technology*
Gautam Kumar, *Google*
Jeongkeun Lee, *Intel*
Alan (Zaoxing) Liu, *Boston University*
Grace Liu, *NYU Shanghai*
Jay Lorch, *Microsoft Research*
Harsha Madhyastha, *University of Michigan*
Morley Z. Mao, *University of Michigan*
James Mickens, *Harvard University*
Radhika Mittal, *University of Illinois at Urbana–Champaign*
Jayashree Mohan, *Microsoft Research India*
Iqbal Mohomed, *Samsung AI Center Toronto*
Shuai Mu, *Stony Brook University*
Rajalakshmi Nandakumar, *Cornell Tech*
Srinivas Narayana, *Rutgers University*
Ravi Netravali, *Princeton University*
Amy Ousterhout, *University of California, Berkeley*
Aurojit Panda, *New York University*
Peter Pietzuch, *Imperial College London*
Sanjay Rao, *Purdue University*
Jen Rexford, *Princeton University*
Nirupam Roy, *University of Maryland, College Park*
Ahmed Saeed, *Georgia Institute of Technology*
Raja Sambasivan, *Tufts University*
Stefan Schmid, *Technische Universität Berlin*
Aaron Schulman, *University of California, San Diego*
Siddhartha Sen, *Microsoft Research*
Srinivasan Seshan, *Carnegie Mellon University*
Muhammad Shahbaz, *Purdue University*
Rachee Singh, *Microsoft Research*
Dimitrios Skarlatis, *Carnegie Mellon University*
Alex Snoeren, *University of California, San Diego*
Brent Stephens, *University of Utah*
Mina Tahmasbi, *Cornell University*
Amy Tai, *Google*
Doug Terry, *Amazon*
Amin Vahdat, *Google*
Hakim Weatherspoon, *Cornell University*
Michael Wei, *VMware Research*
John Wilkes, *Google*
Keith Winstein, *Stanford University*
Yiting Xia, *Max Planck Institute for Informatics*
Tianyin Xu, *University of Illinois at Urbana–Champaign*
Neeraja Yadwadkar, *The University of Texas at Austin*
Francis Yan, *Microsoft Research*
Ellen Zegura, *Georgia Institute of Technology*
Ennan Zhai, *Alibaba*
Ying Zhang, *Meta*
Ben Zhao, *University of Chicago*
Zhizhen Zhong, *Massachusetts Institute of Technology*
Danyang Zhuo, *Duke University*

Poster Session Co-Chairs
Soudeh Ghorbani, *Johns Hopkins University*
Francis Yan, *Microsoft Research*

Test of Time Awards Committee
Aditya Akella, *University of Wisconsin–Madison*
Sujata Banerjee, *VMware Research*
Ranjita Bhagwan, *Microsoft Research India*
Jon Howell, *VMware Research*
James Mickens, *Harvard University*
Amar Phanishayee, *Microsoft Research*
George Porter, *University of California, San Diego*
Vyas Sekar, *Carnegie Mellon University*
Minlan Yu, *Harvard University*

Steering Committee
Aditya Akella, *University of Wisconsin–Madison*
Sujata Banerjee, *VMware Research*
Ranjita Bhagwan, *Microsoft Research India*
Casey Henderson, *USENIX Association*
Jon Howell, *VMware Research*
Arvind Krishnamurthy, *University of Washington*
Jay Lorch, *Microsoft Research*
James Mickens, *Harvard University*
Amar Phanishayee, *Microsoft Research*
George Porter, *University of California, San Diego*
Vyas Sekar, *Carnegie Mellon University*
Renata Teixeira, *Netflix*

External Reviewers

Vamsi Addanki
Anirudh Badam
Sujata Banerjee
Michael Barrow
Theophilus Benson
Jeremias Blendin
Vijay Chidambaram
Asaf Cidon
Angela Demke Brown
Fahad Dogar
Rodrigo Fonseca
Phillipa Gill
Brighten Godfrey
Ramesh Govindan
Kurtis Heimerl
Kyle Jamieson
Anuj Kalia
Ana Klimovic
Ana Klimovic
Morten Konggaard Schou
Yanfang Le
Christos Liaskos
Jonathan Mace
Georgios Nikolaidis
KyoungSoo Park
Chunyi Peng
Chunyi Peng
Ben Pfaff
George Porter
Costin Raiciu
Robert Ricci
Amedeo Sapio
Michael Schapira
Malte Schwarzkopf
Marco Serafini
Elahe Soltanaghai
Laurent Vanbever
Deepak Vasisht
Shivaram Venkataraman
Ymir Vigfusson
Jia Wang
Walter Willinger
Michelle X. Yeo
Yiying Zhang
Lin Zhong

Message from the NSDI '23 Program Co-Chairs

Welcome to NSDI '23! This year marks the 20th anniversary of the NSDI conference. In these two decades, networked systems have transformed the way that we live, work, and interact with one another. NSDI papers have spearheaded this revolution, providing many of the key technological advances behind industries such as Cloud Computing, Big Data, Software-Defined Networks, and more. With this latest iteration of NSDI, we hope to extend our community's track record of enabling and accelerating seismic shifts in computing via foundational research.

NSDI '23 received 560 submissions across two deadlines (272 in the Spring and 288 in the Fall), an increase of 40% from NSDI '22. To handle this record number of submissions, we assembled a Program Committee of 99 experts from academia and industry. The reviewing process included two rounds of double-blind review, an online discussion phase, and a two-day online PC meeting for each of the two deadlines. A total of 96 papers were accepted, resulting in an acceptance rate of 17%.

We thank our Program Committee members, who wrote over 1.6 million words of thoughtful, high-quality feedback across 2172 reviews, and discussed the papers extensively online and during the PC meetings. Many thanks to our poster chairs, Francis Yan and Soudeh Ghorbani, for bringing back the poster session to NSDI after a three-year hiatus. We thank our stand-in conflict PC chairs: Ben Y. Zhao, Siddhartha Sen, Indranil Gupta, and Katerina Argyraki. We would also like to thank Ellen Zegura, Rebecca Isaacs, and Matthew Caesar for helping us select the Best Paper award winners this year; and Aditya Akella, Sujata Banerjee, Ranjita Bhagwan, Jon Howell, James Mickens, Amar Phanishayee, George Porter, Vyas Sekar, and Minlan Yu for serving on the Test-of-Time awards committee. We are also grateful to Amar Phanishayee, Vyas Sekar, Arvind Krishnamurthy, Jay Lorch, Aditya Akella, and the rest of the NSDI Steering Committee for their advice and insight from running past NSDI instances. We would like to thank Sudarsanan Rajasekaran of MIT, who helped us immensely with the logistics of the PC meetings. We also thank Casey Henderson, Jasmine Murcia, Ginny Staubach, Jessica Kim, Sarah TerHune, Heidi Sherwood, Liz Markel, Camille Mulligan, Cathy Bergman, Nicole Santiago, Olivia Vernetti, Arnold Gatilao, Mo Moreno, and the rest of the USENIX staff for all their hard work behind the scenes. Finally, we would like to thank all the authors for submitting their best work to NSDI.

We look forward to seeing you all in Boston for the 20th iteration of NSDI!

Mahesh Balakrishnan, *Confluent*
Manya Ghobadi, *Massachusetts Institute of Technology*
NSDI '23 Program Co-Chairs

20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)

April 17–19, 2023
Boston, MA, USA

Monday, April 17

RDMA

SRNIC: A Scalable Architecture for RDMA NICs 1
Zilong Wang, *Hong Kong University of Science and Technology;* Layong Luo and Qingsong Ning, *ByteDance;* Chaoliang Zeng, Wenxue Li, and Xinchen Wan, *Hong Kong University of Science and Technology;* Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, and Tao Ding, *ByteDance;* Kai Chen, *Hong Kong University of Science and Technology;* Chuanxiong Guo

Hostping: Diagnosing Intra-host Network Bottlenecks in RDMA Servers. 15
Kefei Liu, *BUPT;* Zhuo Jiang, *ByteDance Inc.;* Jiao Zhang, *BUPT and Purple Mountain Laboratories;* Haoran Wei, *BUPT and ByteDance Inc.;* Xiaolong Zhong, *BUPT;* Lizhuang Tan, *ByteDance Inc.;* Tian Pan and Tao Huang, *BUPT and Purple Mountain Laboratories*

Understanding RDMA Microarchitecture Resources for Performance Isolation 31
Xinhao Kong and Jingrong Chen, *Duke University;* Wei Bai, *Microsoft;* Yechen Xu, *Shanghai Jiao Tong University;* Mahmoud Elhaddad, Shachar Raindel, and Jitendra Padhye, *Microsoft;* Alvin R. Lebeck and Danyang Zhuo, *Duke University*

Empowering Azure Storage with RDMA. 49
Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill, *Microsoft*

Learning with GPUs

Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. 69
Bingyang Wu and Zili Zhang, *Peking University;* Zhihao Bai, *Johns Hopkins University;* Xuanzhe Liu and Xin Jin, *Peking University*

ARK: GPU-driven Code Execution for Distributed Deep Learning 87
Changho Hwang, *KAIST, Microsoft Research;* KyoungSoo Park, *KAIST;* Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong, *Microsoft Research*

BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing 103
Tianfeng Liu, *Tsinghua University, Zhongguancun Laboratory, ByteDance;* Yangrui Chen, *The University of Hong Kong, ByteDance;* Dan Li, *Tsinghua University, Zhongguancun Laboratory;* Chuan Wu, *The University of Hong Kong;* Yibo Zhu, Jun He, and Yanghua Peng, *ByteDance;* Hongzheng Chen, *ByteDance, Cornell University;* Hongzhi Chen and Chuanxiong Guo, *ByteDance*

Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training. 119
Jie You, Jae-Won Chung, and Mosharaf Chowdhury, *University of Michigan*

RPC and Remote Memory

Remote Procedure Call as a Managed System Service 141
Jingrong Chen, Yongji Wu, and Shihan Lin, *Duke University;* Yechen Xu, *Shanghai Jiao Tong University;* Xinhao Kong, *Duke University;* Thomas Anderson, *University of Washington;* Matthew Lentz, Xiaowei Yang, and Danyang Zhuo, *Duke University*

Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory 161
Chenxi Wang, Yifan Qiao, Haoran Ma, and Shi Liu, *UCLA;* Yiying Zhang, *UCSD;* Wenguang Chen, *Tsinghua University;* Ravi Netravali, *Princeton University;* Miryung Kim and Guoqing Harry Xu, *UCLA*

Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony . . . 181
Yifan Qiao and Chenxi Wang, *UCLA;* Zhenyuan Ruan and Adam Belay, *MIT CSAIL;* Qingda Lu, *Alibaba Group;* Yiying Zhang, *UCSD;* Miryung Kim and Guoqing Harry Xu, *UCLA*

NetRPC: Enabling In-Network Computation in Remote Procedure Calls 199
Bohan Zhao, *Tsinghua University;* Wenfei Wu, *Peking University;* Wei Xu, *Tsinghua Univesity*

Congestion Control

Bolt: Sub-RTT Congestion Control for Ultra-Low Latency 219
Serhat Arslan, *Stanford University;* Yuliang Li, Gautam Kumar, and Nandita Dukkipati, *Google LLC*

Understanding the impact of host networking elements on traffic bursts 237
Erfan Sharafzadeh and Sepehr Abdous, *Johns Hopkins University;* Soudeh Ghorbani, *Johns Hopkins University and Meta*

Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT 255
Weitao Wang, *Google LLC and Rice University;* Masoud Moshref, Yuliang Li, and Gautam Kumar, *Google LLC;* T. S. Eugene Ng, *Rice University;* Neal Cardwell and Nandita Dukkipati, *Google LLC*

Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery 275
Taehyun Kim and Deondre Martin Ng, *KAIST;* Junzhi Gong, *Harvard University;* Youngjin Kwon, *KAIST;* Minlan Yu, *Harvard University;* KyoungSoo Park, *KAIST*

Distributed Systems

Hydra: Serialization-Free Network Ordering for Strongly Consistent Distributed Applications 293
Inho Choi, *National University of Singapore;* Ellis Michael, *University of Washington;* Yunfan Li, *National University of Singapore;* Dan R. K. Ports, *Microsoft Research;* Jialin Li, *National University of Singapore*

The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems 321
Lei Zhang, *Emory University and Princeton University;* Zhiqiang Xie and Vaastav Anand, *Max Planck Institute for Software Systems;* Ymir Vigfusson, *Emory University;* Jonathan Mace, *Max Planck Institute for Software Systems*

DiSh: Dynamic Shell-Script Distribution 341
Tammam Mustafa, *MIT;* Konstantinos Kallas, *University of Pennsylvania;* Pratyush Das, *Purdue University;* Nikos Vasilakis, *Brown University*

Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication 357
Mohammadreza Alimadadi and Hieu Mai, *Stony Brook University;* Shenghsun Cho, *Microsoft;* Michael Ferdman, Peter Milder, and Shuai Mu, *Stony Brook University*

Wireless

LeakyScatter: A Frequency-Agile Directional Backscatter Network Above 100 GHz 375
Atsutse Kludze and Yasaman Ghasempour, *Princeton University*

RF-Bouncer: A Programmable Dual-band Metasurface for Sub-6 Wireless Networks 389
Xinyi Li, Chao Feng, Xiaojing Wang, and Yangfan Zhang, *Northwest University;* Yaxiong Xie, *University at Buffalo SUNY;* Xiaojiang Chen, *Northwest University*

Scalable Distributed Massive MIMO Baseband Processing 405
Junzhi Gong, *Harvard University;* Anuj Kalia, *Microsoft;* Minlan Yu, *Harvard University*

DChannel: Accelerating Mobile Applications With Parallel High-bandwidth and Low-latency Channels 419
William Sentosa, *University of Illinois Urbana-Champaign;* Balakrishnan Chandrasekaran, *Vrije Universiteit Amsterdam;* P. Brighten Godfrey, *University of Illinois Urbana-Champaign and VMware;* Haitham Hassanieh, *EPFL;* Bruce Maggs, *Duke University and Emerald Innovations*

Cloud

SkyPilot: An Intercloud Broker for Sky Computing 437
Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, and Gautam Mittal, *UC Berkeley;* Scott Shenker, *UC Berkeley and ICSI;* Ion Stoica, *UC Berkeley*

Unlocking unallocated cloud capacity for long, uninterruptible workloads 457
Anup Agarwal, *Carnegie Mellon University;* Shadi Noghabi, *Microsoft Research;* Íñigo Goiri, *Azure Systems Research;* Srinivasan Seshan, *Carnegie Mellon University;* Anirudh Badam, *Microsoft Research*

Invisinets: Removing Networking from Cloud Networks 479
Sarah McClure and Zeke Medley, *UC Berkeley;* Deepak Bansal and Karthick Jayaraman, *Microsoft;* Ashok Narayanan, *Google;* Jitendra Padhye, *Microsoft;* Sylvia Ratnasamy, *UC Berkeley and Google;* Anees Shaikh, *Google;* Rishabh Tewari, *Microsoft*

Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. 497
John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, and Yifan Qiao, *UCLA;* Zhihao Jia, *CMU;* Minjia Zhang, *Microsoft Research;* Ravi Netravali, *Princeton University;* Guoqing Harry Xu, *UCLA*

Internet-Scale Networks

OneWAN is better than two: Unifying a split WAN architecture 515
Umesh Krishnaswamy, *Microsoft;* Rachee Singh, *Microsoft and Cornell University;* Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, Himanshu Raj, Luis Irun-Briz, Jamie Gaudette, and Erica Lan, *Microsoft*

RHINE: Robust and High-performance Internet Naming with E2E Authenticity 531
Huayi Duan, Rubén Fischer, Jie Lou, Si Liu, David Basin, and Adrian Perrig, *ETH Zürich*

Enabling Users to Control their Internet 555
Ammar Tahir and Radhika Mittal, *University of Illinois at Urbana-Champaign*

***x*BGP: Faster Innovation in Routing Protocols 575**
Thomas Wirtgen, Tom Rousseaux, Quentin De Coninck, and Nicolas Rybowski, *ICTEAM, UCLouvain;* Randy Bush, *Internet Initiative Japan & Arrcus, Inc;* Laurent Vanbever, *NSG, ETH Zürich;* Axel Legay and Olivier Bonaventure, *ICTEAM, UCLouvain*

Tuesday, April 18

Synthesis and Formal Methods

TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches 593
Aashaka Shah, *University of Texas at Austin;* Vijay Chidambaram, *University of Texas at Austin and VMware Research;* Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi, *Microsoft Research;* Rachee Singh, *Microsoft and Cornell University*

Synthesizing Runtime Programmable Switch Updates 613
Yiming Qiu, *Rice University;* Ryan Beckett, *Microsoft;* Ang Chen, *Rice University*

Practical Intent-driven Routing Configuration Synthesis 629
Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, and Walid Taha, *Meta;* Minlan Yu, *Harvard University;* Jelena Mirkovic, *University of Southern California Information Sciences Institute*

Formal Methods for Network Performance Analysis 645
Mina Tahmasbi Arashloo, *University of Waterloo;* Ryan Beckett, *Microsoft Research;* Rachit Agarwal, *Cornell University*

Data Centers

Flattened Clos: Designing High-performance Deadlock-free Expander Data Center Networks Using Graph Contraction 663
Shizhen Zhao, Qizhou Zhang, Peirui Cao, Xiao Zhang, and Xinbing Wang, *Shanghai Jiao Tong University;* Chenghu Zhou, *Shanghai Jiao Tong University and Chinese Academy of Sciences*

Scalable Tail Latency Estimation for Data Center Networks. 685
Kevin Zhao, *University of Washington;* Prateesh Goyal, *Microsoft Research;* Mohammad Alizadeh, *MIT CSAIL;* Thomas E. Anderson, *University of Washington*

***Shockwave*: Fair and Efficient Cluster Scheduling for Dynamic Adaptation in Machine Learning 703**
Pengfei Zheng and Rui Pan, *University of Wisconsin-Madison;* Tarannum Khan, *The University of Texas at Austin;* Shivaram Venkataraman, *University of Wisconsin-Madison;* Aditya Akella, *The University of Texas at Austin*

Protego: Overload Control for Applications with Unpredictable Lock Contention. 725
Inho Cho, *MIT CSAIL;* Ahmed Saeed, *Georgia Tech;* Seo Jin Park, Mohammad Alizadeh, and Adam Belay, *MIT CSAIL*

Systems for Learning

TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs. 739
Weiyang Wang, Moein Khazraee, Zhizhen Zhong, and Manya Ghobadi, *Massachusetts Institute of Technology;* Zhihao Jia, *Meta and CMU;* Dheevatsa Mudigere and Ying Zhang, *Meta;* Anthony Kewitsch, *Telescent*

ModelKeeper: Accelerating DNN Training via Automated Training Warmup 769
Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury, *University of Michigan*

Shepherd: Serving DNNs in the Wild 787
Hong Zhang, *University of Waterloo;* Yupeng Tang and Anurag Khandelwal, *Yale University;* Ion Stoica, *UC Berkeley*

Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using Syndicate 809
Kshiteej Mahajan, *University of Wisconsin - Madison;* Ching-Hsiang Chu and Srinivas Sridharan, *Facebook;* Aditya Akella, *UT Austin*

Privacy and Security

Addax: A fast, private, and accountable ad exchange infrastructure 825
Ke Zhong, Yiping Ma, and Yifeng Mao, *University of Pennsylvania;* Sebastian Angel, *University of Pennsylvania & Microsoft Research*

SPEEDEX: A Scalable, Parallelizable, and Economically Efficient Decentralized EXchange 849
Geoffrey Ramseyer, Ashish Goel, and David Mazières, *Stanford University*

Boomerang: Metadata-Private Messaging under Hardware Trust 877
Peipei Jiang, *Wuhan University and City University of Hong Kong;* Qian Wang and Jianhao Cheng, *Wuhan University;* Cong Wang, *City University of Hong Kong;* Lei Xu, *Nanjing University of Science and Technology;* Xinyu Wang, *Tencent Inc.;* Yihao Wu and Xiaoyuan Li, *Wuhan University;* Kui Ren, *Zhejiang University*

Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies 901
James Lovejoy, *Federal Reserve Bank of Boston;* Madars Virza and Cory Fields, *MIT Media Lab;* Kevin Karwaski and Anders Brownworth, *Federal Reserve Bank of Boston;* Neha Narula, *MIT Media Lab*

Video

RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics 917
Mehrdad Khani, *MIT CSAIL and Microsoft;* Ganesh Ananthanarayanan and Kevin Hsieh, *Microsoft;* Junchen Jiang, *University of Chicago;* Ravi Netravali, *Princeton University;* Yuanchao Shu, *Zhejiang University;* Mohammad Alizadeh, *MIT CSAIL;* Victor Bahl, *Microsoft*

Boggart: Towards General-Purpose Acceleration of Retrospective Video Analytics 933
Neil Agarwal and Ravi Netravali, *Princeton University*

Tambur: Efficient loss recovery for videoconferencing via streaming codes 953
Michael Rudow, *Carnegie Mellon University;* Francis Y. Yan, *Microsoft Research;* Abhishek Kumar, *Carnegie Mellon University;* Ganesh Ananthanarayanan and Martin Ellis, *Microsoft;* K.V. Rashmi, *Carnegie Mellon University*

Gemel: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge. 973
Arthi Padmanabhan, *UCLA;* Neil Agarwal, *Princeton University;* Anand Iyer and Ganesh Ananthanarayanan, *Microsoft Research;* Yuanchao Shu, *Zhejiang University;* Nikolaos Karianakis, *Microsoft Research;* Guoqing Harry Xu, *UCLA;* Ravi Netravali, *Princeton University*

Data

Fast, Approximate Vector Queries on Very Large Unstructured Datasets 995
Zili Zhang and Chao Jin, *Peking University;* Linpeng Tang, *Moqi;* Xuanzhe Liu and Xin Jin, *Peking University*

Arya: Arbitrary Graph Pattern Mining with Decomposition-based Sampling 1013
Zeying Zhu, *Boston University;* Kan Wu, *University of Wisconsin-Madison;* Zaoxing Liu, *Boston University*

Secrecy: Secure collaborative analytics in untrusted clouds 1031
John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia, *Boston University*

FLASH: Towards a High-performance Hardware Acceleration Architecture for Cross-silo Federated Learning. 1057
Junxue Zhang and Xiaodian Cheng, *iSINGLab at Hong Kong University of Science and Technology and Clustar;* Wei Wang, *Clustar;* Liu Yang, *iSINGLab at Hong Kong University of Science and Technology and Clustar;* Jinbin Hu and Kai Chen, *iSINGLab at Hong Kong University of Science and Technology*

Making Systems Learn

On Modular Learning of Distributed Systems for Predicting End-to-End Latency 1081
Chieh-Jan Mike Liang, *Microsoft Research;* Zilin Fang, *Carnegie Mellon University;* Yuqing Xie, *Tsinghua University;* Fan Yang, *Microsoft Research;* Zhao Lucis Li, *University of Science and Technology of China;* Li Lyna Zhang, Mao Yang, and Lidong Zhou, *Microsoft Research*

SelfTune: Tuning Cluster Managers. 1097
Ajaykrishna Karthikeyan and Nagarajan Natarajan, *Microsoft Research;* Gagan Somashekar, *Stony Brook University;* Lei Zhao, *Microsoft;* Ranjita Bhagwan, *Microsoft Research;* Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal, *Microsoft*

CausalSim: A Causal Framework for Unbiased Trace-Driven Simulation 1115
Abdullah Alomar, Pouya Hamadanian, Arash Nasr-Esfahany, Anish Agarwal, Mohammad Alizadeh, and Devavrat Shah, *MIT*

HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network 1149
Zhenyu Song, *Princeton University;* Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi, *Google*

IoT Networks

OpenLoRa: Validating LoRa Implementations through an Extensible and Open-sourced Framework 1165
Manan Mishra, Daniel Koch, Muhammad Osama Shahid, and Bhuvana Krishnaswamy, *University of Wisconsin-Madison;* Krishna Chintalapudi, *Microsoft Research;* Suman Banerjee, *University of Wisconsin-Madison*

VeCare: Statistical Acoustic Sensing for Automotive In-Cabin Monitoring 1185
Yi Zhang, *The University of Hong Kong and Tsinghua University;* Weiying Hou, *The University of Hong Kong;* Zheng Yang, *Tsinghua University;* Chenshu Wu, *The University of Hong Kong*

SlimWiFi: Ultra-Low-Power IoT Radio Architecture Enabled by Asymmetric Communication 1201
Renjie Zhao, *University of California San Diego;* Kejia Wang, *Baylor University;* Kai Zheng and Xinyu Zhang, *University of California San Diego;* Vincent Leung, *Baylor University*

SLNet: A Spectrogram Learning Neural Network for Deep Wireless Sensing 1221
Zheng Yang and Yi Zhang, *Tsinghua University;* Kun Qian, *University of California San Diego;* Chenshu Wu, *The University of Hong Kong*

Wednesday, April 19

Programming the Network

A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches 1237
Mariano Scazzariello and Tommaso Caiazzi, *KTH Royal Institute of Technology and Roma Tre University;* Hamid Ghasemirahni, *KTH Royal Institute of Technology;* Tom Barbette, *UCLouvain;* Dejan Kostić and Marco Chiesa, *KTH Royal Institute of Technology*

ExoPlane: An Operating System for On-Rack Switch Resource Augmentation 1257
Daehyeok Kim, *Microsoft and University of Texas at Austin;* Vyas Sekar and Srinivasan Seshan, *Carnegie Mellon University*

Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches 1273
Hun Namkung, *Carnegie Mellon University;* Zaoxing Liu, *Boston University;* Daehyeok Kim, *Microsoft Research;* Vyas Sekar and Peter Steenkiste, *Carnegie Mellon University*

RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs 1293
Jiaxin Lin, Adney Cardoza, Tarannum Khan, and Yeonju Ro, *UT Austin;* Brent E. Stephens, *University of Utah;* Hassan Wassel, *Google;* Aditya Akella, *UT Austin*

Alternative Networks

StarryNet: Empowering Researchers to Evaluate Futuristic Integrated Space and Terrestrial Networks 1309
Zeqi Lai and Hewu Li, *Tsinghua University and Zhongguancun Laboratory;* Yangtao Deng, *Tsinghua University;* Qian Wu, Jun Liu, and Yuanjie Li, *Tsinghua University and Zhongguancun Laboratory;* Jihao Li, Lixin Liu, and Weisen Liu, *Tsinghua University;* Jianping Wu, *Tsinghua University and Zhongguancun Laboratory*

Polycorn: Data-driven Cross-layer Multipath Networking for High-speed Railway through Composable Schedulerlets 1325
Yunzhe Ni, *Peking University;* Feng Qian, *University of Minnesota – Twin Cities;* Taide Liu, Yihua Cheng, Zhiyao Ma, and Jing Wang, *Peking University;* Zhongfeng Wang, *China Railway Gecent Technology Co., Ltd;* Gang Huang and Xuanzhe Liu, *Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University;* Chenren Xu, *Zhongguancun Laboratory and Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University*

Augmenting Augmented Reality with Non-Line-of-Sight Perception 1341
Tara Boroushaki, Maisy Lam, and Laura Dodds, *Massachusetts Institute of Technology;* Aline Eid, *Massachusetts Institute of Technology and University of Michigan;* Fadel Adib, *Massachusetts Institute of Technology*

Acoustic Sensing and Communication Using Metasurface 1359
Yongzhao Zhang, Yezhou Wang, and Lanqing Yang, *Shanghai Jiao Tong University;* Mei Wang, *UT Austin;* Yi-Chao Chen, *Shanghai Jiao Tong University and Microsoft Research Asia;* Lili Qiu, *UT Austin and Microsoft Research Asia;* Yihong Liu, *University of Glasgow;* Guangtao Xue and Jiadi Yu, *Shanghai Jiao Tong University*

Performance

Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays 1375
Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica, *University of California, Berkeley*

Electrode: Accelerating Distributed Protocols with eBPF 1391
Yang Zhou, *Harvard University;* Zezhou Wang, *Peking University;* Sowmya Dharanipragada, *Cornell University;* Minlan Yu, *Harvard University*

Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes 1409
Zhenyuan Ruan and Seo Jin Park, *MIT CSAIL;* Marcos K. Aguilera, *VMware Research;* Adam Belay, *MIT CSAIL;* Malte Schwarzkopf, *Brown University*

Enabling High Quality Real-Time Communications with Adaptive Frame-Rate 1429
Zili Meng, *Tsinghua University and Tencent Inc.;* Tingfeng Wang, *Tsinghua University, Tencent Inc., and Beijing University of Posts and Telecommunications;* Yixin Shen, *Tsinghua University;* Bo Wang and Mingwei Xu, *Tsinghua University and Zhongguancun Laboratory;* Rui Han and Honghao Liu, *Tencent Inc.;* Venkat Arun, *Massachusetts Institute of Technology;* Hongxin Hu, *University at Buffalo, SUNY;* Xue Wei, *Tencent Inc.*

Serverless and Network Functions

LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed 1451
Hao Li and Yihan Dang, *Xi'an Jiaotong University;* Guangda Sun, *Xi'an Jiaotong University and National University of Singapore;* Guyue Liu, *New York University Shanghai;* Danfeng Shan and Peng Zhang, *Xi'an Jiaotong University*

Disaggregating Stateful Network Functions 1469
Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, and James Grantham, *Microsoft;* Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, and Balakrishnan Raman, *AMD Pensando;* Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula, *Microsoft*

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing 1489
Minchen Yu, *Hong Kong University of Science and Technology;* Tingjia Cao, *University of Wisconsin-Madison;* Wei Wang, *Hong Kong University of Science and Technology;* Ruichuan Chen, *Nokia Bell Labs*

Doing More with Less: Orchestrating Serverless Applications without an Orchestrator 1505
David H. Liu and Amit Levy, *Princeton University;* Shadi Noghabi and Sebastian Burckhardt, *Microsoft Research*

Real Networks

Enhancing Global Network Monitoring with *Magnifier* 1521
Tobias Bühler and Romain Jacob, *ETH Zürich;* Ingmar Poese, *BENOCS;* Laurent Vanbever, *ETH Zürich*

NetPanel: Traffic Measurement of Exchange Online Service 1541
Yu Chen, *Microsoft 365, China;* Liqun Li and Yu Kang, *Microsoft Research, China;* Boyang Zheng, Yehan Wang, More Zhou, Yuchao Dai, and Zhenguo Yang, *Microsoft 365, China;* Brad Rutkowski and Jeff Mealiffe, *Microsoft 365, USA;* Qingwei Lin, *Microsoft Research, China*

DOTE: Rethinking (Predictive) WAN Traffic Engineering 1557
Yarin Perry, *Hebrew University of Jerusalem;* Felipe Vieira Frujeri, *Microsoft Research;* Chaim Hoch, *Hebrew University of Jerusalem;* Srikanth Kandula and Ishai Menache, *Microsoft Research;* Michael Schapira, *Hebrew University of Jerusalem;* Aviv Tamar, *Technion*

Dashlet: Taming Swipe Uncertainty for Robust Short Video Streaming 1583
Zhuqi Li, Yaxiong Xie, Ravi Netravali, and Kyle Jamieson, *Princeton University*

Cellular

CellDAM: User-Space, Rootless Detection and Mitigation for 5G Data Plane 1601
Zhaowei Tan, Jinghao Zhao, Boyan Ding, and Songwu Lu, *University of California, Los Angeles*

LOCA: A Location-Oblivious Cellular Architecture 1621
Zhihong Luo, Silvery Fu, and Natacha Crooks, *UC Berkeley;* Shaddi Hasan, *Virginia Tech;* Christian Maciocco, *Intel;* Sylvia Ratnasamy, *UC Berkeley;* Scott Shenker, *UC Berkeley and ICSI*

mmWall: A Steerable, Transflective Metamaterial Surface for NextG mmWave Networks 1647
Kun Woo Cho, *Princeton University;* Mohammad H. Mazaheri, *UCLA;* Jeremy Gummeson, *University of Massachusetts Amherst;* Omid Abari, *UCLA;* Kyle Jamieson, *Princeton University*

Building Flexible, Low-Cost Wireless Access Networks With Magma 1667
Shaddi Hasan, *Virginia Tech;* Amar Padmanabhan, *Databricks;* Bruce Davie, *Systems Approach;* Jennifer Rexford, *Princeton University;* Ulas Kozat, Hunter Gatewood, Shruti Sanadhya, Nick Yurchenko, Tariq Al-Khasib, Oriol Batalla, Marie Bremner, Andrei Lee, Evgeniy Makeev, Scott Moeller, Alex Rodriguez, Pravin Shelar, Karthik Subraveti, Sudarshan Kandi, Alejandro Xoconostle, and Praveen Kumar Ramakrishnan, *Meta;* Xiaochen Tian, *Indepenent;* Anoop Tomar, *Meta*

Testing

LinkLab 2.0: A Multi-tenant Programmable IoT Testbed for Experimentation with Edge-Cloud Integration ... 1683
Wei Dong, Borui Li, Haoyu Li, Hao Wu, Kaijie Gong, Wenzhao Zhang, and Yi Gao, *Zhejiang University*

Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker 1701
Yinfang Chen and Xudong Sun, *University of Illinois at Urbana-Champaign;* Suman Nath, *Microsoft Research;* Ze Yang and Tianyin Xu, *University of Illinois at Urbana-Champaign*

Test Coverage for Network Configurations 1717
Xieyang Xu and Weixin Deng, *University of Washington;* Ryan Beckett, *Microsoft;* Ratul Mahajan, *University of Washington;* David Walker, *Princeton University*

Norma: Towards Practical Network Load Testing 1733
Yanqing Chen, *State Key Laboratory for Novel Software Technology, Nanjing University and Alibaba Group;* Bingchuan Tian, *Alibaba Group;* Chen Tian, *State Key Laboratory for Novel Software Technology, Nanjing University;* Li Dai, Yu Zhou, Mengjing Ma, and Ming Tang, *Alibaba Group;* Hao Zheng, Zhewen Yang, and Guihai Chen, *State Key Laboratory for Novel Software Technology, Nanjing University;* Dennis Cai and Ennan Zhai, *Alibaba Group*

Physical Layer

μMote: Enabling Passive Chirp De-spreading and μW-level Long-Range Downlink for Backscatter Devices 1751
Yihang Song and Li Lu, *University of Electronic Science and Technology of China;* Jiliang Wang, *Tsinghua University;* Chong Zhang, Hui Zheng, and Shen Yang, *University of Electronic Science and Technology of China;* Jinsong Han, *Zhejiang University;* Jian Li, *University of Electronic Science and Technology of China*

Channel-Aware 5G RAN Slicing with Customizable Schedulers 1767
Yongzhou Chen and Ruihao Yao, *UIUC;* Haitham Hassanieh, *EPFL;* Radhika Mittal, *UIUC*

RF-Chord: Towards Deployable RFID Localization System for Logistic Networks.....1783
Bo Liang, *Peking University and Alibaba Group;* Purui Wang, *Massachusetts Institute of Technology;* Renjie Zhao, *University of California San Diego;* Heyu Guo, *Peking University;* Pengyu Zhang and Junchen Guo, *Alibaba Group;* Shunmin Zhu, *Tsinghua University and Alibaba Group;* Hongqiang Harry Liu, *Alibaba Group;* Xinyu Zhang, *University of California San Diego;* Chenren Xu, *Peking University, Zhongguancun Laboratory, and Key Laboratory of High Confidence Software Technologies, Ministry of Education* (*PKU*)

Exploring Practical Vulnerabilities of Machine Learning-based Wireless Systems 1801
Zikun Liu, Changming Xu, and Emerson Sie, *University of Illinois Urbana-Champaign;* Gagandeep Singh, *University of Illinois Urbana-Champaign and VMware Research;* Deepak Vasisht, *University of Illinois Urbana-Champaign*

SRNIC: A Scalable Architecture for RDMA NICs

Zilong Wang[1*] Layong Luo[2] Qingsong Ning[2] Chaoliang Zeng[1*] Wenxue Li[1] Xinchen Wan[1*]
Peng Xie[2] Tao Feng[2] Ke Cheng[2] Xiongfei Geng[2] Tianhao Wang[2] Weicheng Ling[2]
Kejia Huo[2] Pingbo An[2] Kui Ji[2] Shideng Zhang[2] Bin Xu[2] Ruiqing Feng[2] Tao Ding[2]
Kai Chen[1] Chuanxiong Guo[3]
[1]*Hong Kong University of Science and Technology* [2]*ByteDance* [3]*Unaffiliated*

Abstract

RDMA is expected to be highly scalable: to perform well in large-scale data center networks where packet losses are inevitable (*i.e.*, high network scalability), and to support a large number of performant connections per server (*i.e.*, high connection scalability). Commercial RoCEv2 NICs (RNICs) fall short on scalability as they rely on a lossless, limited-scale network fabric and support only a small number of performant connections. Recent work IRN improves the network scalability by relaxing the lossless network requirement, but the connection scalability issue remains unaddressed.

In this paper, we aim to address the connection scalability challenge, while maintaining high performance and low CPU overhead as commercial RNICs, and high network scalability as IRN, by designing SRNIC, a Scalable RDMA NIC architecture. Our key insight in SRNIC is that, on-chip data structures and their memory requirements in RNICs can be minimized with careful protocol and architecture co-designs to improve connection scalability. Guided by this insight, we analyze all data structures involved in an RDMA conceptual model, and remove them as many as possible with RDMA protocol header modifications and architectural innovations, including cache-free QP scheduler and memory-free selective repeat. We implement a fully functional SRNIC prototype using FPGA. Experiments show that, SRNIC achieves 10K performant connections on chip and outperforms commercial RNICs by 18x in terms of normalized connection scalability (*i.e.*, the number of performant connections per 1MB memory), while achieving 97 Gbps throughput and 3.3 µs latency with less than 5% CPU overhead, and maintaining high network scalability.

1 Introduction

Datacenter applications are increasingly driving the demands for high-speed networks, which are expected to provide high throughput, low latency, and low CPU overhead, with a large number of connections (*a.k.a.*, connection scalability), over a large-scale network (*a.k.a.*, network scalability). Specifically, bandwidth-intensive applications like distributed machine learning training [13, 23] and cloud storage [16, 18], require 100 Gbps and beyond network bandwidth between servers; online services like search [9, 15] and database [25, 29], demand low latency to minimize query response time; most applications desire a network stack with low CPU overhead to reserve as many CPU cores as possible for computations; cloud storage like Alibaba Pangu [18] requires a large number of performant connections per host to provide mesh communications between chunk servers and block servers; last but not the least, high-speed networks tend to be deployed at larger scale as their application footprints expand [19].

Remote Direct Memory Access (RDMA) is emerging as a popular high-speed networking technique, thanks to its high throughput, low latency and low CPU overhead provided by architectural innovations including kernel bypass and transport offload. With these advantages, RoCEv2 (RDMA over Converged Ethernet Version 2) is becoming the de-facto standard for high-speed networks in modern data centers [4, 42].

Despite high performance and low CPU overhead, commercial RoCEv2 NICs (RNICs) suffer from both network scalability and connection scalability issues. On one hand, the network scalability issue arises from PFC (Priority-based Flow Control) which is required by RDMA to implement a lossless network fabric. PFC brings issues such as head-of-line blocking, congestion spreading, occasional deadlocks, and PFC storms in large-scale clusters [18, 19, 21, 34, 42]. As a result, datacenter operators tend to restrict the PFC configurations within a small network scope (*e.g.*, a moderate cluster). On the other hand, the connection scalability issue is the phenomenon that RDMA performance drops dramatically when the number of connections (*a.k.a.*, queue pairs (QPs)) exceeds a certain small threshold (*e.g.*, 256) [24, 28, 39]. Although commercial RNICs are blackbox, the root cause of this performance collapse phenomenon is explained as cache misses due to context switch between connections [24].

[*] This work is done while Zilong Wang, Chaoliang Zeng, and Xinchen Wan are interns with ByteDance.

To improve network scalability of RNICs, existing work IRN [33] advocates lossy RDMA that eliminates PFC, by replacing go-back-N with more efficient selective repeat (SR). However, the introduction of SR is non-trivial: it adds some SR specific data structures and thus increases memory consumption. To reduce the on-chip memory overhead, IRN makes some RoCEv2 header extensions, but still requires 3-10% more memory than existing RNIC implementations. As a result, IRN achieves high network scalability but leaves the connection scalability issue unaddressed.

In this paper, we propose SRNIC, a Scalable RDMA NIC architecture to address the connection scalability issue, while preserving high performance and low CPU overhead inherited from transport offload as commercial RNICs, and maintaining high network scalability originated from lossy RDMA as IRN. The major insight of SRNIC is that, most on-chip data structures and their memory requirements in RNICs can be eliminated with careful protocol and architecture co-designs, and the connection scalability of RNICs could be, as a result, significantly improved. Guided by this insight, we examine the typical data flow in a lossy RDMA conceptual model (§3.1), analyze all the involved data structures, classify them into two categories: *common data structures* required by RDMA in general, and *selective repeat specific data structures* brought by lossy RDMA, and finally take customized optimization strategies to minimize these two types of data structures respectively to improve the connection scalability (§3.2).

In particular, the cache-free QP scheduler proposed in §4.3 optimizes common data structures for RDMA designs no matter whether the underlying network is lossy or lossless. The optimizations of RDMA header extensions and bitmap onloading introduced in §4.4 are for memory-free selective repeat, hence specific for lossy RDMA.

We have implemented a fully functional SRNIC prototype with FPGA (§5) and evaluated SRNIC's scalability and performance through the testbed and simulations. Experiments (§6) show that SRNIC achieves high connection scalability, while preserving high performance and low CPU overhead as commercial RNICs, and high network scalability as IRN. Specifically, SRNIC supports 10K[1] connections/QPs without performance degradation, which outperforms Mellanox RNIC CX-5 by 18x in terms of normalized connection scalability (*i.e.*, the number of performant connections per 1MB memory). Meanwhile, SRNIC achieves 97 Gbps line-rate throughput and 3.3 µs latency, with only 5% CPU overhead, which are comparable with Mellanox RNICs. In addition, SRNIC shows its high network scalability via high loss tolerance (3x higher goodput than Mellanox RNICs under 1% loss rate) and predictable performance in large-scale lossy networks.

As a summary, Figure 1 shows the design space of RDMA NICs and makes a comparative analysis between different solutions. Although all RDMA hardware solutions provide high throughput, low latency, and low CPU overhead via transport offload and kernel bypass, their scalability varies. Commercial RNICs suffer from both the network scalability issue caused by the troublesome PFC, and the connection scalability issue caused by unknown blackbox implementations. IRN revisits the network supports for RDMA, and eliminates the need of PFC by introducing selective repeat with 3-10% extra memory overhead. As a result, the network scalability is significantly improved, but the connection scalability is left unsolved. SRNIC leverages the lossy RDMA approach of IRN to improve network scalability, and further addresses the connection scalability issue with the design guiding principle: minimize the on-chip memory requirements of RNICs in a simple yet performant way. As a result, SRNIC achieves both high network scalability and connection scalability.

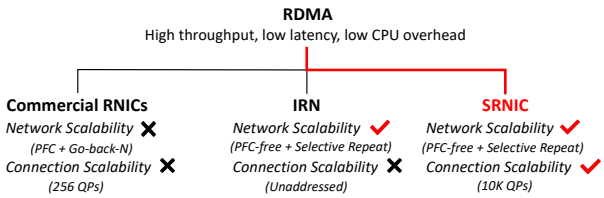


Figure 1: Design space of RDMA NICs.

This paper makes the following major contributions:

- We systematically study and quantify the memory requirements of RDMA NICs, by introducing an RDMA conceptual model (§3).
- We design SRNIC, a scalable and high-performance RDMA NIC architecture, that significantly improves the connection scalability, guided by an insight that the on-chip memory requirements in the conceptual model can be minimized with careful RDMA protocol modifications and architecture innovations, including cache-free QP scheduler and memory-free selective repeat (§4).
- We implement SRNIC using FPGA, with only 4.4 MB on-chip memory. The implementation achieves our design goals on scalability, performance, and CPU overhead (§5 and §6).

2 Background and Motivation

2.1 RDMA Overview

Unlike the traditional software transport TCP, RDMA is a hardware transport that implements the transport functionalities including congestion control and loss recovery entirely in NIC hardware, and provides kernel-bypass and zero-copy interfaces to the user applications. As a result, RDMA achieves high throughput, low latency, and low CPU overhead, compared with software transport TCP [42].

[1] Unless otherwise stated, K is 1024 in measuring the size of memory, data structures and messages, and 1000 in measuring the others.

RDMA was originally designed and simplified for lossless Infiniband [1]. To make RDMA work in Ethernet, RoCEv2 relies on PFC [22] to turn Ethernet into a lossless fabric. However, PFC brings management risks and network scalability challenges (*e.g.*, PFC storms and deadlocks) that affect the entire network's availability and also causes collateral damage to innocent flows due to head-of-line blocking [19, 42]. Besides, with PFC, the lossless network scale is also limited by the switch buffer size. Consequently, datacenters usually limit the scale of RDMA networks [18].

As the network scalability issue of RoCEv2 is mainly caused by PFC, IRN [33] takes the first step to rethink RDMA's network requirements, eliminates PFC and allows RDMA working well in lossy networks, by replacing the default lossy recovery mechanism go-back-N with more efficient selective repeat. However, it leaves the connection scalability challenge unsolved.

2.2 Connection Scalability Issue

Commercial RNICs face a well-known connection scalability issue [24, 27, 28, 39], *i.e.*, the RDMA performance drops significantly as the number of QPs increases beyond a small value (varies from 16 to 500 in different settings [28]). We demonstrate this issue using off-the-shelf commercial RNICs including Mellanox CX-5 and CX-6 [7, 8] with PFC enabled. As shown in Figure 2a, the aggregate throughput of Mellanox CX-6 drops 46% (from 97 to 52 Gbps) when the QP number increases from 128 to 16384, and there is no obvious improvement of connection scalability from CX-5 to CX-6.

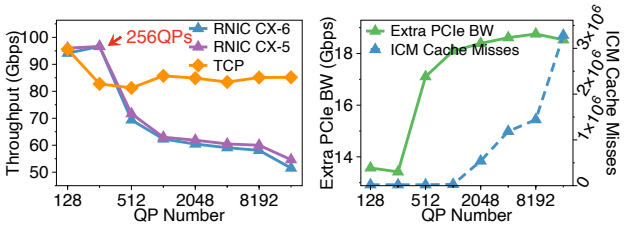
The root cause of RNIC's performance degradation is commonly explained as cache misses [24, 28, 38]. Commercial RNICs usually take a DRAM-free architecture, which does not have DRAM connected directly to the RNIC chip to reduce cost, power consumption, and area, but just has limited on-chip SRAM. As a result, RNICs can cache only a small number of QPs on chip, while storing the others in host memory. When the number of active QPs increases beyond the on-chip memory size, frequent cache misses and context switches between host memory and RNIC cause performance collapse. Our experiments in Figure 2b verify this in some sense. We observed significant extra PCIe bandwidth[2] and an increase in ICM cache miss[3] during the performance collapse. Both metrics reflect certain kinds of cache misses, causing extra PCIe traffic increase after 256 QPs.

Although on-chip SRAM is limited, it is abnormal in that the performance drops so early. Given the on-chip memory size and the QP Context (QPC) size for a QP, we can estimate the maximum number of performant QPs that could be supported without cache misses and performance collapse as:

$$max_QPs = \frac{memory_size}{sizeof(QPC)}. \quad (1)$$

[2]Extra PCIe throughput = PCIe throughput - network throughput.

[3]"ICM Cache Miss" is a counter provided by Mellanox Neohost tool [12].



(a) Aggregate throughput (b) ICM Cache misses and extra PCIe traffic

Figure 2: Connection scalability issue of current RNICs. Compared with TCP, the aggregate throughput of current RNICs collapses when the number of QPs exceeds 256.

Let's take Mellanox CX-5 as an example. Its on-chip memory size is ∼2 MB [24] and a QPC takes ∼375 B [24], so that the maximum number of performant QPs supported by CX-5 could be up to 5.6K (2 MB/375 B), which contradicts the fact shown in Figure 2a that CX-5 performance begins to collapse much earlier at 256 QPs. The contradiction implies that there is room to significantly improve the connection scalability.

Motivated by this contradiction, we systematically analyze the memory requirements of RNICs, and improve the connection scalability based on the insights derived from thorough memory analysis.

3 RNIC Memory Analysis

As commercial RNICs are blackbox, we are not able to use their micro-architectures as a reference. Instead, we leverage a lossy RDMA conceptual model with selective repeat to derive the involved data structures (§3.1). Then, we summarize and classify these data structures into two categories: *common data structures* required by RDMA in general, and *selective repeat specific data structures* brought by lossy RDMA, and discuss different optimization strategies to minimize them respectively to improve the connection scalability (§3.2).

3.1 RDMA Conceptual Model

Figure 3 shows an RDMA conceptual model, based on which, a typical RDMA data flow consists of the following steps:

1. Requester: the user posts a work queue element (WQE) into a send queue (SQ) to issue a SEND request. RNIC fetches the WQE from the SQ to a **WQE Cache**.

2. Requester: RNIC gets the virtual address of the data buffer by parsing the WQE, translates it into the physical address through a **Memory Translation Table** (MTT), and fetches data from the host data buffer using the physical address. RNIC then appends an appropriate RoCEv2 header onto the data and sends out the packet to the responder. The metadata of all outstanding requests is

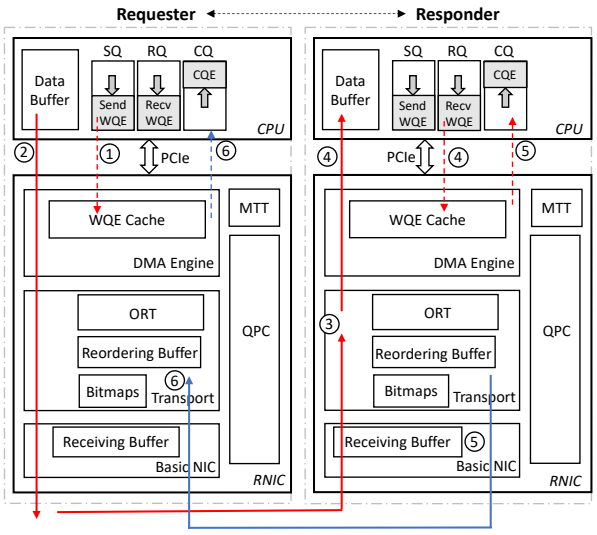


Figure 3: An RDMA conceptual model, and the RDMA data flow using a small SEND message as an example.

stored in an **Outstanding Request Table** (ORT) for fast retransmission in case of packet loss.

3. Responder: the incoming request is first queued in the **Receiving Buffer** and then gets verified. Out-of-order packets will be recorded in **Bitmaps** and reordered using the **Reordering Buffer**.

4. Responder: upon receiving a SEND packet, RNIC fetches a Receive WQE from a receive queue (RQ), queries MTT to get the physical address of the host data buffer, and DMAs the reordered data from the **Reordering Buffer** to the host data buffer.

5. Responder: RNIC replies an acknowledgment (ACK) packet to the requester, and notifies the user with a completion queue element (CQE) to indicate the Receive WQE is consumed.

6. Requester: RNIC receives the ACK, and generates a CQE to indicate the Send WQE is consumed.

Besides, RNIC leverages a **QPC** per QP to track QP/connection related contexts for all modules.

3.2 Data Structures

As concluded in Table 1, we classify the involved data structures into two categories: (1) *common data structures*, required by RDMA in general, and (2) *selective repeat specific data structures*, brought by lossy RDMA.

3.2.1 Common Data Structures

Common data structures are essential to RDMA in general, no matter whether the underlying network is lossy or lossless.

Receiving Buffer. The receiving buffer in the Basic NIC module is used to queue all incoming packets. Its major purpose is to absorb bursts caused by the temporal performance gap between the upstream Ethernet port and the whole downstream RNIC processing logic.

QPC. A QPC maintains for a QP all its contexts, including the DMA states (*e.g.*, the start and end addresses, read and write pointers of SQ & RQ), and connection states (*e.g.*, expected and next packet sequence numbers, window or rate for congestion control). The QPC size we allocate for each QP is 210 B, so the total size for 10K QPs is 2.0 MB.

MTT. RDMA uses virtual addresses in the packet while the PCIe system relies on physical addresses to perform DMA transactions. To perform address translation, RNIC maintains an MTT to map virtual pages of memory regions into physical pages. The size of MTT depends on the total size of memory regions and the page size, irrelevant to the number of connections. For example, considering the total memory region size of 4 GB, the page size of 4 KB, and an MTT entry size of 8 B, the MTT size is equal to $4\,GB/4\,KB * 8\,B = 8\,MB$.

WQE Cache. An SQ WQE cache could be used to cache the Send WQEs fetched from an SQ in host memory. Assuming each QP stores 8 WQEs (64 B*8) in a dedicated cache, 10K QPs consume 4.9 MB on-chip memory. Similarly, RNIC needs to fetch Receive WQEs from the RQ to process incoming SEND requests, and could allocate an RQ WQE cache to store the fetched Receive WQEs. The memory size of the RQ WQE cache is similar to that of the SQ WQE cache.

3.2.2 Selective Repeat Specific Data Structures

These data structures are all introduced by lossy RDMA using selective repeat as the loss recovery mechanism.

Bitmap. Bitmaps are used to track which packets are received or lost [31]. As mentioned in IRN [33], each QP requires five BDP (bandwidth-delay product)-sized bitmaps (500 slots for each bitmap to fit the BDP cap of a network with bandwidth 100 Gbps and RTT 40 µs [5]) and 10K QPs cost 3.0 MB memory in total.

Reordering Buffer. A reordering buffer is used to rearrange the out-of-order packets and ensure in-order delivery to the data buffer in host memory. The reordering buffer is required in a lossy RNIC implementation with the standard RoCEv2 header. As RoCEv2 is designed for the lossless network, its header lacks the necessary information to support out-of-order packet reception without extra reordering buffers.

One option is to allocate a separate reordering buffer for each QP. Each QP requires a BDP-sized (0.5 MB) reordering buffer, so it takes 4.9 GB memory to support 10K QPs. Another option is to maintain a shared reordering buffer for all QPs [31]. However, it does not scale. When multiple QPs experience out-of-order packets, it may soon run out of the shared buffer with limited on-chip SRAM. Hence, we choose the separate reordering buffer option in the analysis.

Category	Data structures	Typical sizes	Optimization ideas	Sizes after optimization
Common	Receiving Buffer	0.6 MB	None	0.6 MB
	QPC	2.0 MB	None	2.0 MB
	MTT	8 MB	Cache (§4.5)	1.2 MB
	WQE Cache	9.8 MB	Cache-free QP scheduler (§4.3)	0
SR Specific	Bitmap	3.0 MB	Bitmap onloading (§4.4.2)	0
	Reordering Buffer	4.9 GB	Header extensions (§4.4.1)	0
	Outstanding Request Table	114.4 MB	Header extensions (§4.4.1)	0

Table 1: Data structures in the RDMA conceptual model. The first three columns show the typical data structures and their memory requirements with 10K QPs. The last two columns summarize our ideas to minimize the on-chip memory requirements of these data structures, and show the memory size after optimization.

Outstanding Request Table. Outstanding request table is used to maintain the mapping between outstanding request packets and their metadata, which are used to quickly locate and retransmit the lost packets. These metadata include (1) packet sequence number (PSN), used to track packet sequences, (2) message sequence number (MSN), used to track message sequences and to locate the WQE associated with that message quickly, and (3) packet offset (PSN_OFFSET), used to locate the data offset inside the corresponding data buffer. With these fields, the outstanding request table size for each QP is 11.7 KB (given the entry size 24 B, entry number 500 sized to BDP), and 10K QPs consume 114.4 MB in total.

In summary, all the data structures derived from the RDMA conceptual model could be classified into two categories: *common data structures* required by RDMA in general, and *selective repeat specific data structures* brought by lossy RDMA. Table 1 summarizes the memory requirements of these data structures in the third column. Both categories require significant memory sizes, and thus need to be optimized to improve connection scalability.

To this end, we make different optimization strategies to minimize these two types of data structures respectively. In particular, all the common data structures required by RDMA should be optimized in a generic way, with architectural innovations that are not specific to lossless or lossy RDMA. The cache-free QP scheduler proposed in §4.3 falls into this strategy. On the other hand, all the selective repeat specific data structures brought by lossy RDMA, could be optimized based on the lossy network assumption. The header extensions and bitmap onloading approaches in the memory-free selective repeat architecture in §4.4 follow this strategy.

4 SRNIC Design

4.1 Design Goal and Guiding Principles

In the design space of RDMA NICs, Mellanox RNICs represent the state-of-the-art in terms of high performance and low CPU overhead, and IRN is the state-of-the-art in network scalability. The design goal of SRNIC is to maximize the connection scalability, while preserving high performance and low CPU overhead as Mellanox RNICs, and maintaining high network scalability as IRN.

To achieve this goal, we follow three design guiding principles:(1) keep as many RDMA functionalities as possible in hardware to achieve high performance and low CPU overhead; (2) handle packet loss as efficient as possible to allow discarding PFC and thus to support large-scale lossy networks; and (3) reduce the on-chip memory requirements as much as possible to support a large number of performant QPs with a limited amount of memory.

4.2 Architecture Overview

Guided by the above principles, we design a scalable RDMA NIC architecture SRNIC, as shown in Figure 4.

The server CPU allocates and manages QPs in the RNIC driver, and runs applications in user space over these QPs. Besides, a software retransmission module resides in user space to maintain the memory-consuming retransmission states collected by hardware and assist packet loss processing (§4.4). A pair of control queues (CtrlQs) is used as the communication channel between the software retransmission module and RNIC hardware.

RNIC hardware consists of three layers: DMA Engine, Transport, and Basic NIC. The *DMA Engine* layer leverages a *QP scheduler* to schedule tens of thousands of QPs from host memory, decides which QP to send data next, and then fetches WQEs and data from that SQ via *data mover*. The *Transport* layer realizes most of RDMA transport functionalities (except for the *software retransmission* in CPU), including a *congestion control* module that implements a hardware-friendly DCTCP [14], and a *hardware retransmission* module that implements the hardware part of selective repeat. The *Basic NIC* layer implements the primary functions of the Ethernet NIC, responsible for sending and receiving RoCEv2 packets via the *100GE MAC*. In addition to these three layers, there are two major data structures: QPC, which maintains all QP-related contexts, and MTT, which stores the mapping between virtual and physical addresses.

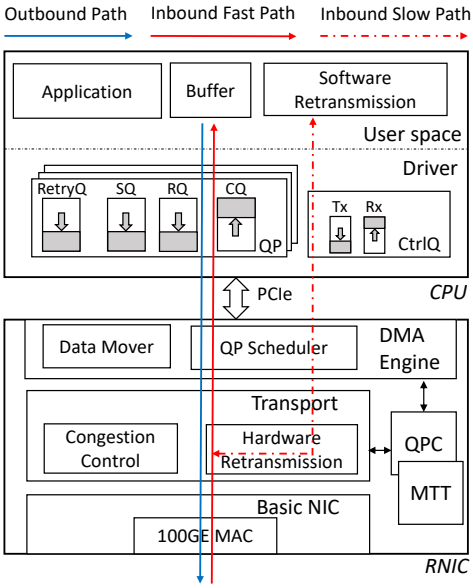


Figure 4: SRNIC architecture.

In order to balance performance and scalability, the data path of SRNIC is divided into a fast path and a slow path (§4.4), which handle sequential and out-of-order (OOO) packets, respectively. The fast path wholly implemented in RNIC processes the majority of traffic consisting of sequential packets, and thus provides hardware-level high performance with low CPU overhead for most packets. The slow path implements software retransmission, processes very little traffic consisting of OOO packets, and onloads bitmaps to host memory for connection scalability.

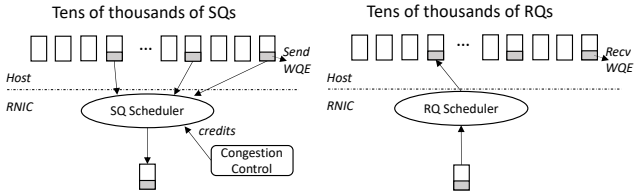
The overhead of the data path separation is very low for two reasons. First, the average packet loss rate in data centers is low (less than 0.01% [20, 41, 43]), and the resulting OOO packets form a very small fraction of traffic. Second, SRNIC only transmits loss events (*i.e.*, metadata of the OOO packets) over PCIe, further reducing the PCIe overhead. For example, the extra PCIe overhead is only 2.46% even with 1% loss rate.

Based on the above architecture, we further make two critical design optimizations: cache-free QP scheduler (§4.3) and memory-free selective repeat (§4.4) to optimize RDMA common data structures and lossy RDMA specific data structures, respectively, in order to address the scalability issues while preserving high performance.

4.3 Cache-free QP Scheduler

4.3.1 SQ Scheduler

An SQ is either *active* when it contains WQEs or *inactive* otherwise. The SQ scheduler (as modeled in Figure 5a) chooses one active SQ each time from tens of thousands of SQs in host memory to send messages next. The design challenges



(a) SQ scheduler model. (b) RQ scheduler model.

Figure 5: The QP scheduler models.

of the SQ scheduler are as follows:

- **Challenge #1**: Active SQs cannot be scheduled blindly, as they are also subject to congestion control, as shown in Figure 5a. Once an SQ is scheduled, if it is not allowed to send messages due to the lack of credits granted by congestion control, the scheduling does not take effect but just wastes time and degrades performance.

- **Challenge #2**: The PCIe round-trip latency between RNIC and host memory is high (around 1 µs in FPGA based RNIC), and it takes at least two PCIe transactions (one WQE fetch and one message fetch), to execute one scheduling decision. Without careful design, the high latency between scheduling iterations will significantly degrade the performance.

- **Challenge #3**: There are tens of thousands of SQs in host memory but very limited on-chip memory within RNIC. It is prohibitive to have separate WQE caches for different SQs in the RNIC.

To address these challenges, SQs should be scheduled when they are both active and have credits (to address Challenge #1), with appropriate batch transactions to hide PCIe latency (to address Challenge #2), and in a WQE-cache-free way (to address Challenge #3).

Guided by these principles, we propose a cache-free SQ scheduler (as shown in Figure 6) that can do fast scheduling among tens of thousands QPs with minimal on-chip memory requirements. It consists of three major components:
Event Mux (EMUX): The EMUX module handles all scheduling related events, including (1) SQ doorbell[4] from the host to indicate which SQ has new WQEs and messages to send; (2) credit update from the *congestion control* module to indicate window or rate adjustment for a connection/SQ; and (3) dequeue event from the *schedule queue* to indicate an SQ is scheduled.

Upon receiving an event, EMUX changes the scheduling states in QPC. There are three scheduling states: an *active* state indicating the SQ has WQEs; a *credit* value indicating

[4]Doorbell is the mechanism for the driver to notify RNIC that a SEND WQE has been posted into an SQ [26]. It is usually implemented by updating the write pointer of the SQ into an RNIC register.

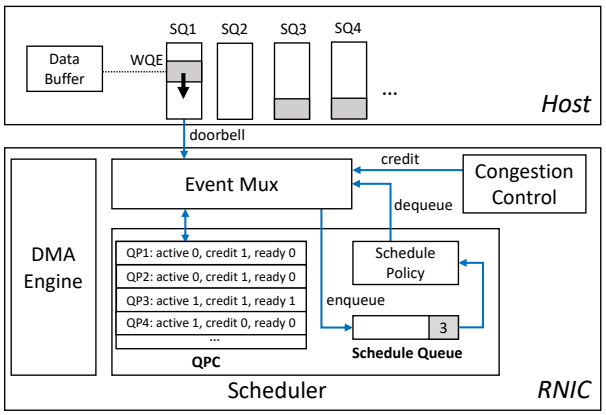


Figure 6: The cache-free SQ scheduler.

the bytes of messages allowed to send, and a *ready* state indicating the SQ is in the *schedule queue* and ready for scheduling. An SQ is ready for scheduling only when it is both active and has available credits, which addresses **Challenge #1**.

Scheduler: The scheduler leverages a *schedule queue* to maintain a list of SQs ready for scheduling. The scheduler implements a round-robin strategy in the *schedule policy* module, by popping a single ready SQ from the head of the *schedule queue* each time, and fetching from that SQ a given amount of WQEs and messages. After this scheduling iteration, if the SQ is still ready for scheduling, it will be pushed back into the *schedule queue* by the EMUX. Other scheduling strategies (*e.g.*, weighted round-robin and strict priority) can be implemented by modifying the *schedule policy* module.

DMA Engine: When an SQ is being scheduled, the DMA engine fetches from that SQ up to n WQEs and $min(burst_size, credit)$ bytes of messages to address **Challenge #2**. After a scheduling iteration, there could be unused WQEs left in RNIC, if the total message size associated with the n WQEs is over $min(burst_size, credit)$ bytes. Unused WQEs are dropped instead of being cached in RNIC, and they will be fetched again next time when its SQ is scheduled. This *fetch-and-drop* strategy enables us to achieve cache-free scheduling to address **Challenge #3**.

There are two critical parameters (n and $burst_size$) to balance tradeoffs. n is the maximum number of WQEs, and $burst_size$ is the maximum bytes of messages allowed to fetch in each scheduling iteration. n reflects the tradeoff between PCIe bandwidth usage and PCIe latency hiding. A smaller n would lead to less PCIe bandwidth waste in the *fetch-and-drop* strategy, but be harder to hide the PCIe latency or saturate the PCIe bandwidth with small messages, while a larger n would perform inversely. In SRNIC, n is set to 8 to balance the PCIe bandwidth utilization and latency hiding. With this setting, the maximum message rate of a single QP is 8 million requests per second (Mrps) (*i.e.*, 8 messages per 1 µs). As for $burst_size$, it reflects the tradeoff between PCIe bandwidth utilization and scheduling granularity. A smaller $burst_size$ would enable finer scheduling granularity and hence less HoL, but be harder to saturate PCIe bandwidth, while a larger $burst_size$ would perform inversely. Based on this analysis, we set $burst_size$ to the PCIe BDP, *i.e.*, 16 KB, to balance performance and scheduling granularity.

In summary, the SQ scheduler adopts a cache-free architecture to do fast scheduling among a large number of SQs with minimal on-chip memory. Specifically, the width of the schedule queue is 2 bytes, *i.e.*, the QPN (QP Number) size, and a schedule queue of 19.5 KB can support 10K SQs.

4.3.2 RQ Scheduler

The RQ scheduler is modeled as shown in Figure 5b. Upon receiving a packet, RNIC gets its QPN by parsing the packet header, fetches a Receive WQE from the RQ indicated by that QPN, and places the packet payload into the data buffer associated with that Receive WQE.

This process seems straightforward, but there is one design decision affecting connection scalability: *do we prefetch and cache Receive WQE in RNIC before the packet arrives?*

If Receive WQEs are prefetched and cached, the incoming packet could hit the WQE cache, reducing the latency by one PCIe round-trip time (*i.e.*, around 1 µs). However, it is hard to predict from which RQ to prefetch Receive WQEs before packets arrive, and thus the cache hit ratio largely depends on the traffic pattern and the cache size. Therefore, we decide to take the cache-free approach without prefetching or caching Receive WQEs, thus improving the connection scalability. Given that the typical RDMA network latency for small messages is tens of microseconds in data centers(*e.g.*, for 1KB messages, RDMA P50 and P99 latency is 24us and 40us, respectively [5]), the increased 1 µs latency is generally negligible. For latency-sensitive scenarios where 1 µs matters, like in rack-scale deployments, a shared Receive WQE cache can be brought back to optimize the latency.

4.4 Memory-free Selective Repeat

The introduction of selective repeat into RNICs increases the challenge to achieve high connection scalability. As analyzed in §3.2.2, the extra data structures brought by selective repeat include outstanding request tables, reordering buffers, and bitmaps, whose memory requirements in total exceed the typical on-chip SRAM sizes of RNICs.

To minimize the memory requirements introduced by selective repeat, SRNIC eliminates the need for outstanding request tables and reordering buffers via RDMA protocol header extensions (§4.4.1), and onloads bitmaps into host memory without sacrificing performance via careful software-hardware co-designs (§4.4.2).

4.4.1 Header Extensions

As described in §3.2, the *outstanding request table* is used to maintain for each QP the mapping between outstanding request packets and their metadata including PSN, MSN, and PSN_OFFSET for fast selective retransmission. We eliminate the need for this data structure, by carrying these per-packet metadata on packet headers, instead of storing them in the on-chip memory. Specifically, we let all outstanding request packets carry these metadata on their headers, and let their response packets echo the same metadata back. In this way, the requester can locate the WQE and its message quickly with metadata in the response packet header.

The *reordering buffer* is used by each QP to rearrange the OOO packets and ensures in-order delivery to the data buffer of user applications. To get rid of the per-QP reordering buffer, our approach is *in-place reordering*, *i.e.*, leveraging the user data buffer pinned in host memory as the reordering buffer. To achieve this, all incoming packets should be placed directly into the user buffer at correct addresses. We make the following header extensions so that RNIC can derive the address for each packet by parsing its header: (1) all SEND packets carry send message sequence number (SSN) and the aforementioned PSN_OFFSET, which can be used by the RNIC responder to locate the corresponding receive WQE and the offset in its associated receive buffer. (2) all WRITE packets carry their target remote addresses [33].

As to RDMA READ, we add acknowledgements to READ requests and responses respectively to add self-clocking for RDMA READ, and schedule RDMA READ at the responder side similar to RDMA WRITE. By doing so, we can apply similar header extensions of SEND and WRITE for READ request and response packets, and more importantly, we can apply window-based congestion control for RDMA.

With these modifications, both sequential and out-of-order packets can be placed directly into the user buffer at the correct address, thus achieving *in-place reordering* and eliminating per-QP reordering buffer in the on-chip memory.

The aforementioned extensions add 8 to 20 bytes of headers to packets. In particular, the header is increased from 58 to 66 bytes for SEND and from 58 to 78 bytes for WRITE, which will decrease the application goodput by 0.7% and 1.8%, respectively, given 1024 byte RoCE MTU.

4.4.2 Bitmap Onloading

As mentioned in §3.2.2, each QP requires five BDP-sized bitmaps, and 10K QPs need 3.0 MB memory to store bitmaps, which alone may exceed the RNIC on-chip memory size (*e.g.*, 2 MB in Mellanox RNIC [24]), thus increasing the challenge to achieve high connection scalability.

We observe that, when there is no packet loss, packets from the same QP are sent and received in order, and an expected PSN (ePSN) in the responder and a last acknowledged PSN

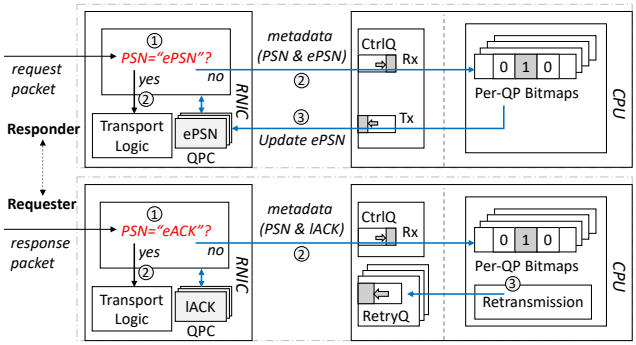


Figure 7: Selective repeat with bitmap onloading.

(lACK) in the requester are enough to track the sequential reception of request and response packets, respectively, without the need of bitmaps; when there is packet loss, OOO packets appear, and bitmaps are only required to track OOO packets.

Based on the above observation, for each QP we maintain an ePSN and a lACK in QPC to process sequential packets in hardware, and onload all bitmaps into host memory to track OOO packets. Assume packet loss rate is low and sequential packets are the majority, most traffic is handled by hardware directly, and little traffic containing the OOO packets is handled by software with the memory-consuming bitmaps in host memory. In this way, we achieve a balance between high performance and high connection scalability.

Figure 7 shows the software-hardware co-designed selective repeat architecture with bitmap onloading. On the responder side, the PSN of an inbound request packet is compared against the ePSN (①). If they match (②), it is a sequential packet and will be handled in the RNIC; otherwise (②), it is an OOO packet and the responder enters into the loss recovery state. In this state, the metadata (PSN and ePSN) of all incoming OOO packets is sent to software, which then fills the bitmaps in host memory to track received packets. After lost packets are received and bitmaps are filled accordingly, a new ePSN is updated (③), and the RNIC exits from the loss recovery state. On the requester side, the PSN of an inbound response packet is compared against an eACK (*i.e.*, a coalesced ACK greater than the lACK) (①). If they match (②), the lACK is updated in hardware; otherwise (*e.g.*, upon receiving NACK or SACK) (②), the requester enters into the loss recovery state. In this state, the metadata of all incoming OOO response packets including PSN and lACK is sent to the software retransmission module, which then manipulates the bitmaps in host memory to track which packets are received by the responder, and makes retransmission decision accordingly. The retransmitted requests are submitted through a Retry Queue (RetryQ) associated with each QP (③). After all retransmitted packets are successfully delivered (indicated by ACKs), the requester exits from the loss recovery state. Another option is to keep bitmaps only in the responder and make the requester stateless. Then, the responder should notify the requester exactly which packets to be retransmitted.

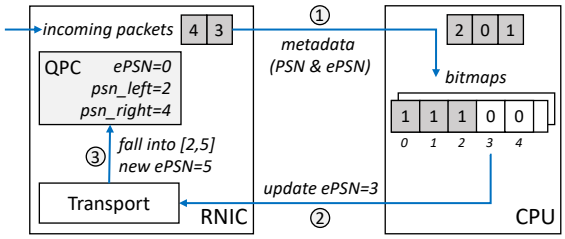


Figure 8: Fast exit from the loss recovery state.

A race condition may arise in the responder when exiting from the loss recovery state. Specifically, when the software updates the new ePSN, there might be inflight metadata of OOO packets with newer PSN between RNIC and CPU. In this case, the updated ePSN is not the latest, and thus the exit fails. To address the race condition problem while preserving high performance, RNIC records the range of the most advanced sequential packets (via $[psn_left, psn_right]$) after it enters the loss recovery state. A QP can exit from the loss recovery state if the updated ePSN falls into $[psn_left, psn_right + 1]$ range, and the ePSN in QPC will be updated to $psn_right + 1$, as illustrated in Figure 8.

4.5 Other Design Considerations

With the cache-free QP scheduler and memory-free selective repeat, all data structures shown in Table 1 are eliminated, except for the receiving buffer, QPC, and MTT.

Receiving Buffer is a shared packet buffer among all QPs and its size is small, so it is not optimized in this paper.

QPC is essential to maintain the per-QP states, and is involved in per-packet processing. To support a large number of performant QPs, we have to store their QPCs entirely in on-chip memory. Therefore, this part is not eliminated, and we preserve as much on-chip memory as possible for QPC to maximize the number of performant QPs.

MTT is memory-consuming as analyzed in §3.2 (*e.g.*, 4 GB memory region requires 8 MB MTT size). Therefore, MTT is maintained in the host memory, and an MTT cache is implemented inside the RNIC by leveraging traffic locality. The cache size does not increase with the number of QPs, and its performance is highly related to traffic patterns. In addition, adopting hugepages (*e.g.*, 2MB/1GB) is a classical optimization to reduce the memory size of address translation tables [24, 40], but requires modification to the applications.

4.6 Design Summary

The last two columns of Table 1 summarize our ideas to minimize the RDMA related data structures, and show the memory requirements after optimizations. Specifically, we eliminate the WQE cache through a cache-free QP scheduler, eliminate all SR-related data structures in on-chip memory through SR-friendly header extensions and bitmap onloading, and minimize the on-chip memory requirements of MTT with a cache, while keeping the large MTT table in host memory.

Resource Usage			
LUT	Register	BRAM	URAM
101102	140816	621	48

Memory Breakdown (MB)				
QPC	MTT	Receiving Buffer	SQ Scheduler	Total
2.3	1.2	0.6	0.3	4.4

Table 2: Resource usage of the SRNIC prototype.

5 Implementation

We build a fully functional prototype of SRNIC using a Xilinx FPGA board with a PCIe Gen3 x16 interface and a 100 Gbps Ethernet port, running at a clock frequency of 300 MHz.

Congestion Control. Since SRNIC introduces ACK based self-clocking for RDMA READ, we therefore can use window-based congestion control for RDMA. Window-based approach in general is more friendly for hardware implementation than rate-based congestion control due to its self-clocking mechanism. More specifically, window-based design is event-driven: congestion window update events are triggered by inbound acknowledgement packets, and window based congestion control for each flow is applied at QP scheduling events. These events are naturally serialized and can be processed one by one. On the other hand, rate-based congestion control is timer-driven. It is challenging to support a large number of timer-based rate limiters in parallel for many concurrent flows. In SRNIC, we use DCTCP.

Memory Consumption. We realize 10K QPs in SRNIC and the resource consumption is broken down in Table 2. SRNIC consumes 4.4 MB on-chip SRAM in total. The QPC table, whose size increases linearly with the QP number, occupies 2.3 MB[5] for 10K QPs. The remaining memories are used by QP-irrelevant data structures, including MTT cache, receiving buffer, and SQ scheduler, which consume constant memories when the QP number increases.

Per Table 2, the precious on-chip SRAM of SRNIC is mainly partitioned between the two most memory-consuming data structures: the QPC table and the MTT cache. A larger QPC table would support more performant QPs, while a larger MTT cache could provide a higher cache hit rate during address translation thus better performance. The best on-chip memory partition strategy between the QPC table and the MTT cache highly depends on scenarios, and it's an interesting problem to explore in the future.

[5] This is slightly larger than 2 MB calculated in Table 1 due to memory alignment overhead, *e.g.*, each memory depth should be a power of 2 in FPGA implementation.

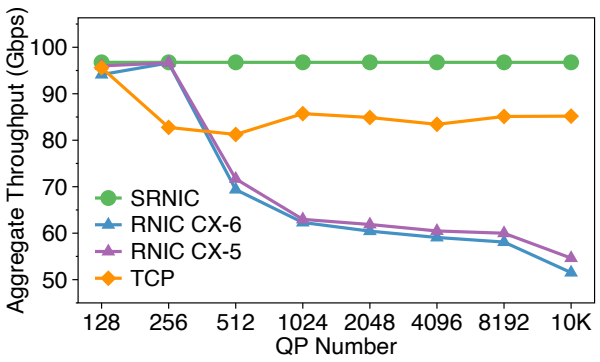


Figure 9: Connection scalability. SRNIC maintains constant high throughput as the number of QPs increases, while the performance of commercial RNICs (Mellanox CX-5 & 6) drops dramatically when the QP number exceeds 256.

6 Evaluation

We evaluate SRNIC using both testbed experiments and large-scale ns-3 simulations [10], and compare it with Mellanox RNICs, IRN, and TCP. Our results reveal that:

- SRNIC achieves high connection scalability: it supports 10K performant QPs, outperforming Mellanox RNIC CX-5 by 18x in terms of normalized connection scalability.
- SRNIC achieves high throughput (97 Gbps), low latency (3.3 µs), and low (5%) CPU overhead.
- SRNIC achieves high network scalability: it is loss-tolerant (up to 75 Gbps goodput under 1% loss rate) and maintains predictable performance over large-scale lossy networks.

6.1 Connection Scalability

We compare SRNIC with Mellanox RNIC CX-5, CX-6, and TCP in terms of connection scalability. The settings of the testbed experiments are as follow. We connect two RNICs directly and launch 16 threads on each side, with each thread executing 512 B send operations. We set the RoCE MTU to 1024 bytes, and use the standard *perftest* benchmarks [11] in all experiments. With the above settings, we measure the aggregate throughput of these solutions while increasing the number of QPs from 128 to 10K, as shown in Figure 9.

SRNIC preserves the highest aggregate throughput almost unchanged at around 97 Gbps when the QP number increases from 128 to 10K. This is expected, as SRNIC keeps the QPC of 10K QPs entirely in the on-chip memory while eliminating or minimizing all other data structures.

TCP also preserves relatively high performance (from 81 to 96 Gbps), as it maintains the contexts of 10K connections in the large host memory, demonstrating high connection scalability but lower and unpredictable performance.

In contrast, the aggregate throughput of Mellanox RNICs CX-5 and CX-6 drops dramatically when the QP number exceeds 256 due to frequent cache misses, as explained in §2.2.

In summary, SRNIC provides much higher connection scalability than commercial RNICs. Specifically, SRNIC realizes 10K QPs with 4.4 MB memory, while Mellanox CX-5 supports 256 QPs with 2 MB memory. To make a fair comparison, we define *normalized connection scalability* as the number of performant connections per 1 MB on-chip memory. SRNIC outperforms Mellanox CX-5[6] by 18x (10 K QPs/4.4 MB vs. 256 QPs/2 MB) in terms of normalized connection scalability.

6.2 Performance and CPU Overhead

We compare SRNIC with CX-6[7] and TCP in terms of throughput, latency, and CPU overhead using a single connection, with the same settings as above (*i.e.*, 1024-byte RoCE MTU, two NICs are connected directly).

Throughput. The throughput comparison is shown in Figure 10a. When the message size exceeds 4 KB, SRNIC and CX-6 both achieve line-rate throughput (97 Gbps), whereas TCP can only achieve up to 37 Gbps since the single CPU core becomes the bottleneck. In our experiments, the maximum message rate that SRNIC can achieve is 6.6 Mrps, comparable to that of the CX-6 (6.3 Mrps). This confirms that RNIC can achieve a high message rate without WQE cache. As mentioned in §4.3.1, the message rate of SRNIC depends on the batch size of the SQ scheduler. In our implementation, the SQ scheduler can request at most 8 WQEs at a time and the average PCIe RTT we measured is 1.1 µs, therefore our result is close to the upper bound of 7.2 Mrps.

Latency. We measure the latency for transmitting 64 B small messages. As Figure 10b shows, the latency of SRNIC is about 3.3 µs, slightly higher than that of CX-6 (1.16 µs). We believe this gap comes from the extra 1 µs added by the cache-free QP scheduler and the clock frequency difference between FPGA (300MHz) and ASIC (GHz) implementations. The latency would be decreased if SRNIC adopts the shared Receive WQE cache or is implemented in ASIC. In contrast, TCP has the highest latency of 24 µs, indicating that bypassing kernel and offloading transport in RDMA is vital for significant latency reduction.

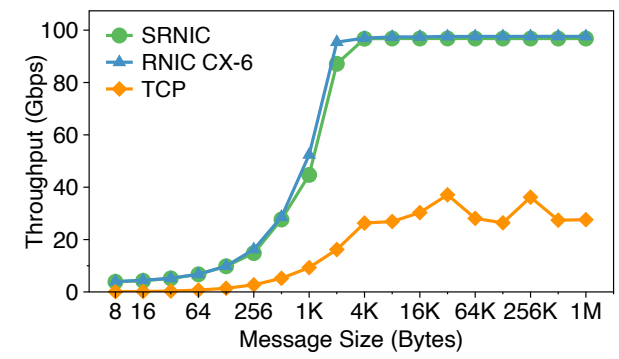
CPU overhead. As shown in Figure 10c, the CPU overhead of SRNIC and CX-6 both maintains at a low level (< 5%) thanks to transport offload and kernel bypass. TCP consumes much more CPU cycles at both the client and server sides (around 100% CPU utilization, not shown in the figure).

6.3 Network Scalability

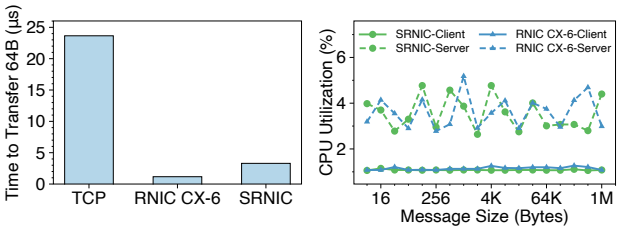
Finally, we evaluate the network scalability of SRNIC. We show the efficiency of loss recovery in SRNIC with testbed

[6] We know the on-chip memory size (*i.e.*, 2 MB) of CX5 [24] but not CX6, so we only compare with CX-5 in terms of normalized connection scalability.

[7] CX-5 and CX-6 behave similarly, so we only show CX-6 thereafter.



(a) Throughput



(b) Latency (c) CPU overhead

Figure 10: Performance and CPU overhead. SRNIC achieves high throughput, low latency, and low CPU overhead, similar to CX-6.

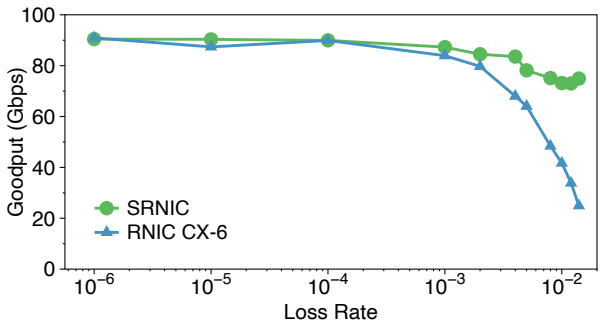
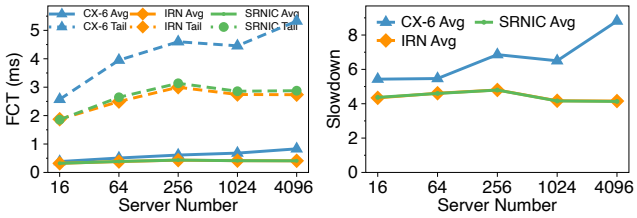


Figure 11: Loss tolerance. SRNIC achieves higher goodput than CX-6 when loss rate increases, as the number of retransmitted packets with selective repeat is much fewer than that with go-back-N.



(a) Average and tail FCT (b) Average slowdown

Figure 12: Performance at different network scales.

experiments, and the performance of SRNIC over large-scale lossy networks via simulations.

Loss tolerance. We compare the goodput of SRNIC with CX-6 at different packet loss rates, which are emulated by placing an FPGA between two RNICs and letting the FPGA randomly drop packets at given rates. We use *perftest* to generate 4 KB messages continuously. We disable congestion control here to exclude the influence of congestion control on loss tolerance, and only compare the loss recovery efficiency between selective repeat in SRNIC and go-back-N in CX-6.

Figure 11 compares SRNIC with CX-6 in terms of goodput under different loss rates. The goodput of CX-6 drops rapidly when the loss rate exceeds 0.1%. In particular, the CX-6 goodput is down to 25 Gbps when the loss rate exceeds 1%. Meanwhile, we monitor the MAC statistics counters in CX-6 and get its raw throughput of ∼97 Gbps, which indicates that most of the RNIC bandwidth is wasted on retransmission caused by go-back-N. The goodput of SRNIC drops much slower than that of CX-6. When the loss rate exceeds 1%, the goodput is still 75 Gbps, 3x higher (75 vs. 25 Gbps) than that of CX-6.

The good loss tolerance of SRNIC comes from both the efficiency of selective repeat and its careful software-hardware co-designs in §4.4.2.

Performance in large-scale lossy networks. We use ns-3 to simulate the transport behavior of SRNIC, and compare it with CX-6 and IRN in large-scale lossy networks. We simulate the fat-tree topologies with the server number ranging from 16 to 4096, with the (ToR, Aggregate, Core) switch number varying among five settings: (1, 0, 0), (4, 4, 0), (8, 8, 0), (64, 64, 16) and (128, 128, 64). The subscription ratio is 1:1 in all topologies. We equip each server with one 100 Gbps NIC connected to one ToR. ToR, Aggregate, and Core switches are connected via 400 Gbps links. The propagation delay of each link is 1 µs.

PFC is enabled for CX-6 but disabled for SRNIC and IRN. We use the traffic trace in Cache_Follower [36], where 53% of the flows are sized between 0 - 100 KB, 18% between 100 KB - 1 MB, and the rest are larger than 1 MB. We set the network load at 0.7 utilization, and configure other algorithm parameters based on their papers.

We primarily focus on three metrics, *i.e.*, average FCT, P99 tail FCT, and average slowdown [33]. The average FCT and tail FCT describe the performance of throughput-intensive flows, while the average slowdown shows the performance of latency-sensitive flows.

As shown in Figure 12, the performance of SRNIC is 1.9 - 2.2x better than CX-6 across all three metrics. As the cluster scale increases, SRNIC maintains stable performance, while the performance gap between SRNIC and CX-6 widens. Meanwhile, SRNIC and IRN perform similarly well as they use the same loss-recovery mechanism (selective repeat) and similar congestion control schemes (DCTCP vs. DCQCN).

7 Discussion

RDMA Protocol for lossy Ethernet. The RDMA protocol was originally designed and simplified for lossless Infiniband, and it *"does not support selective packet retransmission nor the out-of-order reception of packets"*, written in the Infiniband RDMA specification [1]. As a result, the current RDMA, by design, requires a lossless fabric to perform well.

Based on this requirement, when RDMA is introduced into Ethernet-based data centers, Ethernet is turned from lossy to lossless by introducing PFC, rather than re-designing an Ethernet-native or loss-friendly RDMA protocol.

A lossless Ethernet network, however, is inherently difficult to scale and hard to maintain for high availability. It is therefore desirable to look into the other end of the design spectrum: revising the RDMA protocol for a lossy network. This is the path taken by the pioneering work of IRN [33], and SRNIC. We hope these early attempts can inspire the re-design of a new RDMA specification for lossy network, which supports out-of-order packet reception and selective packet retransmission natively and efficiently, and ensures compatibility and interoperability among different protocol versions and RNIC vendors.

SRNIC vs. RoCEv2, iWARP and ToE. There exists a long debate [3, 6] between RoCE and iWARP [35] (ToE [2] is similar to iWARP in the sense of TCP offload). The former takes a bottom-up strategy: start from a minimal, hardware-friendly yet working transport (*e.g.*, go-back-0, no congestion control) and incrementally add more advanced mechanisms (*e.g.*, go-back-N/selective repeat, DCQCN/DCTCP) to make RoCE work better over various networks. The latter takes a top-down strategy: offloading the fully-compatible TCP/IP stack (which is already proven to work well over various networks at scale), and gradually reduce unnecessary complexity to improve hardware friendliness.

SRNIC takes a more balanced approach: it inherits the hardware friendliness (and thus high performance) from RoCE, and introduces only necessary features from TCP such as selective repeat and DCTCP.

SRNIC demonstrates that high network scalability and hardware friendliness can be achieved simultaneously with careful architecture and protocol co-designs. We believe that the best of both RoCE (hardware friendliness) and iWARP/TCP (high network scalability) can coexist as we have shown in SRNIC.

8 Related Work

Several works [30, 32, 42] aim at improving RDMA's network scalability via bringing advanced congestion control algorithms to RNICs.They control the queue length at switches and thus improve RDMA's performance at scale. Note that these works are orthogonal to ours and can be integrated into SRNIC if they are hardware-friendly.

Mellanox tries to improve RNIC's connection scalability via DCT [17] technology, which restricts the number of active connections and avoids QP exhaustion via dynamically creating and destroying QPs. However, such behavior may cause frequent flips of connections, resulting in increased latency and bandwidth waste [27]. StaR [39] improves RNIC's connection scalability at one side by letting the other side save states for it. However, this strategy highly relies on the asymmetric communication pattern, where the client with low concurrency can share its resources with the server with high concurrency, to improve the overall connection scalability.

Other software based transport solutions or DPDK-style NICs, *e.g.*, eRPC [24], FaSST [27], 1RMA [38], and Nitro [37], expect NICs to provide scalable connection-less service including packet transmission and reception, and leverage CPU to implement connection-related semantics. In these solutions, it is the CPU's responsibility to handle most of the transport-related tasks, including packet order maintenance, congestion control, and loss recovery. Though the scalabilities of these approaches are comparable to the software transport TCP, the heavy involvement of CPU results in higher CPU overhead, higher latency, and higher jitter than that of hardware-based transport. In contrast, SRNIC handles almost everything in hardware but leaves only part of retransmission in software, resulting in hardware-level performance in most cases when there is no packet loss, and software-level loss tolerance when packet loss happens.

9 Conclusion

This paper presents the design and implementation of SRNIC, a scalable RDMA NIC architecture, which addresses the connection scalability challenge, while achieving high network scalability, high performance, and low CPU overhead at the same time. Our key insight in SRNIC is to minimize RNIC's memory requirement, by eliminating as many on-chip data structures as possible in a simple yet performant way. Guided by this insight, we make a few RDMA protocol header extensions and architectural innovations to achieve the design goal. Our experiences in SRNIC tell us that existing RDMA header formats originally designed for a lossless environment, are not suitable for much large-scale, lossy data center networks. SRNIC therefore is our first attempt towards more scalable and performant, next-generation RoCE/RDMA designs.

Acknowledgments

We would like to thank our anonymous reviewers and shepherd Yashar Ganjali for their valuable comments. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20-R, GRF-16215119, GRF-16213621, ITF ACCESS, the NSFC Grant 62062005, and a joint HKUST-ByteDance research project.

References

[1] Infiniband architecture volume 1, general specifications, release 1.2.1. www.infinibandta.org/specs, 2008.

[2] Information about the TCP Chimney Offload, Receive Side Scaling, and Network Direct Memory Access features in Windows Server 2008. https://docs.microsoft.com/en-us/troubleshoot/windows-server/networking/information-about-tcp-chimney-offload-rss-netdma-feature, 2008.

[3] The pitfalls in RoCE answered with respect to iWARP. https://www.chelsio.com/wp-content/uploads/2011/05/RoCE-FAQ-1204121.pdf, 2011.

[4] Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE). https://www.infinibandta.org/specs, 2014.

[5] RDMA in Data Centers: Looking Back and Looking Forward. https://conferences.sigcomm.org/events/apnet2017/slides/cx.pdf, 2017.

[6] RoCE vs. iWARP competitive analysis. https://network.nvidia.com/sites/default/files/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf, 2017.

[7] Mellanox ConnectX-5 Product Brief. https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf, 2020.

[8] Mellanox ConnectX-6 Product Brief. https://network.nvidia.com/sites/default/files/doc-2020/pb-connectx-6-en-card.pdf, 2020.

[9] Microsoft Bing. https://www.bing.com/, 2020.

[10] Network Simulator 3. https://www.nsnam.org/, 2021.

[11] OFED Perftest. https://github.com/linux-rdma/perftest/, 2021.

[12] Mellanox NEO-Host. https://support.mellanox.com/s/productdetails/a2v50000000N2OlAAK/mellanox-neohost, 2022.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proc. OSDI*, 2016.

[14] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proc. SIGCOMM*, 2010.

[15] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 2003.

[16] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. POLARDB meets computational storage: Efficiently support analytical workloads in Cloud-Native relational database. In *Proc. FAST*, 2020.

[17] Diego Crupnicoff, Michael Kagan, Ariel Shahar, Noam Bloch, and Hillel Chapman. Dynamically-connected transport service, July 3 2012. US Patent 8,213,315.

[18] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets RDMA. In *Proc. NSDI*, 2021.

[19] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proc. SIGCOMM*, 2016.

[20] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. SIGCOMM*, 2015.

[21] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *Proc. CoNEXT*, 2017.

[22] IEEE. 802.1 qbb—priority-based flow control. 2008.

[23] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proc. OSDI*, 2020.

[24] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *Proc. NSDI*, 2019.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proc. SIGCOMM*, 2014.

[26] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *Proc. ATC*, 2016.

[27] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *Proc. OSDI*, 2016.

[28] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *Proc. NSDI*, 2022.

[29] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *Proc. SIGMOD*, 2016.

[30] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proc. SIGCOMM*. 2019.

[31] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proc. APNet*, 2017.

[32] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proc. SIGCOMM*, 2015.

[33] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proc. SIGCOMM*, 2018.

[34] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. Gentle flow control: avoiding deadlock in lossless networks. In *Proc. SIGCOMM*. 2019.

[35] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. A remote direct memory access protocol specification. Technical report, RFC 5040, October, 2007.

[36] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proc. SIGCOMM*, 2015.

[37] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 2020.

[38] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proc. SIGCOMM*, 2020.

[39] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. Star: Breaking the scalability limit for rdma. In *Proc. ICNP*, 2021.

[40] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA networking for scalable persistent memory. In *Proc. NSDI*, 2020.

[41] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proc. IMC*, 2017.

[42] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proc. SIGCOMM*, 2015.

[43] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proc. SIGCOMM*, 2017.

Hostping: Diagnosing Intra-host Network Bottlenecks in RDMA Servers

Kefei Liu[†], Zhuo Jiang[§], Jiao Zhang[†‡*], Haoran Wei[†§], Xiaolong Zhong[†], Lizhuang Tan[§], Tian Pan[†‡] and Tao Huang[†‡]

[†]BUPT [‡]Purple Mountain Laboratories
[§]ByteDance Inc.

Abstract

Intra-host networking was considered robust in the RDMA (Remote Direct Memory Access) network and received little attention. However, as the RNIC (RDMA NIC) line rate increases rapidly to multi-hundred gigabits, the intra-host network becomes a potential performance bottleneck for network applications. Intra-host network bottlenecks may result in degraded intra-host bandwidth and increased intra-host latency, which can severely impact network performance. However, when intra-host bottlenecks occur, they can hardly be noticed due to the lack of a monitoring system. Furthermore, existing bottleneck diagnosis mechanisms fail to diagnose intra-host bottlenecks efficiently. In this paper, we analyze the symptom of intra-host bottlenecks based on our long-term troubleshooting experience and propose Hostping, *the first bottleneck monitoring and diagnosis system dedicated to intra-host networks*. The core idea of Hostping is conducting loopback tests between RNICs and endpoints within the host to measure intra-host latency and bandwidth. Hostping not only discovers intra-host bottlenecks we already knew but also reveals six bottlenecks we did not notice before.

1 Introduction

RDMA has been applied to many applications [14] [17] [28] [30] [42] [46] in data centers to achieve high throughput and ultra-low latency. As the last hop of network communication, intra-host networking can significantly impact the performance of network applications. However, the intra-host network is far from flawless, and intra-host bandwidth may degrade due to sudden link failures or occupation by other traffic. Previously, the intra-host bandwidth was much greater than the RNIC line rate (e.g., ~63 Gb/s PCIe Gen 3 x8 for 25 Gb/s RNIC), providing sufficient bandwidth redundancy for RNIC traffic. Therefore, the intra-host network rarely became an obstacle to network communication, and bottlenecks in the host network received little attention.

However, bottlenecks in the host network are on the rise. With the increasing demand for high throughput and ultra-low latency, the RNIC line rate increases rapidly (from 25 Gb/s to 200 Gb/s). In contrast, the intra-host bandwidth does not improve equally (e.g., PCIe bandwidth increases from ~63 Gb/s to ~252 Gb/s). As a result, when intra-host bandwidth degrades, traffic on the RNIC is more likely to be throttled. What is worse, both the topology and traffic patterns within the host become much more complicated, making bandwidth degradation caused by sudden link failures or traffic contention happens more frequently. Besides, as intra-host services become more complex, configuration items in the host also increase considerably, leading to a high probability of misconfigurations. Some of them, such as enabling Access Control Service, will redirect GDR (GPU Direct RDMA) traffic to the CPU, leading to a drastic increase in intra-host latency and severe degradation of intra-host bandwidth.

Intra-host bottlenecks [1] may significantly degrade network performance. In our distributed machine learning system, one single intra-host bottleneck can significantly degrade the whole system and may even block the training process. This phenomenon is common in our data center. When it occurs, operators may need hours to days to diagnose the root cause.

Why do intra-host bottlenecks have such a severe impact? If the intra-host bandwidth is lower than the RNIC receiving rate, the RNIC receive buffer may accumulate or even be saturated. When this occurs in a lossy environment (without PFC) [39], RNIC may drop packets. Since RDMA is vulnerable to packet drops, even a low drop rate will result in drastic throughput degradation [24]. While in a lossless environment, RNIC will send PFC pause frames (Tx pause frames) to the upstream switch's egress port to stop its traffic. If the RNIC sends pause frames continually, it may eventually lead to a PFC storm [21] [24] [36], which may bring down the whole network.

The first two authors contributed equally to this paper. This work is done while Kefei Liu, Haoran Wei, and Xiaolong Zhong are doing a joint research project at ByteDance. (*Jiao Zhang is the corresponding author.)

[1] In the following, we use "intra-host bottleneck" as the bottleneck in the host network and "network bottleneck" as the bottleneck in the inter-host network, i.e., switches and cables.

Therefore, when an intra-host bottleneck occurs, it should be discovered, diagnosed, and resolved as soon as possible.

However, due to the lack of an efficient intra-host bottleneck monitoring system, bottlenecks can hardly be noticed when they occur. When customers complain to the network team about performance degradation, the upper layer service usually has been severely influenced by the bottleneck. In addition, the phenomena caused by intra-host and network bottlenecks may be similar. Thus, when network performance degrades, operators need first to judge whether the host or the network should be blamed. Furthermore, when finding the bottleneck lies in the host, operators need to log in to the host, execute a series of test cases and conduct some profiling tools to infer the bottleneck. The whole process is time-consuming. What is worse, existing profiling tools could only be used for specific devices, such as Intel PCM [2] for Intel CPUs, AMD uProf [1] for AMD CPUs, and Nvidia SMI [9] for Nvidia GPUs. As each host may have devices from different vendors, operators may need different toolsets for each diagnosis, which brings additional learning and execution overhead.

To solve the limitations above, we propose Hostping, *the first bottleneck monitoring and diagnosing system dedicated to intra-host networks*. It could be deployed on all RDMA servers with low overhead and adapt to devices from different vendors. When intra-host bottlenecks occur, Hostping could quickly discover them and automatically diagnose their root causes. Thus, when network performance degrades, we can rapidly judge whether the host or the network should be blamed.

We need to address three challenges to achieve these design targets. Firstly, we need to find and measure metrics that could effectively discover and diagnose intra-host bottlenecks. Secondly, we need to keep responsive to intra-host bottlenecks with low overhead. Finally, we need to efficiently diagnose intra-host bottlenecks based on measured data.

Based on our long-term troubleshooting experience, we realized that leveraging intra-host bandwidth and latency as metrics could effectively discover and diagnose most intra-host bottlenecks. This guides the core idea of Hostping: conduct loopback tests between RNICs and endpoints (GPUs and memory nodes [33]) within the host to measure intra-host latency and bandwidth. By registering memory regions in different endpoints, Hostping could evaluate the latency and bandwidth of any intra-host path that a message received by an RNIC can take. To keep Hostping responsive to intra-host bottlenecks without degrading application performance, we design a hardware monitor to determine when to launch it. Finally, we propose an efficient diagnosing mechanism that could effectively identify the root cause of intra-host bottlenecks even under the interference of service traffic on RNICs.

We evaluate Hostping on over 300 servers in our distributed machine learning system. During the deployment, Hostping not only discovers intra-host bottlenecks we already knew but also reveals six bottlenecks we did not notice before, such as CPU root port failures and memory channel flapping. To summarize, this paper makes the following contributions:

- We analyze the symptom of intra-host network bottlenecks based on our long-term troubleshooting experience and realize that most intra-host bottlenecks have one or both of the following symptoms: intra-host bandwidth degradation and intra-host latency increase.
- We design Hostping, the first bottleneck monitoring and diagnosing system dedicated to intra-host networks.
- We propose an efficient diagnosing mechanism that could effectively identify the root cause of intra-host bottlenecks even under the interference of service traffic on RNICs.

2 Background & Motivation

2.1 Intra-host Bottlenecks

When sending/receiving a message, the RNIC will read/write it from/to an intra-host endpoint (e.g., memory node, GPU) through multi-hops in the host network, such as PCIe links, memory channels, and inter-socket buses (e.g., Intel QPI [51]/UPI [11] and AMD xGMI [12]). We refer to the round-trip latency and the maximum available bandwidth between the RNIC and the endpoint as ***intra-host latency*** and ***intra-host bandwidth***[2], respectively.

Previously, intra-host bandwidth was much greater than the RNIC line rate, providing sufficient bandwidth redundancy. Therefore, the host rarely became an obstacle to network communication, and intra-host bottlenecks received little attention. In recent years, with the increasing demand for high throughput and ultra-low latency from applications, the RNIC line rate has increased rapidly. In contrast, the intra-host bandwidth does not improve equally. As a result, when intra-host bandwidth degrades due to link failures or contention from other intra-host traffic, it is more likely to trigger network performance degradation.

What is worse, both the topology and traffic patterns within the host become much more complex, making the intra-host bandwidth degradation commonplace [13] [16] [19] [35] [37]. To satisfy the ever-increasing demand for computation capability, more GPUs and RNICs are integrated into one single host. For example, the latest Nvidia DGX-A100 [5] server incorporates 8 Nvidia A100 GPUs and 4 Mellanox 200 Gb/s RNICs. This leads to much more complicated intra-host traffic patterns and more bandwidth contention. In addition, as the number of root ports [43] on the CPU socket is limited, more PCIe switches are required to interconnect these devices. As a result, the intra-host topology becomes more complex, leading to more frequent intra-host link failures.

[2]It could be further divided into sending bandwidth from the endpoint to the RNIC and receiving bandwidth from the RNIC to the endpoint. If not explicitly mentioned, it indicates the minimum value of the sending and receiving bandwidth.

Furthermore, as intra-host services become more complicated, configuration items in the host also increase considerably, leading to a high probability of misconfigurations. Among them, some misconfigurations may lead to severe intra-host bottlenecks. For example, ACS (Access Control Service) is a PCIe configuration used in IO virtualization. GDR is a widely used communication method in machine learning, which uses the GPU to communicate directly with the RNIC without any involvement of the CPU and host memory. However, all GDR traffic will be redirected to the CPU with ACS enabled, leading to a drastic increase in intra-host latency and severe degradation of intra-host bandwidth.

2.2 The Impact of Intra-host Bottlenecks

When bottlenecks appear in the host, the intra-host bandwidth may be lower than the RNIC receiving rate, and the RNIC receive buffer may accumulate. If the receive buffer is saturated in a lossy environment (without PFC) [39], the RNIC will drop packets. Since RDMA is vulnerable to packet drops, even a low drop rate will result in drastic throughput degradation [24]. While in a lossless environment, when the RNIC receive buffer exceeds a threshold, it will send pause frames to the upstream switch's egress port to stop its traffic. If the RNIC sends pause frames continually, it may finally lead to a PFC storm, which may bring down the whole network.

One single intra-host bottleneck may significantly degrade the distributed machine learning system. To achieve better training performance, developers aggregate more and more servers in a distributed system. However, this leads to more frequent performance bottlenecks. In data-parallel training, before updating the neural network parameters, all involved GPUs need to aggregate their local gradients [16] [28] [45]. In this process, GPUs may communicate in one or several rings [22] [38] [41] consisting of intra-host links (e.g., NVLinks [8], PCIe links) and network links to achieve optimal bandwidth utilization. This ring-based communication is extremely sensitive to network and intra-host bottlenecks. A single RNIC suffering from degraded intra-host bandwidth may significantly slow down the aggregation process of the whole system. We conducted a ring-based nccl all-reduce test [7] with eight hosts, and each host has a 200 Gb/s RNIC for network communication. Fig.1 shows the throughput of each host during the test. In this scenario, an RNIC's PCIe link has degraded bandwidth due to a link failure, leading to a slow sending/receiving rate. As a result, the throughput for all the hosts drops drastically to 50 Gbps (~70% lower than the ideal).

Frequent intra-host bottlenecks bring more challenges for performance bottleneck diagnosis. When packet drops or bandwidth degradation occur on a path, how to diagnose the root cause? This problem generally lies in the network when few intra-host bottlenecks appear, and operators only need to check each link and switch on the path in sequence. However, as intra-host bottlenecks occur much more frequently,

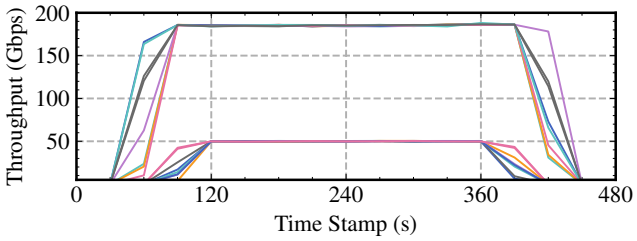


Figure 1: One single bottleneck degrades the throughput of the entire machine learning system by 70%. The upper lines are ideal, and the lower lines are abnormal. The throughput of each host is calculated every 30 seconds.

the same phenomenon may also be caused by the degraded intra-host bandwidth. As a result, operators must first distinguish whether the host or the network should be blamed, which brings more challenges for bottleneck diagnosis.

2.3 Limitations of Existing Intra-host Bottleneck Diagnosis Mechanisms

When an intra-host bottleneck occurs, it must be discovered, diagnosed, and resolved as soon as possible. Unfortunately, as far as we know, there are currently no monitoring and diagnosing systems dedicated to the host network in data centers, and intra-host bottleneck diagnosis is inefficient.

Unresponsive. When bottlenecks occur in a host, they can hardly be noticed in time due to the lack of an efficient intra-host bottleneck monitoring system. However, when customers (e.g., the machine learning team) complain to the network team about performance degradation, the upper layer service has usually been severely influenced. Thus, operators require a responsive monitoring system to quickly discover intra-host bottlenecks, avoiding application performance degradation.

Time-consuming. When a system suffers from degraded performance, operators usually need to run benchmark tests, such as perftest [10] and nccl-test [7], to narrow down the problem. However, these tests reflect "end-to-end" performance, including senders, networks, and receivers. Thus, they cannot quickly determine whether the bottleneck occurs in the network or the host. When finding the bottleneck lies in the host, root cause diagnosis is still challenging due to the complex intra-host topology. Operators need to log in to the host, execute a series of test cases, and conduct some profiling tools to evaluate all intra-host links. The entire process above needs to be conducted manually, which is time-consuming.

Fragmented. When an intra-host link has anomalous performance, operators may need to run some profiling tools to determine whether the link is occupied by other traffic. However, these tools are usually vendor-specific, such as Intel PCM for Intel CPUs, AMD uProf for AMD CPUs, Nvidia SMI for Nvidia GPUs, and Mellanox Neohost [4] for Mellanox RNICs. Unfortunately, each host in data centers may have a different combination of equipment, such as different

network adapters (Mellanox, Broadcom, or Intel), different CPUs (Intel or AMD), and different GPUs (Nvidia or AMD). As a result, when diagnosing bottlenecks in the host, operators need to utilize different combinations of tools, which brings additional learning and execution overhead.

2.4 Targets of Hostping

Considering the limitations above, we desire to develop a dedicated intra-host bottleneck monitoring and diagnosing system, which could be deployed on all RDMA servers with little overhead and adapt to devices from different vendors. When intra-host bottlenecks appear, the system can quickly discover them and automatically diagnose their root causes. Thus, when network performance degrades, we can rapidly judge whether the bottleneck lies in the host or the network. In conclusion, this system should have the following characteristics:

- **Responsiveness:** It should quickly discover intra-host bottlenecks and diagnose their root causes.
- **Deployability:** It should be implementable with commodity hardware.
- **Scalability:** It should be compatible with equipment from different vendors.
- **Lightweight:** It should have negligible interference with services in the host.

3 Hostping Overview

In this section, we will first introduce the challenges we should address to achieve the targets of Hostping (3.1). Then we will analyze the symptoms of intra-host network bottlenecks based on our long-term troubleshooting experience, which guides the core idea of Hostping (3.2). Finally, we will briefly illustrate the framework of Hostping (3.3).

3.1 Challenges

To realize the targets of Hostping, there are three main challenges to be solved:

Find and measure metrics that could effectively discover and diagnose intra-host bottlenecks. As the topology and traffic patterns within the host become much more complex, the root causes of intra-host performance bottlenecks are heterogeneous. We need to find some unified metrics that could effectively uncover intra-host bottlenecks and precisely infer their root causes. Besides, since the intra-host network is like a black box, measuring these metrics with high accuracy is also challenging.

Be responsive to intra-host bottlenecks with low overhead. Diagnosing intra-host performance bottlenecks requires evaluating all the links in the host. Due to the complexity of the host topology, this is not an easy task and will have a non-negligible impact on the applications within the host. For example, active probing consumes CPU memory, GPU video memory, and bus bandwidth. How can we quickly perceive intra-host bottlenecks with low overhead to the performance of applications running in the host?

Effectively diagnose intra-host performance bottlenecks based on measured data. During the operation of Hostping, we will collect many performance data through active probing and monitoring. However, the complex intra-host topology makes it challenging to infer intra-host bottlenecks from scattered data. Besides, the data measured by active probing may be influenced by the service traffic on the RNIC. In this scenario, the degraded performance data does not necessarily mean the emergence of an intra-host bottleneck. We need to find an efficient bottleneck diagnosis mechanism to determine whether there is an intra-host bottleneck and find its root cause effectively based on scattered performance data.

3.2 Symptoms of Intra-host Bottlenecks

As mentioned above, intra-host bottlenecks are varied. How to use the least number of metrics to uncover most intra-host bottlenecks? Based on our long-term troubleshooting experience, we realize that although different root causes may be blamed, most intra-host bottlenecks have one or both of the following symptoms: **intra-host bandwidth degradation** and **intra-host latency increase**. Furthermore, leveraging intra-host bandwidth and latency as metrics could effectively discover and diagnose most intra-host bottlenecks. This guides the core idea of Hostping: conduct loopback tests between RNICs and endpoints within the host to measure intra-host latency and bandwidth. Next, we will introduce these two symptoms and their possible causes.

3.2.1 Bandwidth Degradation

Intra-host bandwidth degrades when an intra-host link is failed or is occupied by other traffic in the host. The RNIC receive buffer will accumulate when the intra-host bandwidth is lower than the RNIC receiving rate. If this situation continues, it will finally trigger packet drops (in lossy environments) or PFC pause frames (in lossless environments), leading to severe network performance degradation.

As the host topology becomes more complicated, the possibility of link failures in the host boosts. In addition, due to the large number of data center hosts, even if link failures are unusual on a particular host, they frequently occur throughout the data center. We encounter abnormal servers even daily in severe cases. What is worse, the locations of failures are varied, requiring a great deal of time for debugging. The host topology inside one of our most used training machines is shown in Fig.2, which has two Intel Xeon CPUs connected through Intel UPI (Intel UltraPath Interconnect). Each CPU

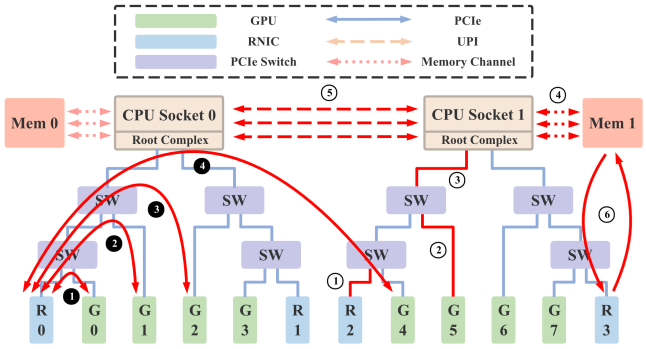


Figure 2: The host topology in one of our most used training machines. ❶-❹ show the GDR distance between an RNIC and a GPU. ①-⑤ show the link failures we encountered in practice, and ⑥ shows intra-host bandwidth degradation due to bandwidth contention.

root complex [43] is attached with four Nvidia A100 GPUs[3] and two Mellanox CX6-DX 200 Gb/s RNICs through multiple PCIe switches. As shown in Fig.2, we have encountered failures of ① RNIC PCIe links, ② GPU PCIe links, ③ CPU root ports, ④ memory channels, and ⑤ UPI in practice. Some issues cannot be detected via static commands such as *lspci* and can only be discovered through benchmark tests.

Furthermore, services in the host are becoming more complicated, leading to more bandwidth contention. When the host bandwidth is occupied by other traffic, traffic on the RNIC may be congested (Fig.2 ⑥). Here, we give two practical examples. First, as RDMA devices are far from flawless, TCP and RDMA traffic may co-exist in the same host to meet high availability and a Service-Level Agreement [21]. However, the processing of TCP in the Linux kernel may consume a lot of memory bandwidth, leading to a slow receiving rate for RDMA traffic. Besides, in the training scenario, a physical machine is usually split into multiple Virtual Machines (VMs) to fully utilize host resources. In this case, communication between two VMs in the same host may trigger loopback traffic, which consumes the RNIC PCIe bandwidth and slows down the receiving rate from other hosts [32]. As shown in Fig.3, both link failures and bandwidth contention may throttle RNIC throughput and trigger a large number of PFC pause frames.

3.2.2 Latency Increase

When sending/receiving a message, the RNIC will read/write it from/to an endpoint (e.g., memory node, GPU) through multi-hops in the host network, such as PCIe links, memory channels, and inter-socket buses. We refer to the round-trip latency from the RNIC receive buffer to the endpoint as *intra-host latency*. Intra-host latency increases when there are too

[3]GPUs are connected via NVLinks and NVSwitches [8] for intra-host GPU-to-GPU communication (not shown in Fig.2).

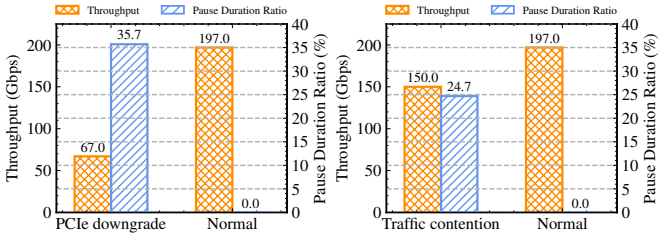


Figure 3: Both link failures (①-⑤) and bandwidth contention (⑥) will lead to intra-host bandwidth degradation, which may throttle RNIC throughput and trigger a large number of PFC pause frames.

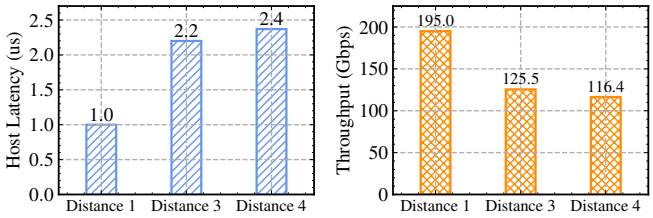


Figure 4: As the GDR read distance increases (from ❶ to ❹), the intra-host latency rises, especially when going through the CPU root complex. Accordingly, the throughput degrades due to limited outstanding read request TLPs.

many hops between the RNIC and the endpoint. High intra-host latency hurts application latency and may significantly degrade intra-host bandwidth. When an RNIC needs to read from an endpoint, it sends PCIe read request TLPs (Transaction Layer Packets) [34] to the endpoint, and the endpoint will respond data to the RNIC after receiving the request. Therefore, when intra-host latency increases, the RNIC needs to send more read requests to sustain the line rate. However, RNICs limit the maximum outstanding read requests. As a result, intra-host bandwidth degrades when intra-host latency increases significantly.

Next, we leverage GDR traffic to illustrate the impact of high intra-host latency on intra-host bandwidth. GDR has been widely used in data centers to improve training performance in distributed machine learning systems. With GDR, the RNIC can write and read GPU video memory directly without using host memory, effectively improving the intra-host latency and intra-host bandwidth. However, GDR suffers from high latency when traffic traverses the CPU root complex. As shown in Fig.2, there are four types of communication distances between an RNIC and a GPU: ❶ traversing a single PCIe switch, ❷ traversing multiple PCIe switches without traversing the CPU root complex, ❸ traversing the CPU root complex without traversing the UPI, and ❹ traversing the UPI.

In the experiment, we use GDR read to test the impact of different communication distances on intra-host latency and bandwidth. We leverage Mellanox Neohost to measure

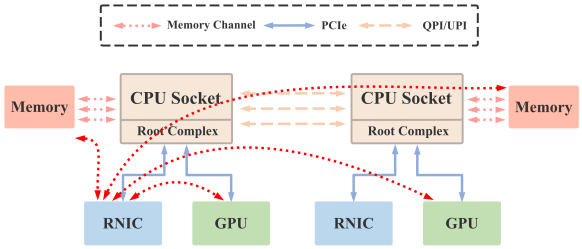


Figure 5: The core idea of Hostping: conduct loopback tests between RNICs and endpoints (GPUs and memory nodes) within the host to measure intra-host latency and bandwidth.

the intra-host latency. Since the latency of ❶ and ❷ is almost the same, we only compare the intra-host latency and the GDR bandwidth of ❶, ❸, and ❹ in the experiment. As shown in Fig.4, when the RNIC communicates with the closest GPU (distance ❶), the host latency is 1 µs, and the RNIC can achieve almost the line rate. For distance ❸, when GDR packets need to pass through the CPU root complex, the host latency rises dramatically to 2.2 µs, and the throughput drops sharply to 125.5 Gbps. As for distance ❹, traversing the UPI bus brings an additional 200 ns delay, and the throughput degrades to 116.4 Gbps. Just 1.4 µs of additional intra-host latency results in a 40% drop in intra-host bandwidth.

3.3 Framework of Hostping

As shown in Fig.5, the core idea of Hostping is conducting loopback tests between RNICs and endpoints within the host to measure intra-host latency and bandwidth and leveraging the measured data to infer intra-host bottlenecks. Hostping is implemented based on commodity RNICs. Thus, it could run on all RDMA servers in data centers.

In the loopback test, the RNIC will read messages from one endpoint to its buffer and then write them back directly. In this process, all communication occurs inside the host without any network participation. Therefore, we could leverage the loopback latency and bandwidth to reflect intra-host latency and bandwidth. Furthermore, by conducting loopback tests between an RNIC and all endpoints in the host, we could evaluate the latency and bandwidth of all intra-host paths that a message received by the RNIC can take. When network performance degrades, if RNICs find no anomalies in loopback tests, we infer that the bottleneck occurs in the network. On the contrary, when the loopback test to an endpoint shows anomalous results, we confirm that a bottleneck exists on the path between the RNIC and the endpoint.

Fig.6 shows the framework of Hostping. The Hostping agent is deployed on RDMA servers and consists of three components: hardware monitor, Hostping engine, and data analyzer. The Hostping engine implements the core logic of Hostping and consists of two functions: (1) leverage RNICs to measure intra-host latency and bandwidth; (2) monitor bus utilization (PCIe links, inter-socket buses, and memory

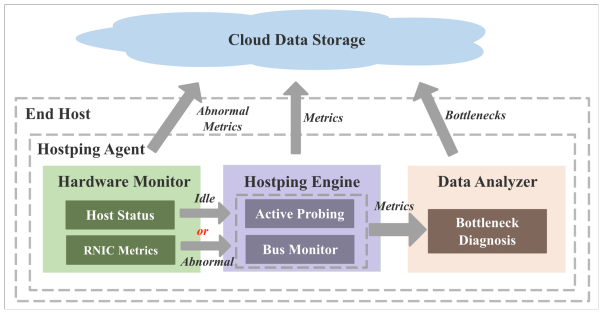


Figure 6: The framework of Hostping.

channels). The hardware monitor judges when to run the Hostping engine based on host status and abnormal metrics on RNICs. The data analyzer is responsible for diagnosing intra-host bottlenecks based on the data collected by the Hostping engine. All these modules will upload the information they collect to the cloud, which will be the basis for subsequent bottleneck diagnosis.

4 Hostping Design

In this section, we will first illustrate the functions of the Hostping engine and how to measure intra-host latency & bandwidth with the loopback test (4.1). Then, we will introduce how to utilize the hardware monitor to keep Hostping highly responsive to intra-host bottlenecks with low overhead (4.2). Finally, we will present how to diagnose intra-host bottlenecks with the data analyzer (4.3).

4.1 Hostping Engine

4.1.1 Measure Intra-host Latency & Bandwidth

Next, we will illustrate how to measure intra-host latency and bandwidth in the Hostping engine. Fig.7 demonstrates the process of the loopback test. First, the Hostping engine leverages `ibv_reg_mr` [3] to register two memory regions (read and write) in an endpoint for sending and receiving, respectively. Next, the Hostping engine uses `ibv_post_send` [3] to post a write WQE (Work Queue Element. Tell the RNIC to read the message of a specified *size* from the read region and write it to the write region) and doorbell the RNIC to fetch the WQE. Then the RNIC will send a request to read the message from the read region. Since the receiver is the same RNIC as the sender, the RNIC will directly write the message back to the write region instead of sending it to the network. Finally, after all PCIe write packets are sent out, the RNIC will generate a completion notification and inform the Hostping engine that the transmission is finished. By measuring the span between the call of `ibv_post_send` and the polling of completion, the Hostping engine could figure out the loopback latency. Moreover, by registering memory regions in different endpoints, we could get the loopback latency between the

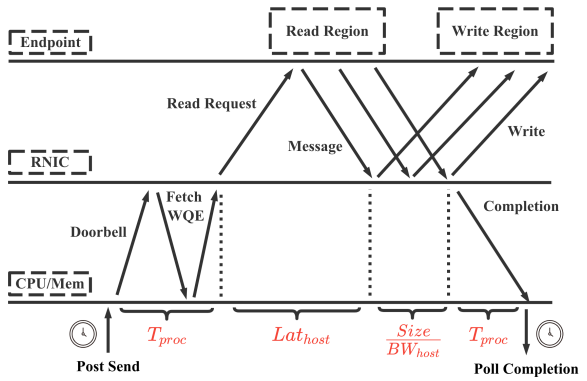


Figure 7: The process of the loopback test. Hostping engine figures out loopback latency by measuring the span between the call of `ibv_post_send` and the polling of completion.

RNIC and any intra-host endpoint. The measured loopback latency could be approximated[4] as follow:

$$Lat = T_{proc} + Lat_{host} + \frac{Size}{BW_{host}} \tag{1}$$

T_{proc} contains two periods: (1) the duration between the call of `ibv_post_send` in the Hostping engine and the RNIC sending the first read request, and (2) the duration between the RNIC sending out all the PCIe write packets and the CPU polling the completion. The intra-host bandwidth (BW_{host}) is determined by the minimum bandwidth in PCIe read and write. Besides, the measured latency also includes intra-host latency (Lat_{host}). When we leverage a small message, the measured latency is close to:

$$\lim_{size \to 0} Lat = T_{proc} + Lat_{host} \tag{2}$$

It is hard to measure T_{proc} on commodity RNICs. Nevertheless, T_{proc} generally remains the same when the RNIC is underutilized and does not suffer from intra-host bottlenecks. In this case, we could leverage the change of small message latency to reflect the variation of intra-host latency. Thus, when the measured latency of a small message increases drastically, we could infer that there is an intra-host bottleneck leading to abnormal intra-host latency. On the contrary, when we use a very large message, the measured latency is close to:

$$\lim_{size \to \infty} Lat = \frac{Size}{BW_{host}} \tag{3}$$

Then the latency reflects intra-host bandwidth. Actually, we do not need to use a very large message in practice. We could use the difference between the latency of large and small messages (Equation 1 - Equation 2) to obtain Equation 3. In practice, our large message size is 128K bytes for 200 Gb/s RNICs, and our small message size is 1 byte.

[4]Here *size* refers to the message size. For simplicity, we do not consider PCIe encapsulation overhead (e.g., TLP header).

4.1.2 Monitor Bus Utilization

While the loopback test could reveal anomalous intra-host paths and links, it fails to diagnose the root cause of anomalies in some scenarios. For example, when the loopback test shows a memory channel has degraded bandwidth, how to further determine whether the root cause lies in traffic contention or a link failure?

To solve this problem, we implement a monitoring module in the Hostping engine to monitor bus utilization (PCIe links, inter-socket buses, and memory channels). Therefore, when the loopback test shows an intra-host link has degraded bandwidth, we could further check its utilization. If the link is overloaded, we infer that the root cause lies in traffic contention. Otherwise, the link is possibly failed. Unlike previous vendor-specific tools, our monitor could automatically adapt to devices from different vendors, and operators no longer need to learn and use various tools for different devices.

4.2 Responsiveness with Low Overhead

When performance bottlenecks occur in the host network, we hope Hostping can automatically, quickly, and accurately locate their root causes. However, high responsiveness and low overhead are usually a trade-off. We can frequently run loopback tests to judge whether there are performance bottlenecks in the host. However, loopback tests consume CPU/GPU memory and intra-host bandwidth, leading to contention with service traffic. Thus, frequent loopback tests will have a non-negligible impact on applications in the host. How could we ensure responsiveness to bottlenecks with low overhead to application performance?

Generally, data center hosts keep switching between busy and idle status. When the host is idle (little traffic on RNICs and all GPUs are inactive), we could frequently run loopback tests to keep responsive to intra-host bottlenecks, regardless of the overhead. When the host is busy with services and the network performance is degraded due to intra-host bottlenecks, abnormal metrics on the RNIC, such as packet drops and Tx pause frames, will usually appear. These metrics are indicators of intra-host bottlenecks. Therefore, we could execute loopback tests when these abnormal metrics appear. This way, Hostping keeps responsive to intra-host bottlenecks with low overhead to application performance.

We implement a hardware monitor in the Hostping agent to achieve the targets above. It (1) monitors host status and abnormal metrics on RNICs and (2) determines when to run the Hostping engine. In general, it has two functions:

- Monitor RNIC throughput and GPU status periodically. If the throughput of all RNICs is less than the threshold Thp_{low}, and all GPUs are idle, execute the Hostping engine to detect if there are intra-host bottlenecks. Otherwise, skip this execution.

- Monitor abnormal metrics on all RNICs. If the Tx pause duration ratio is larger than PFC_{high} or packet drops appear, execute the Hostping engine immediately to diagnose intra-host bottlenecks.

4.3 Bottleneck Analysis

In this section, we will demonstrate how the data analyzer leverages the data collected by the Hostping engine to determine whether there is a bottleneck within the host and diagnose its root cause. We first discuss how to diagnose intra-host bottlenecks when the host is idle. In general, the analyzer determines intra-host path status (normal or abnormal) by comparing the measured intra-host path bandwidth with the baseline and leverages path status to infer anomalous links. This idea is inspired by *binary network tomography* [15] [18] [27]. Besides, the analyzer compares measured intra-host path latency with the baseline to assist in root cause diagnosis. The baseline path bandwidth and path latency are obtained via loopback tests on a batch of idle hosts with the same devices and configurations.

$$y_j = \prod_{i:\ \text{link}_i \in \text{path}_j} x_i, \forall j, \tag{4}$$

The measured path bandwidth reflects the minimum link bandwidth on it. As shown in Equation 4, $y_j, x_i \in \{0,1\}$, represents the status of path j and link i, respectively (1 for normal and 0 for abnormal). If the measured path bandwidth is lower than the baseline by $Abnormal_{th}$, we infer that one or more links on this path suffer from degraded bandwidth and mark this path as abnormal. However, a path with the expected bandwidth is not necessarily bottleneck-free. It depends on whether the RNIC can reach the line rate on this path. For affinitive endpoints of the RNIC (memory nodes[5], GPUs under the same root port as the RNIC), the path bandwidth could reach the RNIC line rate. If the bandwidth of these paths is as expected, we consider them normal. However, as demonstrated in Section 3.2.2, the RNIC could not reach the line rate for GPUs under different CPU root ports due to high intra-host latency. In this case, if the bandwidth of a link degrades but is still higher than the RNIC rate, the measured bandwidth is still close to the baseline. For these paths, we only judge whether they are abnormal based on the baseline.

With adequate path status, we can judge the status of each link within the host. Our algorithm is shown in Algorithm 1. In a symmetric topology like Fig.2, conducting loopback tests between RNICs and their affinitive endpoints could evaluate all intra-host links. Nevertheless, we do full-mesh tests when the host is idle to improve the accuracy of bottleneck inference. If no abnormal paths could be found, we conclude that there is no bandwidth bottleneck. Otherwise, we will diagnose anomalous links based on Algorithm 1. First, we mark all intra-host links as uncertain. Next, we traverse all normal paths and mark all links on them as normal. Then, we traverse all abnormal paths. If an abnormal path has uncertain links, we mark all these links as abnormal, and *abnormal_cnt* records how many RNICs mark a link as abnormal. If all the links on an abnormal path are normal, some links may be flapping. Then we set all the links on this path to gray.

When the host is idle, most bottlenecks could be attributed to *link failures* or *misconfigurations*. The analyzer first judges whether the RNIC is a bottleneck. If the path status between an RNIC and all its affinitive endpoints is abnormal, then the RNIC PCIe link may be failed. If the PCIe link connected to a GPU is marked as abnormal, the analyzer will further check the path latency between the GPU and its affinitive RNIC. If the latency is also abnormal, a misconfiguration (e.g., enabling ACS) may be the root cause. Otherwise, a link failure should be blamed. For other abnormal links, the analyzer diagnoses them as failed links. In addition, links marked as gray in three consecutive loopback tests will be identified as flapping links. All abnormal links and their possible root causes will be reported to operators for further operations, such as hardware inspection and reconfigurations.

When abnormal metrics on an RNIC trigger the Hostping engine, the host is usually busy with services, and some RNICs, especially the abnormal RNIC, may have heavy ser-

Algorithm 1 Detect Links with Bandwidth Degradation

Input: *normal* and *abnormal paths*
Output: *abnormal* and *gray links*

```
1:  function DetectAbnormalLinks()
2:      InitLinkStatus()
3:      for path_j in normal paths do
4:          for link_i in path_j do
5:              link_i.status ← normal
6:      for path_j in abnormal paths do
7:          if ∃ links ∈ path_j in uncertain status then
8:              for link_i in all these links do
9:                  link_i.status ← abnormal
10:                 link_i.abnormal_cnt++
11:         if ∃ links ∈ path_j in abnormal status then
12:             for link_i in all these links do
13:                 if marked abnormal by a new RNIC then
14:                     link_i.abnormal_cnt++
15:         if ∀ links ∈ path_j in normal status then
16:             for link_i in path_j do
17:                 link_i.status ← gray
18:     return abnormal links and gray links
19: function InitLinkStatus()
20:     for link_i in all links do
21:         link_i.status ← uncertain
22:         link_i.abnormal_cnt ← 0
```

[5] We draw this conclusion from the server introduced in Section 3.2.1. For some types of servers, the RNIC cannot reach the line rate when communicating with the memory in remote NUMA nodes.

vice traffic. In this case, the path bandwidth measured by these RNICs will degrade due to the contention of service traffic, even if there is no bottleneck. Thus, we cannot judge the status of these paths according to the bandwidth baseline.

Nevertheless, these RNICs could still indicate abnormal paths. In the server introduced in 3.2.1, applications usually use an RNIC to communicate with its affinitive endpoints (memory nodes, GPUs under the same root port) to achieve optimal performance. Furthermore, memory channels and inter-socket buses generally provide considerable bandwidth redundancy. Therefore, the measured bandwidth between the RNIC and its affinitive endpoints is usually identical if no bottleneck occurs, no matter how much influenced by service traffic on the RNIC. Thus, among the RNIC's affinitive endpoints, if the measured path bandwidth to one endpoint is significantly lower than that to the other endpoints (by $Abnormal_{th}$), we infer this path is abnormal. However, we have no idea whether other paths are normal due to degraded RNIC loopback bandwidth, leading to reduced diagnosis accuracy. As a workaround, we could check whether there are idle RNICs on the host, which could still judge path status according to the baseline.

As applications usually use an RNIC to communicate with its affinitive endpoints, bottlenecks generally occur on the paths between the abnormal RNIC and its affinitive endpoints. Thus, we could focus on finding bottlenecks on these paths. When triggered by abnormal metrics, the Hostping engine only conducts loopback tests between RNICs and their affinitive endpoints. First, this method is sufficient to diagnose the status of the memory channel and the inter-socket bus with low overhead to service traffic. In addition, the service traffic may affect the measured bandwidth between the affinitive GPU of the abnormal RNIC and the RNIC under other root ports. As a result, the analyzer may incorrectly judge these paths as abnormal, leading to an inaccurate diagnosis. Thus, for the links under the same root port as the abnormal RNIC, we only use this abnormal RNIC to judge their status.

The inference of abnormal links is still based on Algorithm 1. However, as RNICs with heavy traffic cannot judge whether a path is normal, some normal links may be marked as abnormal. In this case, links with the highest *abnormal_cnt* are most likely abnormal and should receive more attention. When abnormal metrics trigger the Hostping engine, abnormal links are usually fully loaded. Based on this, we can infer the root cause by monitoring these links. Links with utilization higher than $Util_{high}$ will be diagnosed as *overloaded links*, while *link failures* or *misconfigurations* may be the root cause of other abnormal links. However, as the abnormal RNIC suffers from intra-host bottlenecks, the latency measured by it will rise anomalously. Thus, we cannot judge whether the degraded GPU PCIe link is caused by a link failure or a misconfiguration. Operators then need to do a further inspection. Notably, if no abnormal link could be found, the RNIC PCIe link may be the bottleneck, and the analyzer will further check if it is overloaded with loopback traffic to determine whether traffic contention or a link failure should be blamed.

5 Implementation

For the hardware monitor, throughput and abnormal metrics are provided by our RNIC vendors, and GPU status is obtained based on Nvidia Management Library (NVML) [6]. For the threshold, Thp_{low} is 5% of the RNIC line rate to judge whether the RNIC is idle. PFC_{high} is 3% (every second, transmission is paused by 30ms) to trigger the Hostping engine. During the deployment, the monitor checks the host status every five minutes[6] and collects abnormal metrics every second to decide whether to start the Hostping engine.

For the Hostping engine, we implement the probing module with the verbs API and rdma-core libraries [3]. The bus monitor is implemented based on the API and metrics provided by our vendors: Intel's and AMD's API for CPU root ports, memory channels, and inter-socket buses, NVML for GPU PCIe links, and Mellanox's metrics for RNIC PCIe links.

The data analyzer takes the metrics collected by the Hostping engine as input and infers the most susceptible root causes for intra-host bottlenecks. $Abnormal_{th}$ is 20% to judge whether the latency or bandwidth of a path is abnormal, and $Util_{high}$ is 90% to judge whether a bus is overloaded.

The cloud data storage is implemented based on our time-series database. Every time the Hostping engine starts, all the information collected and deduced by the Hostping agent will be uploaded to the cloud. These data help us better understand the frequency and root causes of bottlenecks. Moreover, operators may need historical data to determine the root causes in some scenarios.

6 Evaluation & Intra-host Bottlenecks Found

We evaluate Hostping on over 300 servers in our distributed machine learning system. The host topology is shown in Fig.2 and introduced in Section 3.2.1, which is the most complex intra-host topology in our data center servers. In this section, we will summarize the bottlenecks we found during the deployment of Hostping. For known bottlenecks, Hostping could effectively diagnose their root causes. In addition, Hostping also reveals six bottlenecks we did not notice before. We roughly classify the bottlenecks found by Hostping into three scenarios according to their root causes.

Scenario 1: Intra-host bandwidth degrades due to link failures. As the host topology becomes more complex, link failures occur frequently. During the deployment, we encountered dozens of instances where failed links resulted in degraded intra-host bandwidth, including failures of *#1 RNIC PCIe links* (Fig.8 (a)), *#2 GPU PCIe links* (Fig.8 (b) & Fig.10

[6] As link failures and misconfigurations infrequently appear in a host, 5 minutes is a fine granularity.

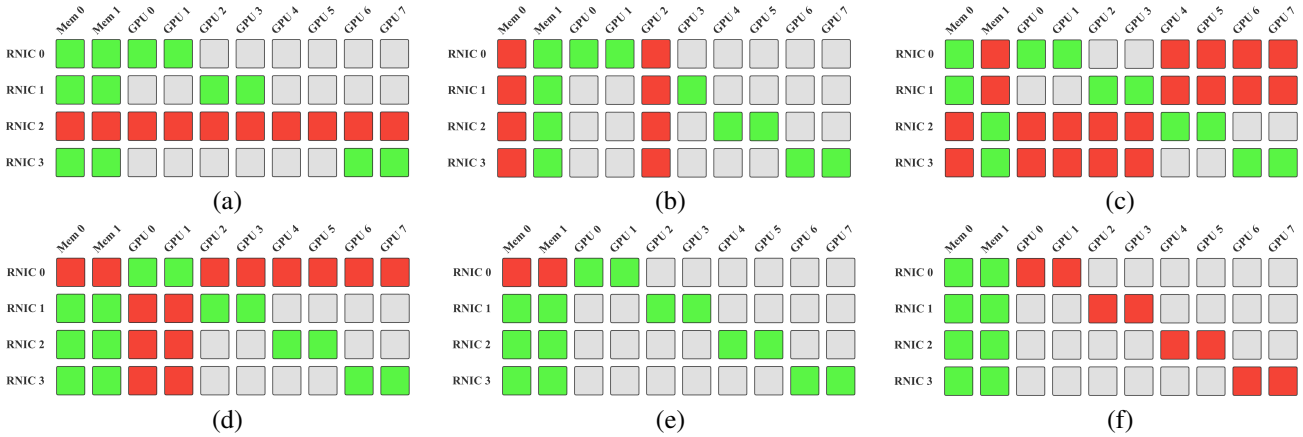


Figure 8: The intra-host end-to-end bandwidth matrices measured by the Hostping engine when hosts are idle. The topology between RNICs and endpoints is shown in Fig.2. (a)-(e) show intra-host bandwidth degradation due to link failures. (f) shows the impact of inappropriate configurations. Green, red, and gray indicate that the path status is normal, abnormal, and uncertain, respectively.

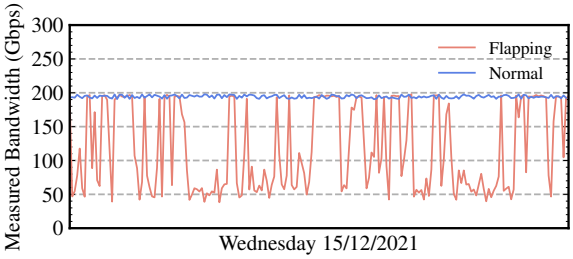


Figure 9: Memory channel flapping on the server. When this occurs, the bandwidth of the memory channel switches between normal and abnormal.

(a)), *#3 memory channels* (Fig.8 (b)), and *#4 UPI* (Fig.8 (c)). For the sake of space, Fig.8 (b) contains #2 and #3. Their problem may be loose PCIe interfaces, dust on connecting fingers, or hardware failures and requires further troubleshooting. Based on the matrix, the analyzer could accurately infer abnormal links. Note that in Fig.10 (a), although RNIC2 has a large amount of service traffic, it could still judge that the path to GPU4 is abnormal according to other measured paths. With Hostping, operators could quickly discover and deal with link failures, avoiding application performance degradation.

[New] *#5 CPU root port failures.* Before deploying Hostping, we only knew four kinds of link failures (#1 to #4). During the deployment, we found that the CPU root port may also experience hardware bandwidth degradation. When this happens, the bandwidth between the RNIC and the GPU under the failed root port is normal. While traffic passes through the failed root port may suffer from degraded bandwidth. The corresponding bandwidth matrix is shown in Fig.8 (d) and (e). They have the same root cause, except that (e) has slight bandwidth degradation, and RNICs under other root ports cannot find anomalies. Nevertheless, Hostping could still accurately diagnose the root cause in this case.

[New] *#6 Memory channel flapping.* With the assistance of Hostping, we found a host suffers from degraded memory channel bandwidth due to a link failure. However, no performance issues could be discovered in subsequent manual testing. By continuously running Hostping and collecting measured data, we found that the root cause lies in the flapping memory channel. As shown in Fig.9, the bandwidth of the host memory channel switches between normal and abnormal. With historical data, we could understand the causes of intra-host bottlenecks more clearly. This case shows the necessity to run Hostping periodically.

Scenario 2: Inappropriate configurations lead to degraded performance. *#7 Enabling ACS results in high PCIe latency.* We have mentioned this case in 2.1. With ACS enabled, all GDR traffic will be guided to the CPU instead of directly to the GPU, resulting in drastic performance degradation. As shown in Fig.8 (f), all PCIe bridges are configured as ACS enabled in this case. As a result, both the latency and bandwidth between the RNIC and the GPU under the same root port turn abnormal. Hostping could accurately diagnose this bottleneck and remind operators to check the configuration.

[New] *#8 Disabling ATS results in high PCIe latency.* We found this case in a virtualized environment. In IO virtualization, if Address Translation Service (ATS) is disabled on the RNIC, all GDR packets will be directed to the CPU root complex for address translation. Similar to #7, with ATS disabled, the latency between the RNIC and the GPU under the same root port increases, leading to drastic bandwidth degradation. By enabling ATS, the translation can be finished in the RNIC to achieve optimal GDR performance.

[New] *#9 Enabling "slow start" on the RNIC.* This is a lossy feature provided by our RNIC vendor. When enabled on an RNIC, the RNIC sending rate will start from a small value instead of the line rate. Although "slow start" could alleviate congestion under Incast scenarios, it increases the completion time of short flows. Thus, it usually remains disabled in most

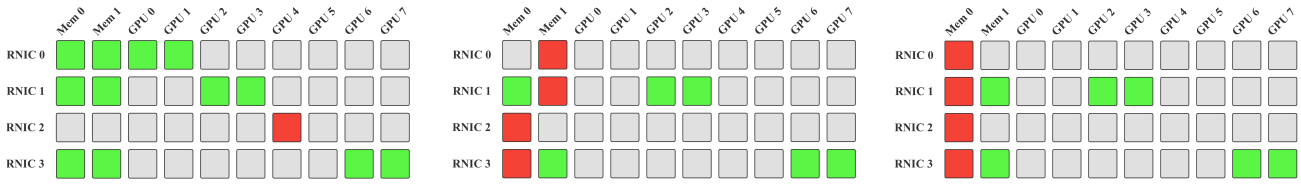


Figure 10: The bandwidth matrices measured when hosts are busy with services. (a) shows hardware bandwidth degradation due to the GPU PCIe link failure. (b) and (c) show degraded intra-host bandwidth caused by traffic contention.

scenarios. When "slow start" is enabled, the RNIC bandwidth to all intra-host endpoints will be lower than the baseline (similar to Fig.8 (a)). At first, Hostping diagnosed the root cause as the RNIC PCIe link failure. However, we found no bottlenecks during continuous bandwidth tests. After configuration inspection, we finally uncovered the root cause.

[New] *#10 Setting "Tx window" too small on the RNIC.* This is also a lossy feature. "Tx window" will limit the maximum in-flight bytes of each queue pair on an RNIC. Therefore, "Tx window" influences the maximum bandwidth a QP could achieve and needs to be set reasonably to alleviate network congestion without degrading throughput. Similar to #9, when an RNIC's "Tx window" is too small, its bandwidth to all intra-host endpoints will be lower than the baseline.

Scenario 3: Intra-host bandwidth degrades due to traffic contention. [New] *#11 Overloaded Inter-socket buses.* During the deployment, we found some malfunctioning applications overloaded the UPI of a host, and cross-socket receiving traffic on RNIC0 triggered a large number of Tx pause frames. Fig.10 (b) shows the corresponding bandwidth matrix measured by Hostping. In this case, RNIC0 and RNIC2 have a large amount of service traffic. Nevertheless, they could still judge that two paths (RNIC0 to mem1 and RNIC2 to mem0) are abnormal according to other measured paths. With the help of the other two idle RNICs, the analyzer infers that the UPI is most likely to be abnormal (with the highest *abnormal_cnt*). Furthermore, leveraging the bus monitor, it diagnoses the root cause as the overloaded UPI. The operator then will find out the traffic source that overloads the UPI.

#12 Overloaded memory channels. TCP and RDMA traffic may co-exist in the same host to keep high availability [21]. In this case, the processing of TCP may consume a lot of memory bandwidth, leading to a slow receiving rate for RDMA traffic. However, we did not discover this case in A100 servers during the deployment of Hostping. As a supplement, we conduct an experiment to evaluate how Hostping behaves when the memory channel is overloaded. We use several processes to overload the channel of mem0. Besides, RNIC0 and RNIC2 receive traffic writing to mem0 and mem1, respectively. Fig.10 (c) shows the corresponding bandwidth matrix. Although RNIC0 and RNIC2 have a large amount of receiving traffic, they could still judge that their paths to mem0 are abnormal. Similar to #11, in this case, the analyzer infers that the channel of mem0 is most likely abnormal and diagnoses the root cause as the overloaded memory channel.

In this scenario, we evaluate the performance of Hostping when intra-host links are overloaded. Furthermore, the results show that Hostping could still effectively diagnose intra-host performance bottlenecks under the interference of service traffic on RNICs.

7 Experiences Learned

Conduct Hostping before running applications. As the intra-host topology becomes more complex, the likelihood of link failures in the host network boosts. Based on our experience, some failures may already exist when the server leaves the factory. Thus, it is essential to evaluate the intra-host network performance before delivering the server to customers. In addition, as intra-host services become more complicated, configuration items in the host also increase considerably, and the configuration methods are varied. For example, the setting of Address Translation Service requires a reboot to take effect. In contrast, Access Control Service is enabled by default and needs to be disabled after each reboot. Furthermore, configurations may not be completed successfully for some reason. Therefore, misconfigurations occur occasionally. We recommend conducting Hostping after each reboot to ensure proper configurations before running applications.

Perceive intra-host bottlenecks with VoQ ECN marking. Although intra-host bottlenecks should be addressed in a targeted manner (e.g., hardware replacement, reconfigurations), we argue that *congestion control mechanisms should be able to perceive intra-host bottlenecks*. Thus, they could alleviate the triggering of packet drops and Tx pause frames to provide better network performance before the bottleneck could finally be resolved. Generally, packets could only be ECN-marked in a switch's egress port, and ECN-based congestion control mechanisms could only perceive network congestion. Fortunately, some latest RNICs provide a new function called VoQ (Virtual Output Queuing) ECN marking, enabling the receiver RNIC to ECN-mark packets when its receive buffer exceeds a threshold. By enabling this function, ECN-based congestion control mechanisms, such as DCQCN [50], can also perceive intra-host bottlenecks. Thus, the sender could timely slow down its sending rate to alleviate the triggering of packet drops or Tx pause frames.

Pay attention to intra-host topologies. Although GDR is

supported at any distance within the host, it is highly recommended not to conduct GDR across CPU root ports. When this occurs, the GDR bandwidth degrades severely, which may lead to a large number of Tx pause frames or packet drops. Furthermore, when testing some AMD servers, we found a large number of Tx pause frames and drastic throughput degradation when the RNIC (200 Gb/s) writes to the memory in the remote socket. This is due to the low cross-socket hardware bandwidth on these servers, and the root cause lies in the host architecture. Thus in these servers, we should avoid using RNICs to communicate with the memory in the remote socket for optimal performance.

8 Related Work

Bottlenecks in the RNIC and intra-host network. With the increasing RNIC line rate, the intra-host network and the RNIC have become potential performance bottlenecks in network communication. Some literature has studied these bottlenecks. Kong et al. [32] implement a tool to help data center operators uncover potential performance bottlenecks in the RNIC. Martinasso et al. [37] analyze congestion behaviors in PCIe fabric and develop a congestion-aware performance model for PCIe communication. Zhang et al. [49] study RDMA sharing characteristics and analyze performance isolation anomalies in RDMA. Neugebauer et al. [40] study the performance impact of PCIe in the host network. Dong et al. [16] analyze different types of traffic congestion in the host network and propose a new server architecture to alleviate intra-host congestion. Faraji et al. [19] show the implication of distance between GPUs on the GPU-to-GPU communication performance in the host network. Farshin et al. [20] study when Intel Data Direct I/O (DDIO) technology becomes a bottleneck in multi-hundred-gigabit networks and how to optimize DDIO-enabled systems for I/O intensive applications. These studies help us better understand the potential bottlenecks in the RNIC and intra-host network.

Bottleneck diagnosis tools. Diagnosis tools could be broadly classified as system-based tools and intra-host tools. System-based tools aim to diagnose performance bottlenecks in the whole system. Pingmesh [25] implements an end-to-end connectivity and latency monitoring system for network troubleshooting and SLA tracking. Netbouncer [44] leverages the IP-in-IP technique to probe designated paths and then diagnoses device and link failures in data center networks. Deepview [48] builds a near-real-time system for virtual disk failure localization. Microscope [23] leverages queuing information at network functions to identify the root causes of performance bottlenecks. SNAP [47] collects network information such as TCP statistics and socket-call logs to pinpoint the problem in data center network applications. In contrast, intra-host tools are dedicated to diagnosing bottlenecks in the host. Haecki et al. [26] implement a latency diagnosis tool to identify the source of network latency in end-host stacks. Mellanox Neohost [4] provides plenty of diagnosis counters on Mellanox RNICs. Nvidia SMI [9] provides the status of Nvidia GPUs. Intel PCM [2] and AMD uProf [1] provide the internal resource utilization of the CPU, including the utilization of buses and interfaces connected to the CPU, such as inter-socket buses, memory channels, and CPU root ports.

9 Conclusion & Future Work

Intra-host networking has become a potential bottleneck for RDMA networks, and intra-host bottlenecks can severely degrade network performance. This paper proposes Hostping to monitor and diagnose intra-host bottlenecks. We analyze the symptom of intra-host bottlenecks based on our long-term troubleshooting experience and realize that most intra-host bottlenecks have one or both of the following symptoms: intra-host bandwidth degradation and intra-host latency increase. Thus, Hostping measures intra-host bandwidth and latency as performance metrics to detect and diagnose intra-host bottlenecks. Furthermore, we propose an efficient diagnosing mechanism that could effectively identify the root cause of intra-host bottlenecks even under the interference of service traffic on RNICs. During the deployment, Hostping not only discovers performance bottlenecks we already knew but also reveals six bottlenecks we did not notice before.

The deployment of Hostping makes us realize that more work needs to be done. Firstly, when the host is busy with services, due to the influence of service traffic, it is challenging to accurately diagnose intra-host link status based on binary path status. If the end-to-end traffic information within the host can be obtained, it will provide more insights into intra-host bottlenecks. Secondly, after finding an overloaded link, we hope Hostping could automatically identify the traffic source, such as malfunctioning applications. Finally, in addition to intra-host network bottlenecks, RNIC bottlenecks, such as scalability problems [29] [30] [31], can also lead to severe network performance degradation. Thus, it is also important to diagnose bottlenecks in the RNIC.

Acknowledgments

We would like to thank our shepherd, Raja Sambasivan, and the anonymous reviewers who helped us improve the quality of this paper. We would also like to thank Huaping Zhou for his insightful feedback. This work is supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61872401 and Grant 62132022, a BUPT-ByteDance Research Project, and the Fok Ying Tung Education Foundation under Grant 171059.

References

[1] AMD uProf. https://developer.amd.com/amd-uprof/.

[2] Intel Performance Counter Monitor. https://github.com/opcm/pcm.

[3] Linux rdma-core. https://github.com/linux-rdma/rdma-core.

[4] Mellanox Neohost. https://support.mellanox.com/s/productdetails/a2v50000000N2OlAAK/mellanox-neohost.

[5] Nvidia DGX-A100. https://www.nvidia.com/en-us/data-center/dgx-a100/.

[6] Nvidia Management Library. https://developer.nvidia.com/nvidia-management-library-nvml.

[7] Nvidia nccl-tests. https://github.com/NVIDIA/nccl-tests.

[8] Nvidia NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.

[9] Nvidia System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.

[10] OFED perftest. https://github.com/linux-rdma/perftest.

[11] Intel® Xeon® Scalable Processors Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/2nd-gen-xeon-scalable-datasheet-vol-1.pdf, 2019.

[12] Workload Tuning Guide for AMD EPYC™ 7002 Series Processor Based Servers. https://developer.amd.com/wp-content/resources/56745_0.80.pdf, 2020.

[13] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-Aware GPU Scheduling for Learning Workloads in Cloud Environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[14] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 329–344, 2020.

[15] Ítalo Cunha, Renata Teixeira, Nick Feamster, and Christophe Diot. Measurement Methods for Fast and Accurate Blackhole Identification with Binary Tomography. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 254–266, 2009.

[16] Jianbo Dong, Zheng Cao, Tao Zhang, Jianxi Ye, Shaochuang Wang, Fei Feng, Li Zhao, Xiaoyong Liu, Liuyihan Song, Liwei Peng, et al. EFLOPS: Algorithm and System Co-Design for a High Performance Distributed Training Platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 610–622. IEEE, 2020.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[18] Nick Duffield. Network Tomography of Binary Network Performance Characteristics. *IEEE Transactions on Information Theory*, 52(12):5373–5388, 2006.

[19] Iman Faraji, Seyed H Mirsadeghi, and Ahmad Afsahi. Topology-Aware GPU Selection on Multi-GPU Nodes. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 712–720. IEEE, 2016.

[20] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689, 2020.

[21] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[22] Andrew Gibiansky. Bringing HPC techniques to deep learning. *Baidu Research, Tech. Rep*, 2017.

[23] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based Performance Diagnosis for Network Functions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 390–403, 2020.

[24] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of*

the 2016 ACM SIGCOMM Conference, pages 202–215, 2016.

[25] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, and Hua and Chen. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. *Computer communication review*, 45(4):139–152, 2015.

[26] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 861–877, 2022.

[27] Yiyi Huang, Nick Feamster, and Renata Teixeira. Practical Issues with Using Network Tomography for Fault Diagnosis. *ACM SIGCOMM Computer Communication Review*, 38(5):53–58, 2008.

[28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

[29] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.

[32] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, 2022.

[33] Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, 2013.

[34] Jason Lawley. Understanding Performance of PCI Express Systems. *WP350 (v1. 2). Xilinx*, 97, 2014.

[35] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[37] Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R Alam, Thomas C Schulthess, and Torsten Hoefler. A PCIe Congestion-Aware Performance Model for Densely Populated Accelerator Servers. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 739–749. IEEE, 2016.

[38] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash. *arXiv preprint arXiv:1811.05233*, 2018.

[39] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 313–326, 2018.

[40] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[41] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[42] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 317–332, 2016.

[43] Richard Solomon. PCI Express Basics. *PCI-SIG, Oct*, 2011.

[44] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[45] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat-Resilient Allreduce Tree for Distributed Machine Learning. In *4th Asia-Pacific Workshop on Networking*, pages 52–57, 2020.

[46] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast Distributed Deep Learning over RDMA. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[47] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling Network Performance for Multi-Tier Data Center Applications. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[48] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 519–532, 2018.

[49] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1307–1326, 2022.

[50] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

[51] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010.

Understanding RDMA Microarchitecture Resources for Performance Isolation

Xinhao Kong Jingrong Chen Wei Bai[†] Yechen Xu[#] Mahmoud Elhaddad[†]
Shachar Raindel[†] Jitendra Padhye[†] Alvin R. Lebeck Danyang Zhuo

Duke University [†]*Microsoft* [#]*Shanghai Jiao Tong University*

Abstract

Recent years have witnessed the wide adoption of RDMA in the cloud to accelerate first-party workloads and achieve cost savings by freeing up CPU cycles. Now cloud providers are working towards supporting RDMA in general-purpose guest VMs to benefit third-party workloads. To this end, cloud providers must provide strong performance isolation so that the RDMA workloads of one tenant do not adversely impact the RDMA performance of another tenant. Despite many efforts on network performance isolation in the public cloud, we find that RDMA brings unique challenges due to its complex NIC microarchitecture resources (e.g., the NIC cache).

In this paper, we aim to systematically understand the impact of RNIC microarchitecture resources on performance isolation. We present a model that represents how RDMA operations use RNIC resources. Using this model, we develop a test suite to evaluate RDMA performance isolation solutions. Our test suite can break all existing solutions in various scenarios. Our results are acknowledged and reproduced by one of the largest RDMA NIC vendors. Finally, based on the test results, we summarize new insights on designing future RDMA performance isolation solutions.

1 Introduction

Multiplexing workloads from different tenants on a shared computing infrastructure enables the modern cloud computing era. The global cloud infrastructure revenue has already surpassed 400 billion US dollars and is forecast to grow to reach around 1 trillion US dollars in the next decade [7].

It is well known that having different tenants' workloads share computing resources can lead to unpredictable application performance interference [12, 18, 66] and privacy leakage [32, 39]. This drives plenty of studies focusing on performance isolation in the cloud, especially for performance-critical applications that have stringent service-level objectives [11, 12, 18, 41, 63, 66, 70]. The state of the art in practice has also significantly advanced: CPU vendors even implement hardware mechanisms to control and isolate access to CPU caches [20]. Side channels through shared resources are being patched over time [39].

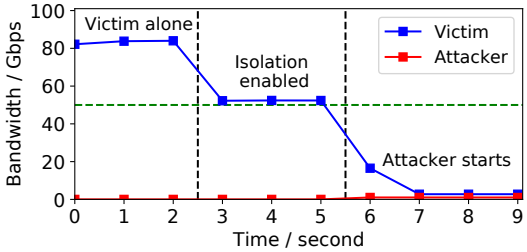


Figure 1: Violations of performance isolation under existing methods

In this paper, we visit one particular hardware device, the RDMA NIC (RNIC). RDMA offloads the network stack from OS kernel to NIC hardware to provide high throughput and ultra-low processing latency with near-zero CPU overhead. RDMA has been deployed in datacenters at scale to improve performance and free up CPU cores for first-party workloads like storage and ML [14, 17, 38, 51]. Now cloud providers are working towards supporting RDMA in general-purpose guest VMs to benefit third-party workloads. To this end, cloud providers must provide strong performance isolation for tenants sharing the same RNIC.

Many efforts have been made to improve network performance isolation in the public cloud, with a special focus on bandwidth and packet processing capacity [3, 15, 16, 25, 34, 62, 64]. However, RDMA brings new challenges due to its unique and complex NIC *microarchitecture resources* (e.g., NIC caches and processing units). Their existence and impact on performance are already known to the research community [29, 33]. To avoid performance anomalies, developers carefully design RDMA systems to avoid exhausting these microarchitecture resources [5, 9, 10, 27, 30, 50, 61]. Our study is from a different angle: we look at how these microarchitecture resources affect RDMA performance isolation from a public cloud provider's perspective. The cloud provider has no knowledge and control of tenants' RDMA applications, and tenants can consume RNIC microarchitecture resources in arbitrary manners.

To demonstrate RNIC microarchitecture resources' significant impact on performance isolation, we test the state-of-the-art approach: using SR-IOV with separated hardware traffic class (HW TC). Both SR-IOV and HW TC are hardware mechanisms available on commodity RNICs. HW TC leverages multiple hardware queues (usually 8 queues) in RNICs. We can assign each tenant application to use one queue. We run one victim traffic between two virtual machines using `ib_write_bw`, a standard RDMA bandwidth testing tool in Perftest [56]. Each virtual machine is on a different server, and the two servers are equipped with 100 Gbps NVIDIA ConnectX-5 RNICs. Figure 1 shows the bandwidth. The bandwidth test achieves 80 Gbps. We start one virtual machine on each server to represent an attacker (i.e., a buggy or malicious tenant application) and enable performance isolation to grant half of the total bandwidth to the victim and the attacker. The victim traffic reduces to 50 Gbps, which is expected. However, when we start a carefully designed attacker traffic of only 1 Gbps to intentionally exhaust one of the RNIC microarchitecture resources, the victim immediately drops to 2 Gbps, violating the performance isolation guarantee (i.e., 50 Gbps of guaranteed network bandwidth for the victim).

We develop a set of experiments to study how RNIC microarchitecture resources are used by different types of RDMA operations. Our experiments surface several interesting findings, including: (1) Exception or error handling pauses the RNIC's pipelines and causes other tenants' performance to drop drastically. (2) Control verbs cause a severe increase in cache misses and impair other tenants' performance. (3) Data verbs can exhaust different types of microarchitecture resources and violate performance isolation. To the best of our knowledge, we are the first to systematically study the impact of all types of control verbs and exceptions on RDMA microarchitecture resource consumption.

We leverage these findings to create an RDMA operation model to describe the relationship between the RDMA verb operations and the microarchitecture resources consumed. Our model allows us to understand how to exhaust each of the RNIC resources. Using the operation model, we create the first test suite, Husky, to systematically test and evaluate RNIC performance isolation solutions. Unfortunately, running our test suite on commodity RNICs reveals bad news: *there is currently no solution that can provide RNIC performance isolation.* We have already reported all of our findings to three major RNIC vendors, NVIDIA, Chelsio, and Intel. Our results are fully reproduced and acknowledged by NVIDIA, one of the largest RDMA NIC manufacturers. Finally, we present new insights on how future performance isolation solutions should be built. We hope these insights can benefit future RNIC design and RDMA software development.

This paper makes the following contributions:

- We identify multiple interactions between RDMA operations and the RNIC microarchitecture resources, including the previously unknown impact of error handling and control operations.
- We introduce the first RDMA operation model to describe how RNIC microarchitecture resources are consumed in verb operations (the standard RDMA programming API) and why these microarchitecture resources affect performance isolation.
- We build the first test suite to systematically test and evaluate RNIC performance isolation solutions. We show that none of the existing performance isolation solutions can pass our test suite. Husky test suite is available at `https://github.com/host-bench/husky`.

This work demonstrates that providing performance isolation for RDMA in the public cloud is much more difficult than one may think. There must be a higher standard for future RDMA performance isolation solutions: they should carefully consider RNIC microarchitecture resources and be evaluated by systematic benchmarks.

2 Background and Motivation

We first present the background knowledge of the network performance isolation in the public cloud. Then we introduce RDMA and discuss new challenges presented by the RDMA network performance isolation.

2.1 Network Performance Isolation in the Public Cloud

Tenants in the cloud mainly cause contention on two types of network resources. The first the most obvious one is the bandwidth in the network fabric. To mitigate bandwidth contention among tenants, one line of work [58, 60, 62] statically limits per-tenant bandwidth. Another line of work [1, 3, 4, 6, 16, 24, 25, 37, 58, 59, 68] gives each tenant a minimum bandwidth guarantee and allows tenants to use spare bandwidth capacity. The second type of resource is the packet processing resources at the end host. Per-packet processing costs depend on many factors, such as cache misses and operations to perform. Recently, PicNIC [34] provides isolation for such software packet processing. People also leverage specialized hardware to achieve the same goal [64].

It is worthwhile to note that network performance isolation is very different from network virtualization. Network virtualization orchestrates network resources to provide each tenant with an illusion of an independent network. A tenant should not impact the connectivity of the network of another tenant. The goal of network virtualization is to achieve low overhead [19, 31, 57]. In comparison, network performance isolation focuses on how to manage resource contentions to ensure that tenants can achieve guaranteed performance.

2.2 RDMA Overview

RDMA allows the NIC to directly transfer data between the wire and the application memory. The networking protocol is implemented in the NIC. Figure 2 presents the overview of

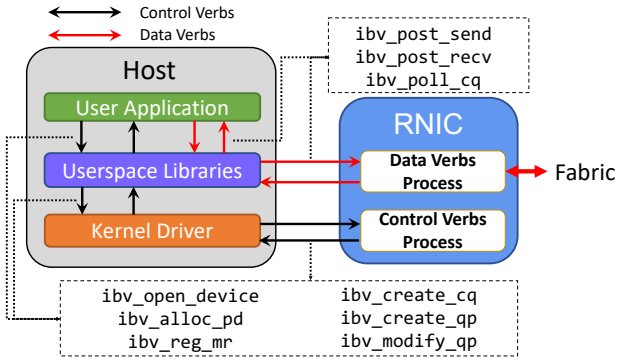


Figure 2: Overview of RDMA workflow. Verbs processing logics are heavily offloaded to the RNIC.

the RDMA workflow. It classifies standard RDMA programming interface, a.k.a., verbs, into two categories: control and data. An application first needs to call several *control verbs* to allocate necessary objects, such as queue pair (QP) and completion queue (CQ), to set up a reliable connection (RC), an unreliable connection (UC), or an unreliable datagram (UD) transmission endpoint. Then the application needs to register a memory region (MR). This registration essentially pins the memory in the host DRAM and obtains the mapping from virtual addresses to physical addresses, which enables the RNIC to directly read from or write to this memory region. All these control verbs are processed by the following procedure: RDMA's userspace libraries and kernel drivers process the verb request, generate a request command, put the command in a negotiated command queue, and ring the RNIC's doorbell (e.g., memory-mapped registers). The RNIC fetches the command from the command queue, processes it, and pushes the response back to the queue. The drivers then process the response and return the object to the application.

After the above initialization, the application can start data transmissions between local and remote memory. There are several types of operations that applications can use, such as SEND/RECV, WRITE, READ, and ATOMIC. We name these operations as *data verbs*. To issue a data verb, the application generally posts a request to its send queue and rings the RNIC's doorbell through userspace libraries. The RNIC then parses the request, reads data from the host memory, segments data into packets, and transmits packets. This procedure bypasses the kernel. There are certain differences in processing different types of requests. For example, for SEND/RECV messages, the receiver should post enough RECV requests before the sender issues SEND requests. Otherwise, the incoming SEND requests may be dropped or need retransmissions because the receiver RNIC lacks receive requests to process them, which is known as the receive not ready (RNR) error. For WRITE/READ data to/from the remote end or execute ATOMIC operations, the sender should specify correct remote virtual addresses and memory keys. An invalid address or a wrong key will trigger a memory protection error and cause the QP to transition into the error state.

2.3 Why RDMA Performance Isolation is Hard?

As shown above, RDMA offloads many host network functionalities to the RNIC, which has many invisible hardware components, and each component may individually become a performance bottleneck. Figure 3 shows the hardware components of a commodity RNIC. We draw this figure based on publicly available documents from NVIDIA [44, 46, 48]. In addition to the packet buffers (TX/RX Buffer), the RNIC also has multiple processing units (PU) and many types of internal caches. Each internal cache is used to store a specific type of metadata. For example, in NVIDIA RNICs, the Interconnect Context Memory (ICM) cache stores QP contexts; the Memory Translation Table (MTT) and Memory Protection Table (MPT) store entries for memory address translation and protection information; and the Work Queue Entry (WQE) cache stores prefetched send WQEs and posted receive WQEs. As these caches are derived from the design needs, other RNICs include similar components. We name these RNIC hardware components *microarchitecture resources* based on the analogy for CPU hardware. CPUs are designed to conform to a standard instruction set architecture (e.g., ARM, x86), but the CPU designers can make the microarchitecture-level decisions, such as how many levels of caches and the cache sizes. RNICs are similar because RNIC vendors have to provide the same programming interface for RDMA application developers, but the vendors can decide on these microarchitecture-level details, e.g., RNIC caches.

Many previous efforts have already identified some impacts of these microarchitecture resources on RDMA application performance. For example, [5, 29, 50] find that an RNIC caches QP contexts. A QP context cache miss can trigger an additional PCIe round trip for the RNIC to fetch the context from the host DRAM, thus degrading application performance. For example, 200 connections can cause an 90% request rate drop on NVIDIA ConnectX-3 NIC [5]. However, these efforts study microarchitecture resources from the perspective of an *application developer*. After a performance degradation, they identify the bottleneck resource, seek more efficient methods to use data verbs, and modify their applications correspondingly.

However, in public clouds, cloud providers have no control over tenants' applications. Tenants thus can consume RNIC's microarchitecture resources as they wish, even maliciously. Therefore, from the perspective of the *cloud provider*, we need to understand the microarchitecture resource consumption of most of (if not all) RDMA verbs, not just common data verbs. Only with this knowledge can we properly allocate RNIC's microarchitecture resources to different tenants to deliver predictable performance.

3 RNIC Microarchitecture Resources

In this section, we present a study on all the RNIC microarchitecture resources that we are currently aware of. Prior works have already identified several particular forms of resource

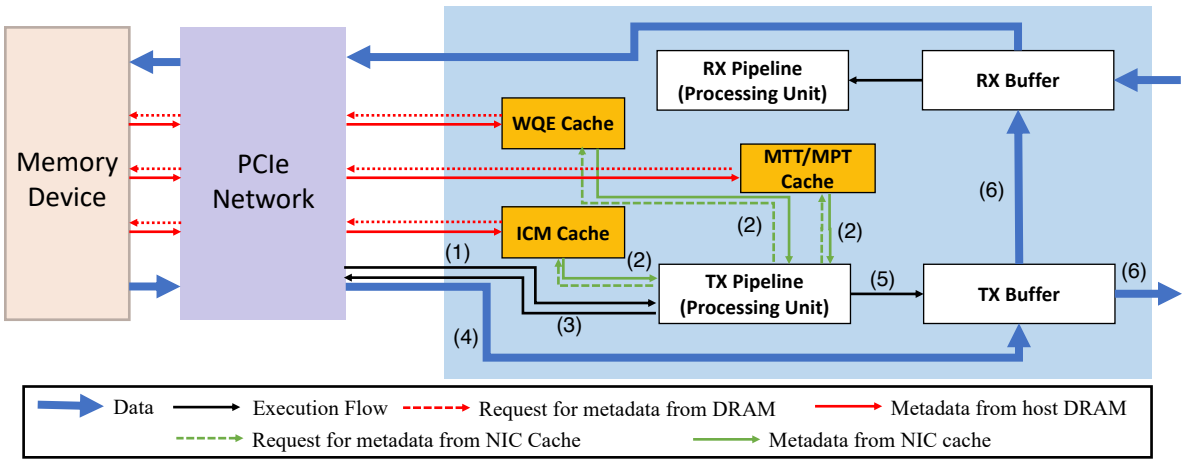


Figure 3: RDMA NIC microarchitecture hardware details: when the doorbell is rung, the RNIC first fetches the control/data verbs request from the host DRAM. (1) To fetch and process this request, the RNIC may need several metadata (e.g., QP contexts) and there are different types of caches inside the RNIC that can store this metadata. The RNIC can get the metadata directly from these caches, (2) or fetch them from DRAM if a cache miss happens (red lines in the figure). Then the RNIC processes the request and (3) sends the response back to the host DRAM for control verbs or issues DMA requests to read payload for data verbs. After (4) reading data from the host DRAM, the RNIC (5) processes the data into network packets and (6) sends them to the fabric. The symmetric receiver side is not shown for simplicity.

contention. But our goal here is to systematically study all possible types of resource contention. For each microarchitecture resource, we study how it is consumed by three categories of RDMA operations: (1) control verbs that allocate objects for applications (e.g., `ibv_create_qp`), (2) data verbs that initiate data transfer (e.g., `ibv_post_send`), and (3) exception handling operations that handle exceptions or errors (e.g., RNR errors). Due to space limitations, we first present a few *key findings* that have significant implications on RNIC performance isolation. After that, we summarize several other findings. We present a detailed analysis of NVIDIA's responses to these findings in Appendix B.

3.1 Methodology

Our findings center around how to exhaust RNIC microarchitecture resources through the verbs interface [21], the standard RDMA programming API. For each key finding, we demonstrate it with a concrete setting, which consists of a victim workload and an attacker workload. *Although we use the terminology attacker, the attacker tenant does not get unauthorized access to other tenants through vulnerabilities. Instead, the attacker is just a normal RDMA application that issues standard RDMA verbs.* Each tenant has one client and one server. The clients of the victim and the attacker locate on the same physical machine and share the same RNIC. The servers of the victim and the attacker are colocated on a different physical server. During the measurement, we do not enable any isolation mechanism. We will study existing performance isolation solutions in §5.

We focus on the performance interference between the victim and the attacker through the exhaustion of microarchitecture resources. We first run only the victim to saturate the link bandwidth capacity (bits per second) or the RNIC's maximum request rate (requests per second). We then start the attacker and measure the two metrics for both the victim and the attacker. If there is no microarchitecture resource contention, the sum of the performance metrics of the two tenants should match the RNIC's limit in the specification. Modern RNICs specify their bandwidth capacity and request rate limits. If the sum of the two tenants' performance metrics falls below both specified limits, we attribute this to the contention of microarchitecture resources. For example, assume there is no attacker, and the victim can achieve 100 Gbps. However, with an X Gbps attacker, the victim reduces to Y Gbps, and $X+Y<100$. Let us also assume the total request rate is below the RNIC specification. In this situation, we conclude that some microarchitecture resource is bottlenecked. The traffic is using RC connection unless otherwise noted.

We test four types of 100 Gbps RNICs: NVIDIA ConnectX-5 EN and ConnectX-6 Dx, Chelsio T62100-LP-CR, and Intel E810. NVIDIA NICs runs RoCE, and the Chelsio NIC runs iWARP. Intel E810 supports both RoCE and iWARP, but we currently only test its RoCE implementation. RoCE and iWARP are two standard ways to run RDMA over Ethernet-based networks. Our testbed consists of two servers, each equipped with an RNIC, and the two RNICs are connected via a 100 Gbps switch. For NVIDIA RNICs, we have access to their hardware counters, e.g., cache miss counters, through their network adapter management tool `NEO-Host` [44]. These hardware counters allow us to pinpoint which resource is oversubscribed. For example, when the ICM cache miss counter increases quickly with a certain application workload, we learn that this workload heavily uses this cache, making it oversubscribed. Since other RNICs do not expose such counters, we experiment other RNICs based on their end-to-end performance metrics (e.g., bandwidth).

Scenarios	Alone	Registration	Deregistration
BW / Gbps	96.6	95.9	**48.0**
Miss Rate	17.2%	22.9%	49.1%

Table 1: MR control verbs exhaust the MTT cache and reduce bandwidth.

3.2 NIC Caches

We are aware that an RNIC has at least three types of caches, as shown in Figure 3. The RNIC stores several types of metadata in these caches to accelerate the request processing, such as the QP contexts in the ICM cache. Prior works have identified some RNIC cache contention problems caused by data verbs with particular patterns. For example, transmitting small messages across many RC QPs simultaneously and random accesses to a large number of memory regions can cause certain types of severe cache misses (e.g., ICM and MTT/MPT) [29, 53]. ScaleRPC [5] found that this scalability problem can reduce the WRITE request rate by 90%.

In addition to these well-known problems, we observe a new, and even more severe way to exhaust caches:

Key finding #1: control verbs can cause excessive cache misses and a drastic performance reduction. Control verbs (e.g., `ibv_reg_mr`) are used to create and destroy objects like MRs and QPs, which will be used by data verbs to transfer data. To the best of our knowledge, there is no study on how control verbs consume RNIC microarchitecture resources. We find that control verbs can easily trigger excessive cache misses, thus degrading bandwidth and request rate.

We demonstrate this finding with a simple experiment on NVIDIA ConnectX-5 RNICs. We let the victim tenant use 6 cores, 16 connections per core, to issue 512B WRITE requests to exhaust the bandwidth capacity of the RNIC (i.e., 100 Gbps). Table 1 shows the results. The victim can achieve 96.6 Gbps with 17.2% MTT cache miss rate. The victim can still achieve line rate under such cache miss rate because QP multiplexing and the RNIC pipeline design can mask the overhead of cache misses to some degree. We let a single-threaded attacker keep registering memory regions (MRs) using `ibv_reg_mr` (∼5K registration per second) on the victim's sender side. In this scenario, the victim's bandwidth is almost not affected, staying at 95.9 Gbps with the miss rate slightly increased to 22.9%. However, if the attacker keeps deregistering MRs, we can see a significant impact on the victim: the cache miss rate increases to 49.1%, and the bandwidth degrades to 48 Gbps. The overhead under such a high cache miss rate becomes significant and can no longer be masked by the RNIC processing pipeline. It is worthwhile to note that the attacker does not need to issue any data verbs, so the attacker consumes no network bandwidth or request rate at all. Fortunately, we observe that such interference is negligible at the receiver side.

Compared with data verbs, we find that control verbs are easier to cause performance interference. To overfill cache resources, we need to launch enough in-flight data verbs and force them to randomly access a large number of objects (e.g., MRs). For example, on NVIDIA ConnectX-5 RNIC, we find that it takes 6 threads to access more than 18K MRs with 96 QPs to cause serious enough MTT cache misses that can degrade bandwidth by 40.1%. We believe cache misses due to data verbs will become less serious since RNIC vendors keep increasing on-chip cache resources. In contrast, control verbs impact cache resources by their special semantics instead of simply consuming them, and thus the impact from control verbs can be hard to mitigate. For example, we speculate that the MR deregistration may invalidate the entire MTT/MPT cache to avoid accessing outdated MRs. This causes cache misses for accessing other MRs.

We also conduct the same experiments on Chelsio and Intel NICs, and we observe similar results.

3.3 Processing Units

The RNIC has several processing units (PUs) to process verbs requests. Due to the lack of public available counters to monitor the status of PUs, we use the request rate as the metric to measure how PUs are consumed by different verbs. We summarize the following two key findings:

Key finding #2: performance interference between different data verbs depends on the complexity of verbs. Different data verbs have different complexities. Simple verbs, like `send` and `read`, only copy data between machines. Complex verbs, such as `fetch_and_add`, atomically add a 64-bit value to the memory of a remote address. This operation leverages PCIe features (e.g., read-modify-write transactions), and may also acquire a lock on the target address. These complex verbs consume more PU resources, resulting in a lower request rate [29]. Our new discovery here is that this difference in resource consumption can also open a new pathway for performance interference through resource exhaustion: a victim's performance can be substantially penalized when colocated with an attacker that uses complex verbs intensively.

To understand this effect, we first measure the data verbs request rate when competing with other data verbs. We begin with the NVIDIA 100 Gbps ConnectX-5 RNIC. We set up two workloads for each test, and each workload runs 8 QPs across 8 dedicated CPU cores to saturate the RNIC's rate. To avoid RNIC severe cache misses, we only use 128 QPs in total and 16 MRs. We observe less than 1% cache miss in all the PU tests. To avoid reaching the bandwidth capacity limit, we use 8B as the request size of all data verbs. We first set up one workload (victim) using a particular type of data verbs, and then set up the attacker workload with different types of data verbs. We show their request rate results in Figure 4.

Our first takeaway is that in addition to the ATOMIC operations [29], the READ operations are also more expensive than SEND/RECV and WRITE. When they are running alone (as victim traffic), FAA and CAS only achieve 5.2 Mrps and 4.8

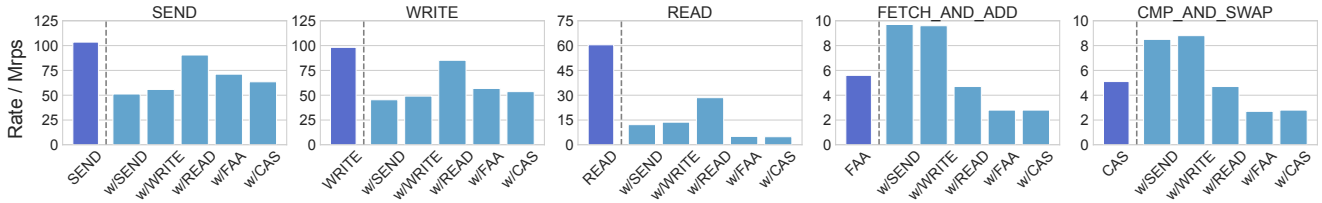


Figure 4: The contention of different data verbs on PU (NVIDIA). The leftmost bar on each subfigure is the request rate of running the victim only. The right 5 bars of each subfigure are the victim's rate when the attacker is running.

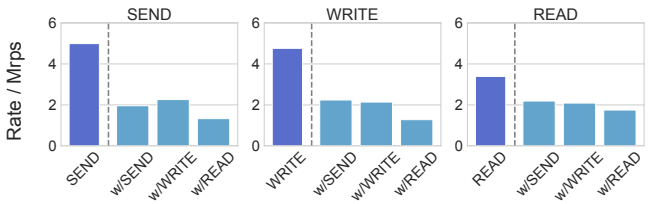


Figure 5: The contention of different data verbs on PU (Chelsio). The leftmost bar on each subfigure is the request rate of running the victim only. The right 3 bars of each subfigure are the victim's rate when the attacker is running.

Mrps respectively. READ achieves approximately 60 Mrps. SEND and WRITE can achieve more than 90 Mrps.

The second and the more important takeaway is that the contention behavior between different combinations of data verb operations can vary. For example, when the victim runs a READ workload alone, it can achieve 60 Mrps. If the attacker runs a CAS workload, the victim's request rate immediately drops to 3 Mrps. If the attacker runs a READ workload, the victim's request rate only drops to 30 Mrps. This means the complex verbs (e.g., CAS) can consume more resources and penalize other colocated verb workloads. One non-intuitive behavior we want to highlight is that the request rate of the victim running FAA or CAS can actually increase if the attacker runs a SEND or WRITE workload under this setting[1].

We also conduct similar tests on 100 Gbps Chelsio T62100-LP-CR RNIC, and the results are shown in Figure 5. This iWARP RNIC does not support ATOMIC operations. We observe that the iWARP RNIC's request rate for all types of data verbs is lower compared with RoCE RNICs, which matches findings from previous works [8, 49, 71]. We find that the contention among data verbs on Chelsio's RNIC also varies. For example, the victim with WRITE workload can achieve 4.76 Mrps without interference. The attacker can cause the victim's request rate to drop 55.0% with SEND workload and 73.1% with READ workload. The specific patterns are different from NVIDIA RNIC, but this result still demonstrates our key finding: the PU overhead of different data verbs varies.

Key finding #3: error handling can stall RNIC processing units and hang all the applications. RNICs need to handle a few types of errors, including transport timeout (the responder side does not send an ACK or NACK), Receive Not Ready (RNR) error (the responder does not have enough receive requests for arriving send requests), local or remote protection error (the posted request does not reference a valid local or remote memory region), and local operation error (an opcode is operated on the wrong type of QP). Handling these errors require resources from RNIC processing units and some errors can be expensive for RNICs to handle.

[1] We report this to the RNIC vendor and this observation is acknowledged. However, the root cause currently has not been figured out yet.

Scenario	Victim Bandwidth	SEND Bandwidth
Victim Only	97.07	-
w/o RNR	93.53	4.01
w/ RNR	**0.018**	**0**

Table 2: The impact of RNR errors on bandwidth. The unit is Gbps.

On NVIDIA ConnectX-5 and ConnectX-6 RNICs, we find handling RNR errors can *completely* stall the RNIC processing units. For the victim, we use Perftest [56] to keep 128 outstanding 64KB WRITE requests on a single QP to saturate the bandwidth capacity. For the attacker, we only use a single QP (i.e., the SEND application in the table) to keep only one in-flight 4KB SEND request to consume a small amount of bandwidth. As shown in Table 2, if the SEND application generates traffic normally (e.g., the responder posts enough receive requests), it consumes 4 Gbps bandwidth, and the bandwidth for the victim only drops approximately 3.5 Gbps. However, when the SEND application triggers RNR errors (e.g., the responder side does not post any receive requests), both the SEND application and the victim are *stalled*. We test this RNR errors with both directions and see the same results. The reason is that the RNIC of the RNR receiver is stalled, and the RNIC cannot even process the ACK packet. The victim therefore is stalled even when they are sending traffics in the opposite direction.

We conduct the same experiments using both Intel and Chelsio NICs. We observe that the victim's QP connections are also terminated unexpectedly during data transfer for Intel E810. Fortunately, we do not see such RNR issue for Chelsio T62100-LP-CR. Our best guess is that the iWARP is designed on the top of TCP and aimed at running on a lossy fabric, so it may have a more effective error handling mechanism.

3.4 PCIe Bandwidth

The RNIC is connected to the PCIe controller and transfers data from/to the CPU using PCIe lanes. The impact of PCIe

on the networking stacks has been studied by several prior works [29, 34, 52]. Based on existing PCIe models, we further study how RDMA verbs consume and even use up the PCIe bandwidth. Previous works have already identified how RDMA loopback traffic can exhaust PCIe bandwidth [26, 33]. We therefore focus on the normal RDMA TX and RX traffic. To transfer an RDMA message, PCIe introduces the following types of extra bytes: (1) an MMIO to ring the doorbell on the RNIC (64B, depending on cache line size), (2) a Work Queue Element (WQE) (36B or 64B), (3) the PCIe protocol overhead (e.g., TLP headers), and (4) extra PCIe operations triggered by cache misses. Our key observation for PCIe bandwidth is:

Key finding #4: PCIe bandwidth will only become the bottleneck when the request size is in a specific range. We only need a single tenant to demonstrate this key finding. We run the experiment on NVIDIA 100 Gbps ConnectX-5 RNIC. The PCIe bandwidth capacity is 128 Gbps (PCIe Gen 3.0 x16). We use 96 QPs across 6 cores to saturate the PCIe TX bandwidth. Each QP keeps 256 outstanding WRITE requests. We vary the request size and collect both the NIC and the PCIe bandwidth consumption by reading the RNIC's counters. The result is shown in Figure 6. We first observe that when the payload size is small, the commodity RNIC can mitigate the WQE overhead by embedding the small message in the WQE. As shown in the green rectangle, when the request size is smaller than 28B, increasing the request size does not cause more PCIe bandwidth consumption because the payload is embedded in the same MMIO operation with the WQE.

Our second observation is that PCIe TX bandwidth may only become the bottleneck when the payload size of the request is in a specific range. The reason is that short requests are first throttled by the request rate before exhausting PCIe bandwidth while large requests are always throttled by the RNIC's bandwidth capacity. We confirm this observation through a theoretical PCIe consumption model and we present two concrete examples. We assume the network MTU is 4096B and the maximum payload per PCIe transaction is 128B (the worst setting to maximize the PCIe overhead). The TLP overhead depends on the implementation [52] and we assume it as 20B, a typical size for a PCIe 3.0 device. Transmitting a 29-byte message will consume at most 127 network bytes and at least 189 PCIe bytes [29, 69]. Therefore, to saturate the link bandwidth (100 Gbps), we need at least 148.8 Gbps PCIe bandwidth, which is much larger than the PCIe 3.0x16 capacity. Appendix A includes the detailed computation. Our measurement shows that the actual consumption can be even higher, as shown in Figure 6. The consumption model for PCIe RX bandwidth (i.e., the RNIC to the host) is similar to that of TX. Additionally, too many cache misses may also cause high PCIe bandwidth consumption due to lots of PCIe reads to fetch metadata. However, in most scenarios, the large number of cache misses will first slow down the RNIC execution (e.g., introduce extra latency) and the PCIe bandwidth is therefore less consumed. In our measurement

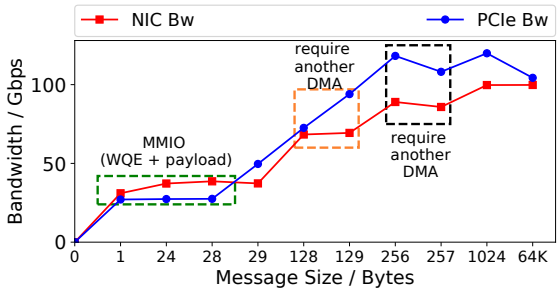


Figure 6: The PCIe bandwidth and RNIC bandwidth consumed by the application.

of cache misses, we do not observe cases where PCIe TX bandwidth is exhausted.

Both the theoretical model and our experimental results demonstrate that the PCIe bandwidth can become the bottleneck, but only for a particular request size range.

3.5 Other findings

We also have several other interesting findings. In the interest of space, we only briefly present them here. However, we do use these findings to guide our test suite design in §4.

Other finding #1: Data verbs contend for different RNIC caches. We conduct the scalability test using different data verbs, and observe different types of cache contention. For example, a large number of RC QPs that issue READ and WRITE will mainly cause ICM cache misses. A large number of UD QPs that issue SEND/RECV requests or many RC QPs that issue ATOMIC requests can cause severe RECV WQE cache misses. This observation indicates that data verbs contend for cache differently, similar to the contention on RNIC PUs.

Other finding #2: Wide range access across many objects (QP, CQ, MR) causes ICM cache misses. The scalability issue has been well studied, but our measurement reveals new observations. In addition to QP and MR, the context of the completion queue (CQ) is also stored in the ICM cache. Thus, accessing a large number of CQs can also trigger severe ICM cache misses. In addition, allocating a large number of these objects does not necessarily cause severe ICM cache misses. Wide range access across the objects (i.e., poor locality) is the key to triggering severe ICM cache misses and performance degradation.

Other finding #3: The impact of control verbs is restricted by its kernel involvement. We observe that all control verbs are first processed by the kernel drivers, thus causing expensive context switch. The execution rates of these control verbs are usually throttled by the kernel instead of RNIC processing. Therefore, control verbs have a limited impact on exhausting RNIC PUs. However, they can still cause significant performance interference and affect the other applications by triggering severe cache misses, as our key finding #1 shows.

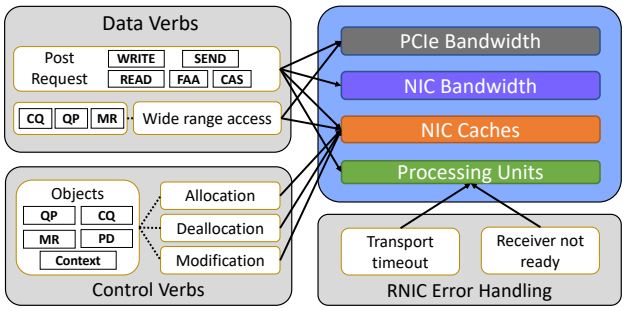


Figure 7: The relationship between verbs and microarchitecture resources. The arrow indicates heavy resource consumption.

3.6 The Resource Consumption Model

We summarize our findings in an RDMA operation model shown in Figure 7. This model describes which microarchitecture resource a verb operation consumes *heavily*. Note that a verb operation can also use other microarchitecture resources that are not captured by our experiments. This is because the usages of these resources are low and do not lead to resource contention. This model is *qualitative*: we do not try to understand the exact resource usage since we have no visibility into proprietary RNIC hardware. For example, we know a certain traffic pattern can trigger a certain type of cache misses, but we does not figure out the total size of the cache or how much of the cache an operation consumes. Even so, we show that this model is sufficiently powerful for us to create the first test suite for RNIC performance isolation, and it can capture a wide range of workloads that can break existing performance isolation solutions.

4 The Husky Test Suite

After we understand how different RDMA operations use these microarchitecture resources, we can design a test suite to evaluate performance isolation solutions. Our goal is the following: given an RNIC hardware and a performance isolation solution, we want to find a set of workloads combinations for an attacker and a victim that can break the performance isolation. We need to check different victim workloads for completeness because different victim workloads are sensitive to exhaustion of different microarchitecture resources.

Our test suite must be general: we will use it to test various RNIC performance isolation solutions on different RNICs. This means we cannot rely on tools and features from specific vendors, such as Mellanox Neo-Host [44]. In addition, different RNICs have different amounts of microarchitecture resources. And existing performance isolation solutions may only be able to mitigate contention on specific resources.

To this end, we build Husky to systematically test and evaluate RNIC performance isolation solutions. Husky targets at four types of resources: NIC bandwidth, PCIe bandwidth, NIC PU, and NIC cache. For each type of resource, we design synthetic workloads with different types of behaviors (e.g., control verbs) to exhaust this resource. More specifically, we exhaust NIC BW with long messages using different opcodes (e.g., WRITE); we exhaust PCIe bandwidth with loopback traffic and specific message patterns (from key finding #4); we exhaust NIC PU with expensive data verbs (key finding #2), small messages, or error handling behaviors (key finding #3); we exhaust different types of RNIC cache with intensive control verbs (key finding #1) and a wide range access of data verbs. We vary parameters (e.g., connection types) of some synthetic workloads to be more inclusive. In all, Husky includes 52 attacker synthetic workloads (6 for NIC BW, 4 for PCIe BW, 14 for NIC PU, and 28 for NIC cache) and 20 synthetic victim workloads. Many of the attacker workloads cannot be directly generated with existing RDMA traffic engines. We therefore extend Collie [33]'s traffic engine, the most flexible one to the best of our knowledge, to generate these synthetic RDMA traffics, including flexible control verbs workloads and error handling workloads.

Husky's framework can also easily allow running real applications as additional victim workloads. Husky currently contains two real applications, including the OSU benchmark [54] and eRPC-based Masstree key-value store [27, 40]. The OSU benchmark contains workloads such as allreduce and allgather. Note that we can integrate any RDMA applications into Husky. We test all the (victim, attacker) workload pair exhaustively from our test suite.

One key question is how to define a violation of performance isolation. Our definition of violation depends on the concrete isolation solution. Husky uses a user-specified predicate to compute the expected performance results when isolation is enabled. Husky compares the actual performance with the expected performance to identify violation. For example, most of existing performance isolation solutions only provide bandwidth guarantee. The expected performance for these isolation solutions therefore is a guaranteed bandwidth, B_g. We assume the application can consume bandwidth of B_a when running alone. The bandwidth of this application should be at least $(1-\alpha)\min(B_a, B_g)$ under *any* attacker workload, where α is a tolerance level. A lower α means stricter isolation. We use an example to demonstrate how this definition works: let us assume that attacker and the victim are configured to share the same 100 Gbps network and we set α to be 25%. If the victim can achieve 60 Gbps when running alone, it should be able to achieve at least $(1-25\%)\min(60, 50) = 37.5$ Gbps under the attacker's workload. If the victim can only achieve 10 Gbps when running alone, its consumed bandwidth should not be less than $(1-25\%)\min(10, 50) = 7.5$ Gbps. In practice, we find all existing performance isolation solutions for commodity RNICs are bandwidth guarantee or can be translated into bandwidth guarantee. We use this definition for performance isolation violation in §5 and set α to be 25%.

5 Evaluation

We use a NVIDIA testbed to evaluate existing RDMA performance isolation solutions. There are two servers in the testbed,

Resource	Processing Units			RNIC Cache		PCIe BW
Isolation Mechanism	Error Handling (RC)	Error Handling (UD & UC)	Data Verbs	Control Verbs	Data Verbs	Data Verbs
SR-IOV	✓	✗	✗	✗	✗	✗
HW TC	✗	✗	✗	✗	✗	✗
SR-IOV + HW TC	✓	✗	✗	✗	✗	✗
Justitia	✗	✗	✓	✗	✗	✗
Justitia + HW TC	✗	✗	✓	✗	✗	✗

Table 3: Performance isolation violation caused by exhausting microarchitecture resource. Justitia can only provide isolation among applications using the same function, so cannot be combined with SR-IOV. ✓ means performance isolation is properly enforced. ✗ means Husky can find a workload pair (attacker, victim) to violate performance isolation by exhausting microarchitecture resources.

and each is equipped with one 100 Gbps NVIDIA ConnectX-5 RNIC. The server is equipped with Intel Xeon Gold 5215 CPUs, and the RNICs are connected to the server through PCIe 3.0 x16. The RNICs are connected to a 100 Gbps NVIDIA switch. We use Ubuntu 20.04 and the kernel version is 5.11. For NVIDIA NICs, the kernel drivers and verbs libraries are both from 5.4-OFED. The firmware version is 16.31.1014. We also conduct all the experiments also on NVIDIA ConnectX-6 RNICs and the result is similar.

We evaluate 3 different isolation solutions provided by RNIC vendors and prior work: (1) *NVIDIA separate hardware traffic class (HW TC)*. Cloud operators can set separate TCs for different tenants to use, which separate the RNIC bandwidth and packet buffers [47] to enforce performance isolation. Modern RNICs typically only have 8 traffic classes. This means we cannot use HW TC when we want to colocate more than 8 tenants in a physical server. (2) *NVIDIA SR-IOV*. Though the SR-IOV technique is designed for hardware virtualization, it provides separate virtual functions with some separated resources to different tenants and actually achieves some degrees of performance isolation [45]. *(3) Justitia, a software-based performance isolation solution* [71]. Justitia implements data verbs rate-limiting and pacing in RDMA userspace libraries to enforce performance isolation. This means Justitia has no security: malicious applications can easily circumvent the userspace library. Although Justitia's software architecture does not target a multi-tenant public cloud environment, we still use Husky to evaluate the effect of its isolation policy (e.g., its token-based algorithm). We also evaluate all the possible combinations of the above solutions[2]. Unfortunately, though we have a testbed with Chelsio T62100-LP-CR and Intel E810 NICs, we did not enable their hardware-based isolation mechanisms. Justitia also does not support Chelsio or Intel drivers. We therefore are not able to conduct the same evaluation on Chelsio or Intel NICs. [3]

[2]We do not test Justitia with SR-IOV because Justitia only isolates traffic through the same device. When SR-IOV is enabled, tenants are using different devices (i.e., VF) and Justitia does not work for that scenario.

[3]We contact the NIC vendors and have multiple rounds of conversations with their experts. However, we still fail to enable any hardware isolation solution for RDMA on both NICs. In addition, we are not aware of any prior work that can set up such RDMA isolation.

5.1 Testing Existing Performance Isolation Solutions

Based on the types of verbs and the exhausted resources, we categorize the workloads generated by Husky into 6 groups. We distinguish the error handling of RC from UD & UC because they cause different behaviors of RNIC PU, and we observe some isolation solution (e.g., SR-IOV) provides different degrees of isolation on these PU behaviors.

We first take a look at the hardware-based isolation mechanism provided by NVIDIA. For *NVIDIA SR-IOV*, we enable two virtual functions (VF) and assign both the victim tenant and the attacker tenant with one VF. We also enable the VF-based rate limiter and restrict the maximal TX bandwidth of each tenant to be 50 Gbps, which is a typical fair sharing setting for the public multi-tenant environment. Given this configuration, we therefore define the isolation violation for *NVIDIA SR-IOV* as the victim's consumed bandwidth (in terms of bits per second) being reduced by the attacker to less than $(1-\alpha)min(50, B_a)$, where α is 25% and B_a is the victim's bandwidth without attack. For *NVIDIA HW TC*, we assign each tenant with a dedicated TC. For example, the victim exclusively uses TC 0 and the attacker exclusively uses TC 3. We configure TC 0 and TC 3 to equally share the RNIC bandwidth and the NIC buffer (which stores the packets, different from the cache). The violation definition for *NVIDIA HW TC* therefore is the same as that of *NVIDIA SR-IOV*.

The first three rows of Table 3 show the isolation effect provided by SR-IOV, HW TC, and the combination of them. Unfortunately, we find both SR-IOV and HW TC fail to provide enough isolation on RNIC's microarchitecture resources. For example, by exhausting RNIC's cache through either control verbs or data verbs, Husky can successfully affect the colocated victim's applications, even when both SR-IOV and HW TC are enabled. The key reason is that both SR-IOV and HW TC only isolate the architectural resources (e.g., link bandwidth) and do not restrict the cache usage of a single tenant. Husky therefore is able to use an attacker workload that exhausts RNIC cache, such as MTT/MPT cache. Other applications would suffer from severe cache miss and hence the performance drop. In addition, we find that although SR-IOV is mainly aimed at virtualization, it has indeed enforced some isolation, especially for RNIC PUs. The RC RNR error

handling can cause RNIC PUs to pause and even hang the colocated applications if there is no performance isolation mechanism enabled. With SR-IOV, the RC RNR error does not affect tenants running on other VFs. However, the similar RNR exception handling process for UD and UC still violates the isolation of SR-IOV. Due to the RNIC's black box nature, we do not know the root cause of such a difference. Our best guess is that some part of the RNIC's PUs (e.g., that handles RC RNR) is isolated by different VFs, while other parts are not well isolated. These hardware-based solutions also cannot isolate PCIe bandwidth well. We observe that an attacker can consume substantial PCIe bandwidth and reduce the victim's usable bandwidth.

We then evaluate the software-based solution, Justitia. Justitia is not designed for the public cloud and requires the tenant to cooperate (e.g., using modified RDMA libraries). Husky can certainly break its isolation by bypassing the modified libraries, but this would defeat the purpose of testing Justitia. We therefore require all of Husky's traffics (both the victim and the attacker) to go through Justitia's modified drivers and be paced by Justitia. In addition, Justitia only supports limited types of data verbs on the latest drivers (i.e., mlx5), so we restrict the applications to only use the opcodes that Justitia supports. Justitia aims at providing each tenant a fair share of the NIC resource. We only set up two tenants, so we simply define the violation of Justitia as the victim's bandwidth is less than $(1-\alpha)min(B_a, 50)$, similar to the definition for SR-IOV. We also test the combination of Justitia and HW TC.

As shown in Table 3, Justitia does provide some PU isolation but to a limited extent. For example, Justitia takes the RNIC's request rate (i.e., execution throughput) into its isolation consideration. It therefore uses a pacer to control the request rate for each tenant and successfully prevent a single tenant from posting a large number of requests to exhaust the PUs. However, its isolation is violated when the attacker keeps posting requests that trigger error handling on the RNIC. The reason is that these errors are detected and handled by RNIC, which is out of Justitia's control. In addition, Justitia does not take cache and PCIe into consideration. The attacker tenant therefore can still exhaust the RNIC cache and PCIe bandwidth and cause other tenants to suffer from excessive cache misses or low usable PCIe bandwidth.

It is worthwhile to note that these solutions already provide more or less tolerable isolation for architectural resources, e.g., NIC bandwidth. Husky includes a set of workloads that only contend for NIC bandwidth, and we do not see such violation on those workloads when enabling these solutions. However, ignoring microarchitecture resources makes these solutions insufficient for real public cloud deployment.

5.2 Impact for Real Applications

Next, we conduct experiments on a larger testbed to study how microarchitecture resource exhaustion impacts real application workloads when using state-of-the-art performance isolation solutions. We use the allreduce workload [54] on an RDMA-based MPI implementation [55] and eRPC-based Masstree (a key-value store) [27, 40] as two real victim applications. Our testbed consists of four physical servers. Each server is equipped with one 100 Gbps NVIDIA ConnectX-5 RNIC. The other settings are the same as §5.1. The victim applications run their VMs on all the four servers. The attacker tenant controls two VMs, each on a different server. We set up the testbed this way to emulate a real multi-tenant environment because an attacker may not have VMs colocated with all the victim's VMs. However, our results demonstrate that violation of performance isolation in a subset of the victim's VMs is already enough to substantially reduce the overall end-to-end performance of the real distributed applications.

For protection mechanisms, we enable either SR-IOV + HW TC or Justitia + HW TC to provide isolation for the collective communication application. For eRPC-based Masstree, we only enable SR-IOV + HW TC. This is because Justitia only supports high-performance RDMA WRITE on the latest NVIDIA drivers, but eRPC-based Masstree leverages UD SEND/RECV for its communication.

We use four types of attackers from the Husky test suite to demonstrate our results: (1) **BW attack** is the baseline. We use the standard Perftest [56] `ib_write_bw` to set up a bandwidth-hungry application. It uses 16 RC QPs and each QP keeps 128 outstanding 1 MB WRITE requests to saturate the link bandwidth (consuming ∼50 Gbps when rate limiter is enabled). BW attack does not target any microarchitecture resources. (2) **PCIe attack** exhausts PCIe TX bandwidth. It runs 36 RC QPs on 6 cores and keeps 128 outstanding 257 B WRITE requests. It also consumes almost 50 Gbps link bandwidth (less than 20 Mrps) but causes more than 73 Gbps PCIe TX bandwidth consumption. This leaves only about 50 Gbps usable PCIe TX bandwidth (i.e., less than 50 Gbps usable network bandwidth) for the victim. (3) **Cache attack** exhausts RNIC cache. It runs 1536 RC QPs on 6 cores, uses 12288 MRs and each QP keeps only a single 256 B outstanding request. This attacker causes severe cache miss and only uses less than 7 Gbps link bandwidth (i.e., 3 Mrps). (4) **PU attack** pauses RNIC PUs. It runs 1 UC QP on a single core and keeps 128 outstanding SEND/RECV requests. Its receiver side does not post any receive requests, so the RNIC has to handle many receive not ready exceptions. It consumes less than 0.5 Gbps and less than 0.5 Mrps.

We begin with testing the RDMA-based allreduce workload. Allreduce is a collective communication operation widely used in distributed deep learning training. It aggregates a vector across all workers and propagates the result back to all workers. We set up 2 workers on each host (8 in total) to run allreduce. The allreduce buffer size is set to 1 MB. We run allreduce continuously and record the execution rate (allreduce operations per second). The raw rate without any isolation mechanism and interference is shown as the leftmost bars in the figure. The bar of no attack indicates the effect of

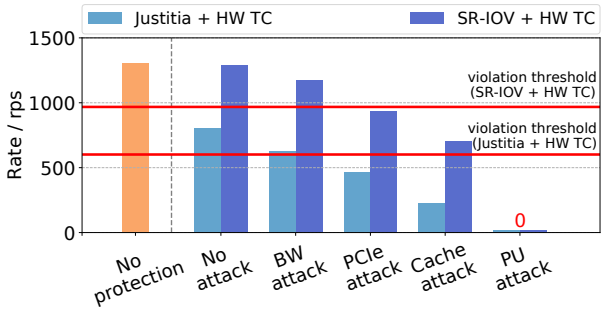


Figure 8: Allreduce results under exhaustion of different resources.

enabling these isolation solutions. When Justitia is enabled, the allreduce rate drops by 38.5%. One possible reason is that Justitia uses a shim layer (the pacer) to exert sender admission control, which introduces extra performance overheads compared to the hardware-based solutions. Since the allreduce workload only uses less than half of the NIC bandwidth (23 Gbps), its performance under attack should be at least $(1-\alpha)P_a$, where α is 25% and P_a is its performance without any attack. We can then compute a violation threshold in allreduce rate for each isolation solution based on the bandwidth the victim should consume.

The result for allreduce is shown in Figure 8. The horizontal red lines show the violation threshold. Bars under the red line indicate isolation violation. P_a for the application with Justitia + HW TC is 38.5% lower than that with SR-IOV + HW TC. This means the violation threshold is also 38.5% lower for Justitia + HW TC. We first observe that the BW attack only causes a negligible performance drop for SR-IOV + HW TC setting. And Justitia + HW TC also achieves the bandwidth isolation goal within the tolerance. We then observe that all the PCIe, Cache, and PU attacks successfully violate the isolation provided by either Justitia + HW TC or SR-IOV + HW TC. For example, the PCIe attack can cause the performance of the allreduce application to drop 27.3% for SR-IOV + HW TC and 42.1% for Justitia + HW TC. The impact of the Cache attack is more significant. Allreduce workload's performance drops more than half (71.3%) for Justitia + HW TC and almost half for SR-IOV + HW TC. We observe that the PU attack is the most powerful. It can directly stall the allreduce application by exhausting the RNIC PUs.

We use the same set of attackers to test the eRPC-based Masstree. We use the default setting of eRPC-based Masstree (e.g., key size and the number of threads). We set up the key-value server in one physical server and three clients each in a different physical server. We colocate one attacker VM with the key-value server and another attacker VM with one of the clients. We collect the execution rate (in terms of the number of GET requests per second) and the latency from all the clients. The Masstree server only uses 14 Mrps and less than 20 Gbps, so we define the isolation violation as the same as the violation of allreduce. Figure 9 and Figure 10 show the GET rates and the latency results. The SR-IOV + HW TC

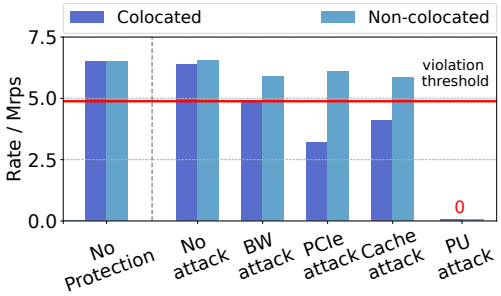


Figure 9: Mastree's GET rate under exhaustion of different resources. colocated means that the client and the attacker are on the same host. Non-colocated means that they are on different hosts.

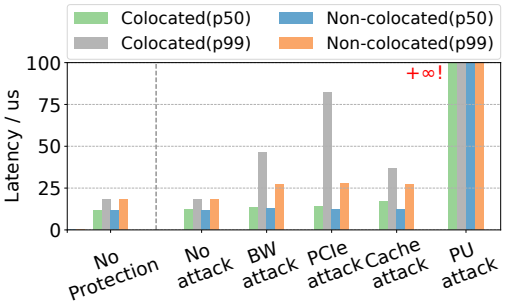


Figure 10: Mastree's latency under exhaustion of different resources.

more or less achieves the BW isolation goal within tolerance. We find that all microarchitecture resource exhaustion attacks successfully violate the isolation for the client that is colocated with an attacker VM. Similar to the allreduce workload, the PU attacker stalls the entire key-value store system. Worse still, it even pauses the clients that are not colocated with an attacker VM. This is because we stall the key-value server.

Another observation is that the performance of eRPC-based Masstree is impaired by the cache exhaustion attack but to a very limited extent. One possible reason is that the eRPC leverages UD transport. A UD QP does not need as much connection metadata as an RC QP does and therefore is less sensitive to the RNIC internal cache miss. In addition, we find that the Masstree is more sensitive to PCIe exhaustion. This is probably due to its small request size. According to our key finding #4, requests of a relatively small size cause more extra PCIe TX bandwidth consumption.

We have several high-level takeaways from the real application results.

Takeaway #1: targeting microarchitecture resources makes violating performance isolation easy. If we treat the RNIC as a black box, it is quite difficult to break performance isolation. The BW attack targets the bandwidth resource, and we observe that all the existing solutions provide good protection. However, once we know a few more details about how an RNIC works (e.g., the potential microarchitecture resources), breaking isolation becomes simple. Our attack is very efficient. For example, Cache Attack only needs 7 Gbps and 3 Mrps. PU Attack stalls victims with even less bandwidth

and request rate. Note that these attacks are only targeting publicly disclosed microarchitecture components.

Takeaway #2: applications' sensitivity for resource contention is different. Applications' end-to-end performance drops can be quite different even for the same attack. The allreduce application is more sensitive to the cache exhaustion while the Masstree is more vulnerable to the PCIe exhaustion.

Takeaway #3: distributed applications need performance isolation on every single server. For both applications, the attacker only has two VMs, but why does the application-level performance drop substantially even if the application is running across four machines? Many modern distributed systems' performance is usually bottlenecked by a few slowest workers in the system. For example, in allreduce, each iteration requires synchronization of all workers. Thus, our attack on one or two workers can slow down the entire allreduce procedure.

5.3 Analysis for Existing Solutions

Our evaluation shows that all existing approaches fail to provide RDMA performance isolation.We now analyze the fundamental restrictions of these solutions and some potential improvements we may achieve.

SR-IOV and separate HW TC. These hardware based solutions already provide some hardware resource isolation (e.g., the hardware queue and the on-NIC packet buffer). Theoretically, RNIC vendors should be able to incorporate more hardware isolation features to these solutions. For example, to statically separate NIC PU or partition NIC cache for different VFs can help to build a better isolation mechanism for SR-IOV. However, these hardware modifications are non-trivial and can hardly be applied to existing hardware. RNIC vendors usually release these new features together with their new hardware products. Cloud providers thus cannot use these features in existing hardware.

Justitia. Justitia provides modified userspace libraries and uses sender admission control to enforce fair sharing of both bandwidth and execution rates for all tenants. Justitia does not work for a multi-tenant cloud because its policies are not enforceable: a malicious application can easily circumvent the modified user libraries. Putting the security aspect aside, it is worthwhile to ask whether a pure software solution like Justitia could in theory support RDMA performance isolation. We do not have a direct answer to this question, and we believe it is an interesting future research direction. We reckon that this can be quite difficult for the following reasons. First, it is challenging to track and control how much cache a tenant has occupied without hardware support. Second, it is challenging to establish a quantitative resource consumption model for verbs. Finally, error handling is deeply integrated into RDMA NIC hardware, and is opaque to software.

6 Guidelines

Our results show that, unfortunately, no existing RNIC performance isolation solution is sufficient. We analyze the failure of existing isolation solutions based on our key findings, and we present several design guidelines for potential future RDMA performance isolation work. These guidelines may also be helpful for RDMA application developers to write better RDMA applications under multi-tenant environments.

Hardware support for isolation is needed. Software approaches like Justitia [71] have a common problem. They only monitor architecture-level metrics, e.g., latency, bandwidth, and request rate. They cannot detect contention in microarchitecture resources, e.g., caches, let alone manage and fair share those resources. We believe future performance isolation solutions will have to leverage hardware support, similar to how modern hypervisors can use Intel Resource Director Technology (RDT) to monitor and manage access to the last-level cache and memory. NVIDIA RNICs expose several useful hardware counters, but they are still insufficient. For example, we can only observe cache misses, but we cannot manage the cache access or split the cache for different tenants.

A layer of indirection is needed. RDMA means kernel bypass for data verbs. This enables low latency and reduced CPU overheads. So where should performance isolation be enforced? We believe that future performance isolation solutions will require a layer of indirection either in NIC or in software. Having the enforcement point in the userland RDMA library (as Justitia) does not work, because it lacks security. Instead, a software indirection can have a microkernel-like design, with a set of cores running the isolation logic in a separate protection domain [43]. RDMA performance isolation should be enforced in such a central controller that takes over both control verbs and data verbs.

Programmer, compiler, and library support for RDMA applications. After a future performance isolation solution is invented, applications may need modification as well. If the future performance isolation solution requires strict partitioning of microarchitecture resources, this means each application has limited microarchitecture resources to use and can lead to substantially reduced performance. The amount of microarchitecture resource an application uses may also vary (depending on how many other tenants are on the same server or other configurations). Building high-performance RDMA applications will require additional effort for the programmer, compiler, and application library to efficiently use these limited resources. For CPU cache, these efforts occurred in the research community two decades ago [35, 36, 42].

7 Discussion

The impact of broken RDMA performance isolation. Our evaluation shows that a malicious tenant can cause other tenants' to suffer from drastic performance drop or even get stuck. In addition, a broken performance isolation exposes vulnerability for malicious users to conduct side-channel attacks. Since the tenant can affect others' performance on the same host, it can set up side channels that leak access pat-

terns of victim nodes or deliver information by affecting the host's performance in a pattern [65]. RDMA performance isolation therefore is a critical feature for a secured RDMA public cloud.

What RDMA performance isolation solution should cloud providers use today? One good news is that we are not aware of any cloud provider that currently using commodity RNICs to provide RDMA-capable VMs with partitioned host resources. To rent an RDMA-capable VM, customers have to rent the entire physical machine. This means currently we do not need an RNIC performance isolation solution at all, because the RNIC only runs a single tenant's traffic. To move forward to multi-tenant usage of an RNIC, we believe performance isolation is still a major blocker, and multi-tenancy should not be enabled until a mature performance isolation solution is ready, one that can at least pass our test suite.

Generalizability to other kernel bypass host networking architectures. Our test suite design is based on the *verbs* interface, which is RDMA-specific. However, we believe our methodology should be generalizable to find violations of performance isolation in other kernel bypass architectures, e.g., DPDK [13], 1RMA [64], as these implementations commonly require RDMA-like mechanisms in the DMA portion of the design. The industry trend today is to offload functions to hardware accelerators. For example, RDMA is offloading congestion control and reliable message delivery into the hardware. Microarchitecture resources in hardware are critical to delivering these offloaded functions. Paying attention to these microarchitecture resources for performance isolation is going to be increasingly important.

8 Related Work

Microarchitecture resources in RNICs. The existence of RNIC microarchitecture resources is well-known in the networking community, and many studies focus on how to design RDMA applications to circumvent certain RNIC performance anomalies due to these resources. For example, HERD [28], FaSST [30], and eRPC [27] avoid using RDMA reliable connection to mitigate the QP context cache miss for better scalability. ScaleRPC [5] and Flock [50] multiplex reliable connections in a time-sharing manner to mitigate the scalability problem. Kalia et al. [29] studies the RNIC's PCIe behaviors and provides guidelines for writing efficient RDMA programs. Unfortunately, these works only focus on optimizing applications to fully utilize the limited resources in RNICs. However, public cloud providers cannot control the third-party tenants' applications. Collie [33] conducts a systematic search on RDMA performance anomalies, and the anomalies are mostly due to oversubscribed microarchitecture resources. However, since Collie only focuses on first-party traffic, it just builds a search space based on normal operations. It therefore only considers normal data verbs and fails to uncover findings related to other types of behaviors. For example, the key findings #1, #2, and #3 in §3 are fundamentally not covered by Collie's search space because Collie does not take control verbs, error handling, and expensive atomic verbs into consideration. In all, prior works focus more from the perspective of application developers. Our work is on a complementary aspect by looking from the public cloud provider's perspective: how these microarchitecture resources affect performance isolation. This requires us to be microarchitecture resource aware and take a look at all types of RDMA behaviors, including control verbs and error handling, because we need to deal with misbehaving and even malicious tenants.

Other NIC performance isolation solutions. PicNIC [34] provides isolation for both packet processing and bandwidth on NIC. This allows latency-bound workloads not to be affected by bandwidth-bound workloads. FairNIC [15] isolates resources in SoC-based SmartNICs. Compared with them, our work focuses on the RDMA-related resources on NICs.

Performance isolation in other contexts. Performance isolation problems are not limited to NICs. Other server hardware components also have this issue, and they already have corresponding solutions. There exist several partitioning techniques for CPU caches [11, 20] and memory bandwidth [22]. Network bandwidth in the network fabric is also a crucial resource to isolate [1, 3, 4, 6, 16, 24, 25, 37, 58–60, 62, 68] as well as the switch processing piplines [67].

9 Conclusion

RDMA is a promising networking technology to enable low latency and high CPU efficiency in datacenter networks. To enable RDMA in a multi-tenant environment, performance isolation is an important property, and RDMA NICs (RNICs) bring new challenges due to the existence of microarchitecture resources (e.g., RNIC cache, processing units). We present an RNIC operation model on how these resources are used by different RDMA operations. Using this model, we create Husky, the first test suite to evaluate RNIC performance isolation solutions. Our results show that none of the existing RNIC performance isolation solutions provides sufficient isolation against workloads that try to exhaust these microarchitecture resources. Our findings are acknowledged and reproduced by one of the largest RDMA NIC vendors. We believe that building a usable RNIC performance isolation solution will be a long battle.

Acknowledgement

We thank Chelsio, and Intel for their technical support. We especially thank NVIDIA, who gives us timely and insightful feedback, including the root causes of our findings and the corresponding solutions. We thank our shepherd Brent Stephens and other anonymous reviewers for their insightful feedback. Our work is partially supported by gifts from Adobe, Amazon, Meta, and IBM.

References

[1] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-End Performance Isolation through Virtual Datacenters. In *OSDI*, 2014.

[2] Infiniband Trade Association. Rocev2, 2014.

[3] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[4] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.

[5] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *EuroSys*, 2019.

[6] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG:Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*, 2016.

[7] The Global Cloud Computing Market Size. https://www.yahoo.com/now/global-cloud-computing-market-size-081600295.html, 2021.

[8] Chelsio Communications. 100g network performance for illumos, 2018.

[9] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[10] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP*, 2015.

[11] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *HPCA*, 2018.

[12] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 25–38. IEEE, 2007.

[13] Linux Foundation. Data plane development kit (DPDK). http://www.dpdk.org, 2015.

[14] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *NSDI 21*, 2021.

[15] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *SIGCOMM*, 2020.

[16] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[17] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *SIGCOMM*, 2016.

[18] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 342–362. Springer, 2006.

[19] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *SIGCOMM*, 2020.

[20] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From Concept to Reality in the Intel Xeon Processor E5-2600 v3 Product Family. In *HPCA*, 2016.

[21] Jeff Hilland. RDMA Protocol Verbs Specification. Technical report, Internet Engineering Task Force, 2003.

[22] Derek R. Hower, Harold W. Cain, and Carl A. Waldspurger. PABST: Proportionally Allocated Bandwidth at the Source and Target. In *HPCA*, 2017.

[23] IEEE. 802.3-2018 - ieee standard for ethernet. https://ieeexplore.ieee.org/document/8457469.

[24] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*, 2015.

[25] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[26] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*, 2016.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, 2016.

[31] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*, 2019.

[32] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*, 2019.

[33] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *NSDI*, 2022.

[34] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable Virtualized NIC. In *SIGCOMM*, 2019.

[35] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *ASPLOS IV*, 1991.

[36] A.R. Lebeck and D.A. Wood. Cache Profiling and the SPEC Benchmarks: a Case Study. *Computer*, 27(10):15–26, 1994.

[37] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-Driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.

[38] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *SIGCOMM*, 2019.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[40] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, 2012.

[41] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores. In *HPCA*, 2019.

[42] M. Martonosi, A. Gupta, and T.E. Anderson. Tuning Memory Performance of Sequential and Parallel Programs. *Computer*, 28(4):32–40, 1995.

[43] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019.

[44] Mellanox. Mellanox neo-host network adapter management software. https://support.mellanox.com/s/productdetails/a2v50000000N2OlAAK/mellanox-neohost.

[45] Mellanox Single Root IO Virtualization (SR-IOV). https://docs.nvidia.com/networking/pages/viewpage.action?pageId=12013542.

[46] Mellanox. Proprietary mellanox adapter diagnostics counters. https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=12005244.

[47] Mellanox Quality of Service (QoS). https://docs.mellanox.com/pages/viewpage.action?pageId=19811934, 2018.

[48] Mellanox Adapters Programmer's Reference Manual. https://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf, 2021.

[49] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *SIGCOMM*, 2018.

[50] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *SOSP*, 2021.

[51] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models, 2021.

[52] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *SIGCOMM*, 2018.

[53] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A Fast Transactional Dataplane for Remote Data Structures. In *SYSTOR*, 2019.

[54] OSU benchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/, 2021.

[55] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. The MVAPICH project: Transforming Research into High-Performance MPI Library for HPC Community. *Journal of Computational Science*, 2021.

[56] OFED perftest. https://github.com/linux-rdma/perftest, 2021.

[57] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces. In *VEE*, 2015.

[58] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[59] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. Elasticswitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.

[60] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[61] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA Is Turing Complete, We Just Did Not Know It Yet! In *NSDI*, 2022.

[62] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *NSDI*, 2011.

[63] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI*, 2012.

[64] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *SIGCOMM*, 2020.

[65] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: Remote oracles for the masses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 693–710, Santa Clara, CA, August 2019. USENIX Association.

[66] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *ASPLOS VIII*, 1998.

[67] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation mechanisms for High-Speed Packet-Processing pipelines. In *NSDI*, 2022.

[68] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.

[69] Understanding Performance of PCI Express Systems. https://docs.xilinx.com/v/u/en-US/wp350, 2018.

[70] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *EuroSys*, 2013.

[71] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI*, 2022.

A Network v.s. PCIe

To transmit a payload through Ethernet-based IP-routed RDMA network (i.e., RoCEv2), the network protocol introduces the following overhead.

1. **Ethernet overhead**. Each Ethernet frame includes 14-byte Ethernet (exclude VLAN) header and 4-bytes CRC as L2 overhead. In addition, each Ethernet frame has L1 overhead - each frame is preceded by a 7-byte preamble and 1-byte start-of-frame delimiter. The frame is also followed by an inter-frame gap. The gap should be at least 12-byte. The total Ethernet overhead per frame therefore is 38-byte [23].

2. **IP overhead**. IP overhead comes from the IP header, with a least size 20-byte.

3. **UDP overhead**. UDP overhead comes from the 8-byte UDP header.

4. **Infiniband overhead**. The Infiniband protocol implements headers inside the UDP payload. A simple WRITE message through reliable connection (RC) needs 12-byte Base Transport Header (BTH), 16-byte RDMA Extended Transport Header (RETH), and 4-byte invariant CRC. Hence, the Infiniband protocol overhead is at least 32-byte [2].

To transmit the payload from the host DRAM to the RNIC, the RNIC PCIe behaviors include the following overhead.

1. **Ringing the doorbell**. To post a work request, users need to ring the RNIC's doorbell through memory-mapped IO (MMIO). Each MMIO has a fixed aligned size 64-byte.

2. **Work Queue Element**. The RNIC needs to fetch a work queue element (WQE) from host DRAM to the NIC. A WQE for RC/UC is 36-byte, and 68-byte for UD.

3. **TLP overhead**. Each PCIe transaction has PCIe Transaction Layer Packet (TLP) header, and the header size varies for different PCIe implementation. We assume its least size as 20-byte according to [29, 69].

We next shows the computation of the 29-byte payload example in §3. The 29-byte payload is obviously less than the MTU, and can be sent using a single network packet. Therefore, the network bytes consumed by this payload is:

$$\begin{aligned}\text{Bytes(network)} &= \text{Bytes(payload)} + \text{Bytes(Ethernet)} \\ &+ \text{Bytes(IP)} + \text{Bytes(UDP)} + \text{Bytes(IB)} \\ &= 29 + 38 + 20 + 8 + 32 \\ &= 127\text{(bytes)}\end{aligned}$$

For PCIe consumption, the 29-byte payload is larger than the maximal inline size (28-byte). So it cannot be delivered

in the same PCIe transaction as the WQE. It therefore needs three PCIe transactions: (1) Doorbell, (2) WQE, and (3) payload, and consume the following bytes:

$$\begin{aligned}\text{Bytes(PCIe)} &= \text{Bytes(payload)} + \text{Bytes(payload TLP)} \\ &+ \text{Bytes(WQE)} + \text{Bytes(WQE TLP)} \\ &+ \text{Bytes(Doorbell)} + \text{Bytes(DB TLP)} \\ &= 29 + 20 + 36 + 20 + 64 + 20 \\ &= 189\text{(bytes)}\end{aligned}$$

Therefore, the PCIe consumption for such payload when saturating the link capacity (100 Gbps) is:

$$\begin{aligned}\text{Bandwidth(PCIe)} &= \text{Bandwidth(network)} * \frac{\text{Bytes(PCIe)}}{\text{Bytes(network)}} \\ &= 100 * \frac{189}{127} = 148.8\text{(Gbps)}\end{aligned}$$

B Response from NIC Vendors

We report our findings and results to the NIC vendors, including NVIDIA, Intel, and Chelsio. NVIDIA, one of the largest RDMA NIC vendors, has spent substantial effort on acknowledging and reproducing our experiments. They have successfully reproduced all of our findings in their own environment. In addition, NVIDIA provides us with detailed analysis and feedback. We would like to share them here.

Key finding #1: control verbs can cause excessive cache misses and a drastic performance reduction. NVIDIA provides a more accurate analysis of this finding: the deregistration control verbs can cause drastic performance reduction mainly because of the NIC internal QoS scheduling policy. The deregistration control verbs have higher priority than other types of operations and will be scheduled first. Consequently, these deregistration verbs trigger excessive cache misses and cause the performance to drop drastically. NVIDIA has already figured out a solution to address this issue. The high-level idea is to tune the NIC internal QoS policy so that deregistration does not have such a high priority. They are planning for a firmware upgrade to fix this issue.

Key finding #2: performance interference between different data verbs depends on the complexity of verbs. NVIDIA is familiar with this phenomenon and will roll out new firmware upgrades to address this issue.

Key finding #3: error handling can stall RNIC processing units and hang all the applications. NVIDIA provides a more accurate explanation of this phenomenon: for unreliable transport types (UC and UD), there is not the same specific RNR exception handling procedure as RC. Instead, they have other processing logic that involves firmware that handles out-of-order packets. This is the root cause of the performance interference when attacking using unreliable transport types. NVIDIA also provides a potential solution to mitigate such interference. NVIDIA Connect-X series NICs support monitoring per-VM consumption of the NIC resources. The cloud operators therefore can enforce VM capabilities policy based on the visibility of NIC resources consumption. Furthermore, NVIDIA is planning to introduce an additional layer of protection in the coming NIC firmware/hardware release to completely eliminate the attack vector for RC.

Key finding #4: PCIe bandwidth will only become the bottleneck when the request size is in a specific range. Though PCIe bandwidth contention is not a unique interference brought by RDMA, NVIDIA still acknowledged and confirmed our observation on the PCIe consumption for RDMA NIC.

We thank NVIDIA for their kind and great support. We believe the above understanding will benefit cloud operators and RDMA application developers. In addition, our collaboration with NVIDIA also demonstrates how Husky can help to improve existing RDMA solutions and build robust RDMA performance isolation in the future.

Empowering Azure Storage with RDMA

Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg*, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu*, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel*, Jordan Rhee*, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun*, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, Brian Zill

Microsoft

Abstract

Given the wide adoption of disaggregated storage in public clouds, networking is the key to enabling high performance and high reliability in a cloud storage service. In Azure, we choose Remote Direct Memory Access (RDMA) as our transport and aim to enable it for both storage frontend traffic (between compute virtual machines and storage clusters) and backend traffic (within a storage cluster) to fully realize its benefits. As compute and storage clusters may be located in different datacenters within an Azure region, we need to support RDMA at regional scale.

This work presents our experience in deploying intra-region RDMA to support storage workloads in Azure. The high complexity and heterogeneity of our infrastructure bring a series of new challenges, such as the problem of interoperability between different types of RDMA network interface cards. We have made several changes to our network infrastructure to address these challenges. Today, around 70% of traffic in Azure is RDMA and intra-region RDMA is supported in all Azure public regions. RDMA helps us achieve significant disk I/O performance improvements and CPU core savings.

1 Introduction

High performance and highly reliable storage is one of the most fundamental services in public clouds. In recent years, we have witnessed significant improvements in storage media and technologies [73] and customers also desire similar performance in the cloud. Given the wide adoption of disaggregated storage in the cloud [35, 46], the network interconnecting compute and storage clusters becomes a key performance bottleneck for cloud storage. Despite the sufficient bandwidth capacity provided by Clos-based network fabrics [25, 48], the legacy TCP/IP stack suffers from high processing delay, low single-core throughput, and high CPU consumption, thus making it ill-suited for this scenario.

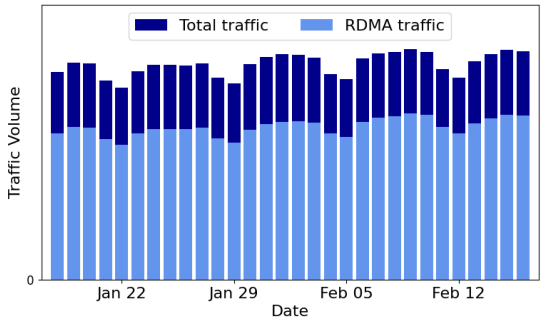


Figure 1: Traffic statistics of all Azure public regions between January 18 and February 16, 2023. Traffic was measured by collecting switch counters of server-facing ports on all Top of Rack (ToR) switches. Around 70% of traffic was RDMA.

Given these limitations, Remote Direct Memory Access (RDMA) offers a promising solution. By offloading the network stack to the network interface card (NIC) hardware, RDMA achieves ultra-low processing latency and high throughput with near zero CPU overhead. In addition to performance improvements, RDMA also reduces the number of CPU cores reserved on each server for network stack processing. These saved CPU cores can then be sold as customer virtual machines (VMs) or used for application processing.

To fully utilize the benefits of RDMA, we aim to enable it for *both* storage frontend traffic (between compute VMs and storage clusters) and backend traffic (within a storage cluster). This is different from previous work [46] that targets RDMA only for the storage backend. In Azure, due to capacity issues, corresponding compute and storage clusters may be located in different datacenters within a region. This imposes a requirement that our storage workloads rely on support for RDMA at regional scale.

In this paper, we summarize our experience in deploying intra-region RDMA to support Azure storage workloads.

*Albert Greenberg is now with Uber. Chen Liu is now with Meta. Shachar Raindel and Jordan Rhee are now with Google. Weixiang Sun is now with a stealth startup. This work was performed when they were with Microsoft.

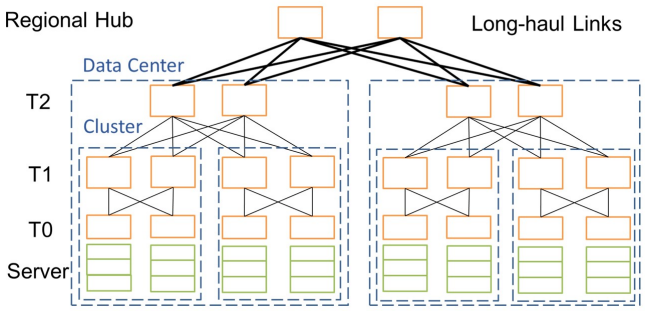


Figure 2: The network architecture of an Azure region.

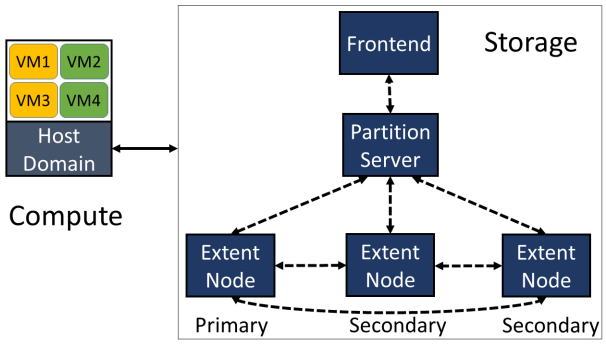


Figure 3: High-level architecture of Azure storage.

Compared to previous RDMA deployments [46, 50], intra-region RDMA deployment introduces many new challenges due to high complexity and heterogeneity within Azure regions. As Azure infrastructure keeps evolving incrementally, different clusters may be deployed with different RDMA NICs. While all the NICs support DCQCN [112], their implementations are very different. This results in many undesirable behaviors when different NICs communicate with each other. Similarly, heterogeneous switch software and hardware from multiple vendors significantly increase our operational effort. In addition, long-haul cables interconnecting datacenters cause large propagation delays and large round-trip time (RTT) variations within a region. This brings new challenges to congestion control.

We have made several changes to our network infrastructure, from application layer protocols to link layer flow control, to safely enable intra-region RDMA for Azure storage traffic. We developed new RDMA-based storage protocols with many optimizations and failover support, and seamlessly integrated them into the legacy storage stack (§4). We built RDMA Estats to monitor the status of the host network stack (§5). We leveraged SONiC to enforce a unified software stack across different switch platforms (§6). We updated firmware of NICs to unify their DCQCN behaviors and used the combination of Priority-based Flow Control (PFC) and DCQCN to achieve high throughput, low latency and near zero packet losses (§7).

In 2018, we started to enable RDMA for storage backend traffic. In 2019, we started to enable RDMA to serve customer frontend traffic. Figure 1 gives traffic statistics of all Azure public regions between January 18 and February 16, 2023. As of February 2023, around 70% of traffic in Azure was RDMA and intra-region RDMA was supported in all Azure public regions. RDMA helps us achieve significant disk I/O performance improvements and CPU core savings.

2 Background

In this section, we first present background on Azure's network and storage architecture. Then, we introduce the motivation for and challenges to enabling intra-region RDMA.

2.1 Network Architecture of an Azure Region

In cloud computing, a region [2, 5, 8] is a group of datacenters deployed within a latency-defined perimeter. Figure 2 shows the simplified topology of an Azure region. The servers within a region are connected through an Ethernet-based Clos network with four tiers of switches[1]: tier 0 (T0), tier 1 (T1), tier 2 (T2) and regional hub (RH). We use external BGP (eBGP) for routing and equal-cost multi-path (ECMP) for load balancing. We deploy the following four types of units.

- Rack: a T0 switch and the servers connected to it.
- Cluster: a set of racks connected to the same set of T1 switches.
- Datacenter: a set of clusters connected to the same set of T2 switches.
- Region: datacenters connected to the same set of RH switches. In contrast with short links (several to hundreds of meters) in datacenters [50], T2 and RH switches are connected by *long-haul* links whose lengths can be as long as tens of kilometers.

There are two thing to notice about this architecture. First, due to long-haul links between T2 and RH, the base round-trip time (RTT) varies from a few microseconds within a datacenter to as large as 2 milliseconds within a region. Second, we use two types of switches: pizza box switches for T0 and T1, and chassis switches for T2 and RH. The pizza box switch, which has been widely studied in the research community, typically has a single switch ASIC with shallow packet buffers [31]. In contrast, chassis switches are built using multiple switch ASICs with deep packet buffers based on the Virtual Output Queue (VoQ) architecture [3, 6].

2.2 High Level Architecture of Azure Storage

In Azure, we disaggregate compute and storage resources for cost savings and auto-scaling. There are two main types of

[1] In this paper, we use switch to denote the layer 3 switch which can perform IP routing. We use the terms switch and router interchangeably.

clusters in Azure: compute and storage. VMs are created in compute clusters but the actual storage of Virtual Hard Disks (VHDs) resides in storage clusters.

Figure 3 shows the high-level architecture of Azure storage [35]. Azure storage has three layers: the frontend layer, the partition layer, and the stream layer. The stream layer is an append-only distributed file system. It stores bits on the disk and replicates them for durability, but it does not understand higher level storage abstractions, e.g., Blobs, Tables and VHDs. The partition layer understands different storage abstractions, manages partitions of all the data objects in a storage cluster, and stores object data on top of the stream layer. The daemon processes of the partition layer and the stream layer are called the Partition Server (PS) and the Extent Node (EN), respectively. PS and EN are co-located on each storage server. The frontend (FE) layer consists of a set of servers that authenticate and forward incoming requests to corresponding PSs. In some cases, FE servers can also directly access the stream layer for efficiency.

When a VM wants to write to its disks, the disk driver running in the host domain of the compute server issues I/O requests to the corresponding storage cluster. The FE or PS parses and validates the request, and generates requests to corresponding ENs in the stream layer to write the data. At the stream layer, a file is essentially an ordered list of large storage chunks called *"extents"*. To write a file, data is appended to the end of an active *extent*, which is replicated three times in the storage cluster for durability. Only after receiving successful responses from all the ENs, the FE or PS sends the final response back to the disk driver. In contrast, disk reads are different. The FE or PS reads data from any EN replica and sends the response back to the disk driver.

In addition to user-facing workloads, there are also many background workloads in the storage clusters, e.g., garbage collection and erasure coding [57]. We classify our storage traffic into two categories: frontend (between compute and storage servers, e.g., VHD write and read requests) and backend (between storage servers, e.g., replication and disk reconstruction). Our storage traffic has incast-like characteristics. The most typical example is data reconstruction, which is implemented in the stream layer [57]. The stream layer erasure codes a sealed extent to several fragments, and then sends encoded fragments to different servers to store. When the user wants to read a fragment which is unavailable due to a failure, the stream layer will read the other fragments from multiple storage servers to reconstruct the target fragment.

2.3 Motivation for Intra-Region RDMA

Storage technology has improved significantly in recent years. For example, Non-Volatile Memory Express (NVMe) Solid-State Drives (SSDs) can provide tens of Gbps of throughput with request latencies in the hundreds of microseconds [105]. Many customers demand similar performance in the cloud. High performance cloud storage solutions [1, 4] impose stringent performance requirements to the underlying network due to the disaggregated and distributed storage architecture (§2.2). While datacenter networks generally provide sufficient bandwidth capacity, the legacy TCP/IP stack in the OS kernel becomes a performance bottleneck due to its high processing latency and low single-core throughput. What is worse, the performance of the legacy TCP/IP stack also depends on OS scheduling. To provide predictable storage performance, we must reserve enough CPU cores on both compute and storage nodes for the TCP/IP stack to process peak storage workloads. Burning CPU cores takes away the processing power that could otherwise be sold as customer VMs, thus increasing the overall cost of providing cloud services.

Given these limitations, RDMA offers a promising solution. By offloading the network stack to the NIC hardware, RDMA achieves predictable low processing latency (a few microseconds) and high throughput (line rate for a single flow) with near zero CPU overhead. In addition to its performance benefits, RDMA also reduces the number of CPU cores reserved on each server for network stack processing. These saved CPU cores can then be sold as customer VMs or used for storage request processing.

To fully achieve the benefits of RDMA, we must enable RDMA for both storage frontend traffic and backend traffic. Enabling RDMA for backend traffic is relatively easy because almost all the backend traffic stays within a storage cluster. In contrast, frontend traffic crosses different clusters within a region. Even though we try to co-locate corresponding compute and storage clusters to minimize latency, sometimes they may still end up located in different datacenters within a region due to capacity issues. This imposes the requirement that our storage workloads rely on support for RDMA at regional scale.

2.4 Challenges

We faced many challenges when enabling intra-region RDMA because our design was limited by many practical constraints.

Practical considerations: We aimed to enable intra-region RDMA over the legacy infrastructure. While we had some flexibility to reconfigure and upgrade software stacks, e.g., the NIC driver, the switch OS, and the storage stack, it was *operationally infeasible* to replace the underlying hardware, e.g., the NICs and switches. Hence, we adopted RDMA over commodity Ethernet v2 (RoCEv2) [29] to keep compatibility with our IP-routed networks (§2.1). Before starting this project, we had deployed a significant number of our first generation RDMA NICs, which implement go-back-N retransmission in the NIC firmware with limited processing capacity. Our measurements showed that it took hundreds of microseconds to recover a lost packet, which was even worse than the TCP/IP software stack. Given such a large performance degradation, we made the decision to adopt Priority-based Flow Control

(PFC) [60] to eliminate packet losses due to congestion.

Challenges: Before this project, we had deployed RDMA in some clusters to support Bing services [50], and we learnt several lessons from this deployment. Compared to intra-cluster RDMA deployments [46, 50], intra-region RDMA deployments introduce many new challenges due to the high complexity and heterogeneity of the infrastructure.

- Heterogeneous NICs: Cloud infrastructure keeps evolving incrementally, often one cluster or one rack at a time with the latest generation of server hardware [91]. Different clusters within a region may have different NICs. We have deployed three generations of commodity RDMA NICs from a popular NIC vendor: Gen1, Gen2 and Gen3. Each NIC generation has a different implementation of DCQCN. This results in many undesired interactions when different NIC generations communicate with each other.
- Heterogeneous switches: Similar to server infrastructure, we keep deploying new switches to reduce costs and increase the bandwidth capacity. We have deployed many switch ASICs and multiple switch OSes from different vendors. However, this has increased our operational effort significantly because many aspects are vendor specific, for example, buffer architectures, sizes, allocation mechanisms, monitoring and configuration, etc.
- Heterogeneous latency: As shown in §2.1, there are large RTT variations from several microseconds to 2 milliseconds within a region, due to long-haul links between T2 and RH. Hence, RTT fairness re-emerges as a key challenge. In addition, the large propagation delay of long-haul links also imposes large pressure on PFC headroom [12].

Like other services in public clouds, availability, diagnosis, and serviceability are key aspects for our RDMA storage system. To achieve high availability, we always prepare for unexpected *zero-day* problems despite large investments in testing. Our system must detect performance anomalies and perform automatic failover if necessary. To understand and debug faults, we must build fine-grained telemetry systems to deliver crystal clear visibility into every component in the end-to-end path. Our system also must be serviceable: storage workloads should survive NIC driver updates and switch software updates.

3 Overview

We have made several changes to our network infrastructure, from application layer protocols to link layer flow control, to safely empower Azure storage with RDMA. We developed two RDMA-based protocols: sU-RDMA (§4.1) and sK-RDMA (§4.2), which we have seamlessly integrated into our legacy storage stack to support backend communication and frontend communication, respectively. Between the storage protocols and the NIC, we deployed a monitoring system RDMA Estats (§5), giving us visibility into the host network stack by providing an accurate breakdown of cost for each RDMA operation.

In the network, we use the combination of PFC and DCQCN [112] to achieve high throughput, low latency, and near zero losses due to congestion. DCQCN and PFC were the state-of-the-art commercial solutions when we started the project. To optimize the customer experience, we use two priorities to isolate storage frontend traffic and backend traffic. To mitigate the switch heterogeneity problem, we developed and deployed SONiC [15] to provide a unified software stack across different switch platforms (§6). To mitigate the interoperability problem of heterogeneous NICs, we updated the firmware of NICs to unify their DCQCN behaviors (§7). We carefully tuned DCQCN and switch buffer parameters to optimize performance across different scenarios.

3.1 PFC Storm Mitigation Using Watchdogs

We use PFC to prevent congestion packet losses. However, malfunctioning NICs and switches can continually send PFC pause frames in the absence of congestion [50], thus *completely* blocking the peer device for a long time. Moreover, these endless PFC pause frames can eventually propagate into the whole network, thus causing collateral damage to innocent devices. Such *endless* PFC pause frames are called a PFC storm. In contrast, normal congestion-triggered PFC pause frames only *slow down* the data transmission of the peer device through *intermittent* pauses and resumes.

To detect and mitigate PFC storms, we designed and deployed a PFC watchdog [11, 50] on every switch and bump-in-the-wire FPGA card [42] between T0 switches and servers. When the PFC watchdog detects that a queue has been in the paused state for an abnormally long duration, e.g., hundreds of milliseconds, it disables PFC and drops all the packets on this queue, thereby preventing PFC storms from propagating into the whole network.

3.2 Security

We use RDMA to empower first-party storage traffic in a trusted environment, including storage servers, the host domain of compute servers, switches and links. Therefore we are secure against issues described in [69, 94, 104, 109].

4 Storage Protocols over RDMA

In this section, we introduce two storage protocols built on top of RDMA Reliable Connections (RC): sU-RDMA and sK-RDMA. Both protocols aim to optimize performance while keeping good compatibility with legacy software stacks.

4.1 sU-RDMA

sU-RDMA [87] is used for storage backend (storage to storage) communication. Figure 4 shows the architecture of our

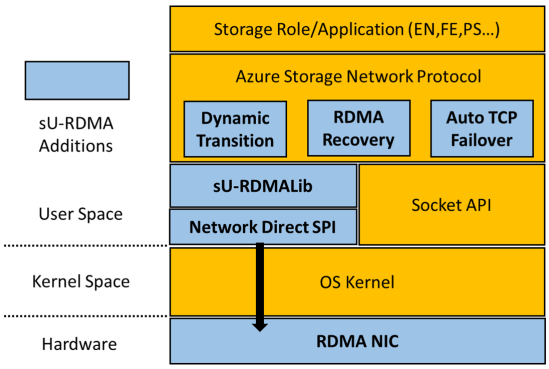


Figure 4: Azure storage backend network stack.

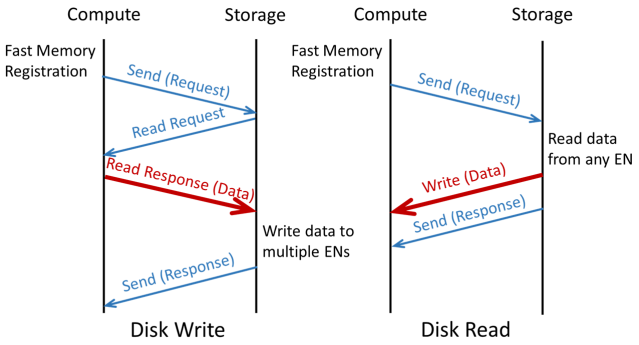


Figure 5: sK-RDMA's data flow. We use blue arrows and red arrows to represent control messages and data massages, respectively. Arrow width represents data size.

storage backend network stack with the sU-RDMA modules highlighted. The Azure Storage Network Protocol is an RPC protocol directly used by applications to send request and response objects. It leverages socket APIs to implement connection management, sending and receiving messages.

To simplify RDMA integration with storage stack, we built sU-RDMALib, a user space library that exposes socket-like byte-stream APIs to upper layers. To map socket-like APIs to RDMA operations, sU-RDMALib needs to handle the following challenges:

- When the RDMA application cannot directly write into an existing memory regions (MR), it must either register the application buffer as a new MR or copy its data into an existing MR. Both options can introduce large latency penalties and we should minimize these overhead.
- If we use RDMA `Send` and `Receive`, the receiver must pre-post enough `Receive` requests.
- The RDMA sender and receiver must be in agreement on the size of data being transferred.

To reduce memory registrations, which are especially expensive for small messages [44], sU-RDMALib maintains a common buffer pool of pre-registered memory shared across multiple connections. sU-RDMALib also provides APIs to allow applications to request and release registered buffers. To avoid Memory Translation Table (MTT) cache misses on the NIC [50], sU-RDMALib allocates large memory slabs from the kernel and registers memory over these slabs. This buffer pool can also autoscale based on runtime usage. To avoid overwhelming the receiver, sU-RDMALib implements a receiver-driven credit-based flow control where credits represent the resources (e.g., available buffers and posted `Receive` requests) allocated by the receiver. The receiver sends credit update messages back to the sender regularly. When we started designing sU-RDMALib, we did consider using RDMA `Send` and `Receive` with a fixed buffer size *S* for each `Send`/`Receive` request to transfer data. However, this design causes a dilemma. If we use a large *S*, we may waste much memory space because a `Send` request fully uses the receive buffer of the `Receive` request, regardless of its actual message size. In contrast, a small *S* causes large data fragmentation overhead. Hence, sU-RDMALib uses three transfer modes based on the message size [87].

- Small messages: Data is transferred using RDMA `Send` and `Receive`.
- Medium messages: The sender posts a RDMA `Write` request to transfer data, and a `Send` request with "Write Done" to notify the receiver.
- Large messages: The sender first posts a RDMA `Send` request carrying the description of the local data buffer to the receiver. Then the receiver posts a `Read` request to *pull* the data. Finally, the receiver posts a `Send` request with "Read Done" to notify the sender.

On top of sU-RDMALib, we built modules to enable dynamic transitions between TCP and RDMA, which is critical for failover and recovery. The transition process is gradual. We periodically close a small portion of all connections and establish new connections using the desired transport.

Unlike TCP, RDMA uses rate based congestion control [112] without tracking the number of in-flight packets (the window size). Hence, RDMA tends to inject excessive in-flight packets, thus triggering PFC. To mitigate this, we implemented a static flow control mechanism in the Azure Storage Network Protocol by dividing a message into fixed-sized chunks and only allowing a single in-flight chunk for each connection. Chunking can significantly improve performance under high-degree incast with negligible CPU overhead.

4.2 sK-RDMA

sK-RDMA is used for storage frontend (compute to storage) communication. In contrast with sU-RDMA which runs RDMA in user space, sK-RDMA runs RDMA in kernel space. This enables the disk driver, which runs in kernel space in the host domain of compute servers, to directly use sK-RDMA to

issue network I/O requests. sK-RDMA leverages and extends Server Message Block (SMB) Direct [14] which provides socket-like kernel-mode RDMA interfaces. Similar to sU-RDMA, sK-RDMA also provides credit-based flow control and dynamic transition between RDMA and TCP.

Figure 5 shows sK-RDMA's data flow for reading and writing disks. The compute server first posts a Fast Memory Registration (`FMR`) request to register data buffers. Then it posts an RDMA `Send` request to transfer a request message to the storage server. The request carries a disk I/O command, and a description of `FMR` registered buffers available for RDMA access. According to the InfiniBand (IB) specification, the NIC should wait for the completion of the `FMR` request before processing any subsequently posted requests. Hence, the request message is actually pushed onto the wire after the memory registration. The data transfer is initiated by the storage server using RDMA `Read` or `Write`. After the data transfer, the storage server sends a response message to the compute server using RDMA `Send With Invalidate`.

To detect data corruptions, which can happen *silently* due to various software and hardware bugs along the path, both sK-RDMA and sU-RDMA implement a Cyclical Redundancy Check (CRC) on all application data. In sK-RDMA, the compute server calculates the CRC of the data for disk writes. These calculated CRCs are included in the request messages, and used by the storage server to validate the data. For disk reads, the storage server performs the CRC calculations and includes them in the response messages, and the compute server uses them to validate the data.

5 RDMA Estats

To understand and debug faults, we need fine-grained telemetry tools to capture behaviors of every component in the end-to-end path. Despite many existing tools [51, 97, 114] to diagnose switch and link faults, none of these tools gives us good visibility into the RDMA network stack at end hosts.

Inspired by diagnostic tools for TCP [79], we developed RDMA Extended Statistics (Estats) to diagnose performance problems in both the network and the host. If an RDMA application is performing poorly, RDMA Estats enables us to tell if the bottleneck is in the sender, the receiver, or the network.

To this end, RDMA Estats provides a fine-grained breakdown of latency for each RDMA operation, in addition to collecting regular counters such as bytes sent/received and number of NACKs. The requester NIC records timestamps at one or more measurement points as the work queue element (WQE) traverses the transmission pipeline. When a response (ACK or `read` response) is received, the NIC records additional timestamps at measurement points along the receive pipeline (Figure 6). The following measurement points are required in any RDMA Estats implementation in Azure

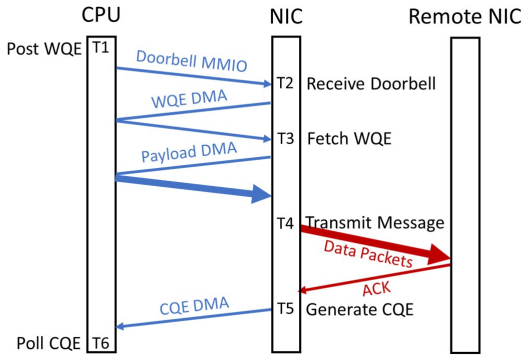


Figure 6: RDMA Estats measurement points. There are four NIC timestamps and two host timestamps. We use blue arrows and red arrows to represent PCIe transactions and network transfers, respectively. Arrow width represents data size.

T_1: WQE posting: Host processor timestamp when the WQE is posted to the submission queue.

T_5: CQE generation: NIC timestamp when the completion queue element (CQE) is generated in the NIC.

T_6: CQE polling: Host timestamp when the CQE is polled by software.

In Azure, the NIC driver reports various latencies derived from the above timestamps. For example, $T_6 - T_1$ is the operation latency seen by the RDMA consumer, while $T_5 - T_1$ is the latency seen by the NIC. A user-mode agent groups the latency samples by connection, operation type, and (success/-failure) status to create latency histograms for each group. By default, a histogram covers a one-minute interval. Each histogram's quantiles and summary statistics are fed into Azure's telemetry pipeline. As our diagnostics evolved, we added to our user-mode agent the ability to collect and upload NIC and QP state dumps during high latency events. Finally, we extended the scope of event-triggered data collection by the user-mode agent to include NIC statistics and state dumps in case of events not specific to RDMA (e.g., servicing operations that impact connectivity).

The collection of latency samples adds overhead to the WQE posting and completion processing code paths. This overhead is dominated by keeping the NIC and host time stamps synchronized. To reduce the overhead, we developed a clock synchronization procedure that attempts to minimize the frequency of reading the NIC clock registers, while maintaining low deviations.

RDMA Estats can significantly reduce the time to debug and mitigate storage performance incidents by quickly ruling out (or in) network latency. In §8.3, we share our experience in diagnosing the FMR hidden fence bug using RDMA Estats.

6 Switch Management

6.1 Overcoming Heterogeneity with SONiC

Our RDMA deployment heavily relies on the support of switches. However, heterogeneous switch ASICs and OSes from multiple vendors have brought significant challenges to network management. For example, commercial switch OSes are designed to satisfy diverse requirements of all the customers, thus leading to complex software stacks and slow feature evolution [39]. In addition, different switch ASICs provide different buffer architectures and mechanisms, thus increasing the effort to qualify and test them for Azure's RDMA deployment.

Our solutions to the above challenges were two-fold. On one hand, we worked closely with our vendors to define concrete feature requirements and test plans, and to understand their low-level implementation details. On the other hand, in collaboration with many partners, we developed and deployed an in-house cross-platform switch OS called Software for Open Networking in the Cloud (SONiC) [15]. Based on a Switch Abstraction Interface (SAI) [20], SONiC manages heterogeneous switches from multiple vendors with a simplified and unified software stack. It breaks apart monolithic switch software into multiple containerized components. Containerization provides clean isolation, improves development agility, and enables choices on a per-component basis. Network operators can customize SONiC with only the features they require, thereby creating a "lean stack".

6.2 Buffer Model and Configuration Practices of SONiC on Pizza Box Switches

SONiC provides all the features required by RDMA deployments, such as ECN marking, PFC, a PFC watchdog (§3.1) and a shared buffer model. In the interest of space, we briefly introduce the buffer model and configuration practices of SONiC on pizza box switches, which are used at T0 and T1 (§2.1). We provide a buffer configuration example in §*A*.

We typically allocate three buffer pools on a pizza box switch: (1) the `ingress_pool` for ingress admission control of all packets, (2) the `egress_lossy_pool` for egress admission control of lossy packets, and (3) the `egress_lossless_pool` for egress admission control of lossless packets. Note that these buffer pools and queues are not backed by separate dedicated buffers, but instead are essentially counters applied to a single physical shared buffer and used for admission control purposes. Each counter is updated only by the packets mapped to it, and the same packet can be mapped to multiple queues and pools simultaneously. For example, a lossless (lossy) packet of priority p from source port s to destination port d updates ingress queue (s, p), egress queue (d, p), `ingress_pool` and `egress_lossless_pool` (`egress_lossy_pool`). A packet is accepted only if it passes both ingress and egress admission controls. Counters increment by the size of the admitted packet, and decrement by the size of the departing packet. We use both dynamic thresholds [40] and static thresholds to limit the queue lengths.

We apply ingress admission control only to lossless traffic, and we apply egress admission control only to lossy traffic. If the switch buffer size is B, then the `ingress_pool` size must be smaller than B, reserving enough space for PFC headroom buffer (§7.1). When an ingress lossless queue hits the dynamic threshold, the queue enters the "paused" state, and the switch sends PFC pause frames to the upstream device. Future arriving packets on this ingress lossless queue use the PFC headroom buffer rather than `ingress_pool`. In contrast, for ingress lossy queues we configure a static threshold which equals to the switch buffer size B. Since ingress lossy queue lengths cannot hit the switch buffer size, lossy packets can *bypass* ingress admission control.

At egress, lossy and lossless packets are mapped to the `egress_lossy_pool` and `egress_lossless_pool`, respectively. We configure both the size of the `egress_lossless_pool` and the static thresholds for egress lossless queues to B so that lossless packets bypass egress admission control. In contrast, the size of the `egress_lossy_pool` must be no larger than the size of the `ingress_pool` because lossy packets should not use any of the PFC headroom buffer at ingress. Egress lossy queues are configured to use dynamic thresholds [40] to drop packets.

6.3 Testing RDMA Features with SONiC

We use nightly tests to track the quality of SONiC switches. In this section, we briefly introduce our methods for testing RDMA features with SONiC switches.

Software-based Tests: We leveraged the Packet Testing Framework (PTF) [10] to develop test cases for SONiC in general. PTF is mostly used for testing packet forwarding behaviors, with which testing RDMA features require additional effort.

Our testing approach is inspired by breakpoints in software debugging. To set a "breakpoint" for the switch, we first block the transmission of a switch port using SAI APIs. We then generate a series of packets destined for the blocked port and capture one or several snapshots of the switch states (e.g., buffer watermark), analogous to dumping the values of variables in software debugging. Next, we release the port and dump the received packets. We determine if the test passes by analyzing both the captured switch snapshots and the received packets. We use this approach to test buffer management mechanisms, buffer related counters, and packet schedulers.

Hardware-based Tests: While the above approach gives us good visibility into switch states and packet micro-behaviors, it cannot meet the stringent performance requirements of some tests. For example, to test PFC watchdog [50], we need to generate continuous PFC pause frames at high speed and accurately control their intervals due to the small pause duration

enforced by each PFC frame.

To conduct such performance-sensitive tests, we need to control traffic generation at μs or even ns timescales and have high-resolution measurement of data plane behaviors. This motivated us to build a hardware-based test system by leveraging hardware programmable traffic generators [9]. Our hardware-based system focuses on testing features like PFC, PFC watchdog, RED/ECN marking.

As of February 2023, we built 32 software test cases and 50 hardware test cases for RDMA features. The documentation and implementation of our test cases are available at [18].

7 Congestion Control

We use the combination of PFC and DCQCN to mitigate congestion. In this section, we discuss how we scale both techniques at regional scale.

7.1 Scaling PFC over Long Links

Once an ingress queue pauses the upstream device, it requires a dedicated headroom buffer to absorb in-flight packets before the PFC pause frame takes effect on the upstream device [50, 112]. The ideal PFC headroom value depends on many factors, e.g., link capacity and propagation delay [12]. The total demand on the headroom buffer for a switch is also in proportion to the number of lossless priorities[2].

To extend RDMA from cluster scale [46, 50] to regional scale, we must deal with long links between T2 and RH (tens of kilometers), and between T1 and T2 (hundreds of meters), which demand much larger PFC headroom than that of intra-cluster links. At first glance, it may seem that a T1 switch in our production environment can reserve half of the total buffer for PFC headroom and other usages. At T2 and RH, given the high port density (100s) of chassis switches and long-haul links, we need to reserve several GB of PFC headroom buffer.

To scale PFC over long links, we leverage the fact that pathological cases, e.g., all the ports are congested simultaneously, and ingress lossless queues of a port pause peers sequentially, are likely to be rare. Our solution is two-fold. First, on chassis switches at T2 and RH, we use deep packet buffers of off-chip DRAM[3] to store RDMA packets. Our analysis shows that our chassis switches in production can provide abundant DRAM buffers for PFC headroom. Second, instead of reserving PFC headroom per queue, we allocate a PFC headroom pool shared by all the ingress lossless queues on the switch. Each ingress lossless queue has a static threshold to limit its maximum usage in the headroom pool. We oversubscribe the headroom pool size with a reasonable ratio, thus leaving more shared buffer space to absorb bursts. Our production experience shows that the oversubscribed PFC headroom pool can effectively eliminate congestion losses and improve burst tolerance.

[2]For an ingress port, the worst case is that its lossless queues *sequentially* pause the peer queues, and none of its packets can be drained from the buffer.

[3]Unlike on-chip SRAM, the bandwidth of off-chip DRAM is slightly smaller than the forwarding capacity of the switch ASIC. When all the ports send and receive traffic at line rate, DRAM will suffer from packet drops.

7.2 DCQCN Interoperability Challenges

We use DCQCN [112] to control the sending rate of each queue pair (QP). DCQCN consists of three entities: the sender or reaction point (RP), the switch or congestion point (CP), and the receiver or notification point (NP). The CP performs ECN marking at the egress queue based on the RED algorithm [43]. The NP sends Congestion Notification Packets (CNPs) when it receives ECN-marked packets. The RP reduces its sending rate when it receives CNPs. Otherwise, it leverages a byte counter and a timer to increase the rate.

We deployed three generations of commodity NICs from a popular NIC vendor: Gen1, Gen2 and Gen3, for different types of clusters. While all of them support DCQCN, their implementation details differ significantly. This causes an interoperability problem when different generations of NICs communicate with each other.

DCQCN implementation differences: On Gen1, most of the DCQCN functionality, such as the NP and RP state machines, is implemented in firmware. Given the limited processing capacity of the firmware, Gen1 minimizes CNP generation through coalescing at the NP side. As described in [112], the NP generates at most one CNP in a time window for a flow, if any arriving packets within this window are ECN marked. Correspondingly, the RP reduces the sending rate upon receiving a CNP. In addition, Gen1 also has limited cache resources. Cache misses can significantly impact RDMA's performance [50, 63]. To mitigate cache misses, we increase the granularity of rate limiting on Gen1 from a single packet to a burst of packets. Burst transmissions can effectively reduce the number of active QPs in a fixed interval, thus lowering pressure on the very limited cache resources of Gen1 NICs.

In contrast, Gen2 and Gen3 have hardware-based DCQCN implementations and adopt a RP-based CNP coalescing mechanism, which is the exact opposite of the NP-based CNP coalescing used by Gen1. In Gen2 and Gen3, the NP sends a CNP for every arriving ECN-marked packet. However, the RP only cuts the sending rate for a flow at most once in a time window if it receives any CNPs within that window. It is worthwhile to note that RP-based and NP-based CNP coalescing mechanisms essentially provide the same congestion notification granularity. The rate limiting is on a per-packet granularity on Gen2 and Gen3.

Interoperability challenges: Storage frontend traffic, which crosses different clusters, may lead to communication between different generations of NICs. In this scenario, the DCQCN implementation differences cause undesirable behaviors. First, when a Gen2/Gen3 node sends traffic to a Gen1 node, its per-packet rate limiting tends to trigger many cache misses

on the Gen1 node, thus slowing down the receiver pipeline. Second, when a Gen1 node sends traffic to a Gen2/Gen3 node through a congested path, the Gen2/Gen3 NP tends to send excessive CNPs to the Gen1 RP, thus causing excessive rate reductions and throughput losses.

Our solution: Given the limited processing capacity and resources of Gen1, we cannot make it behave like Gen2 and Gen3. Instead, we try to make Gen2 and Gen3 behave like Gen1 as much as possible. Our solution is two-fold. First, we move the CNP coalescing on Gen2 and Gen3 from the RP side to the NP side. On the Gen2/Gen3 NP side, we add a per-QP CNP rate limiter and set the minimal interval between two consecutive CNPs to the value of CNP coalescing timer of the Gen1 NP. On the Gen2/Gen3 RP side, we minimize the time window for rate reduction so that the RP almost always reduces the rate upon receiving a CNP. Second, we enable per-burst rate limiting on Gen2 and Gen3.

7.3 Tuning DCQCN

There were certain practical limitations when we tuned DCQCN in Azure. First, our NICs only support global DCQCN parameter settings. Second, to optimize customer experience, we classify RDMA flows into two switch queues based on their application semantics, rather than RTTs. Hence, instead of using different DCQCN parameters for inter-datacenter and intra-datacenter traffic, we use global DCQCN parameter settings (on the NICs and switches) that work well given the large RTT variations within a region.

We took a three-step approach to tune DCQCN parameters. First, we leveraged the fluid model [113] to understand theoretical proprieties of DCQCN. Second, we ran experiments with synthetic traffic in our lab testbed to evaluate solutions to the interoperability problem and deliver reasonable parameter settings. Third, we finalized the parameter settings in test clusters, which use the same setup as production clusters carrying customer traffic. We ran stress tests with real storage applications and tuned DCQCN parameters based on the application performance.

To illustrate our findings, we use K_{min}, K_{max}, and P_{max} to denote the minimum threshold, the maximum threshold, and the maximum marking probability of RED/ECN [43], respectively. We make the following three key observations (more experiment results appear in §*B*):

- DCQCN does not suffer from RTT unfairness as it is a rate-based protocol and its rate adjustment is independent of RTT.
- To provide high throughput for DCQCN flows with large RTTs, we use *sparse* ECN marking with large $K_{max} - K_{min}$ and small P_{max}.
- DCQCN and switch buffers should be jointly tuned [112]. For example, before increasing K_{min}, we ensure that ingress thresholds for lossless traffic are large enough. Otherwise, PFC may be triggered before ECN marking.

8 Experience

In 2018, we started to enable RDMA to serve customer backend traffic. In 2019, we started to enable RDMA to serve customer frontend traffic, with storage and compute clusters co-located in the same datacenter. In 2020, we enabled intra-region RDMA in the first Azure region. As of February 2023, around 70% of traffic in Azure public regions was RDMA (Figure 1) and intra-region RDMA was supported in all Azure public regions.

8.1 Deployment and Servicing

We took a three-step approach to gradually enable RDMA in production environments. First, we leveraged the lab testbed to develop and test each individual component. Second, we conducted end-to-end stress tests in test clusters with the same software and hardware setups as those of production counterparts. In addition to normal workloads, we also injected common errors, e.g., random packet drops, to evaluate the robustness of the system. Third, we cautiously increased the deployment scale of RDMA in production environments to carry more customer traffic. During our deployment, NIC driver/firmware and switch OS updates were common. Thus it was crucial to minimize the impact of such updates to customer traffic.

Servicing switches: Compared to switches in T1 or tiers above, T0 switches, especially in compute clusters, were more challenging to service as they could be a single point of failure (SPOF) for customer VMs. In this scenario, we leveraged fast reboot [17] and warm reboot [19] to reduce the data plane disruption time from a few minutes to less than a second.

Servicing NICs: In some cases, servicing the NIC driver or firmware required unloading the NIC driver. The driver could safely unload only after all the NIC resources had been released. To this end, we needed to signal consumers, e.g., disk driver, to close RDMA connections and shift traffic to TCP. Once RDMA and other NIC features with similar concerns had been disabled, we could reload the driver.

8.2 Performance

Storage backend: Currently almost all the storage backend traffic in Azure is RDMA. It is no longer feasible to run large-scale A/B tests with customer traffic because the CPU cores saved by RDMA have been used for other purposes, not to mention customer experience degradation. Hence we demonstrate results of an A/B test conducted in a test cluster in 2018. In this test, we ran storage workloads with high transactions per second (TPS) and switched transport between RDMA and TCP. Figure 7 plots normalized CPU utilization of storage

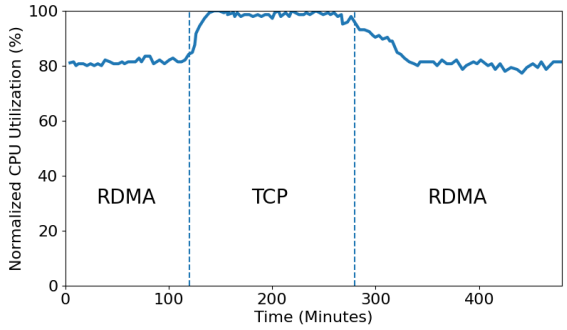


Figure 7: Average CPU usage of storage servers of a storage tenant. We normalize results to the maximum CPU usage. We switched traffic between RDMA and TCP twice.

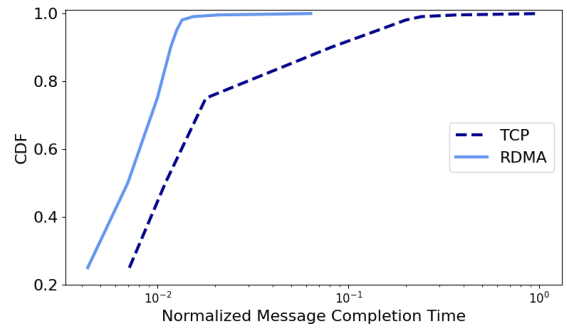


Figure 8: Message completion times of storage backend traffic measured in a test cluster. We normalize results to the maximum message completion time.

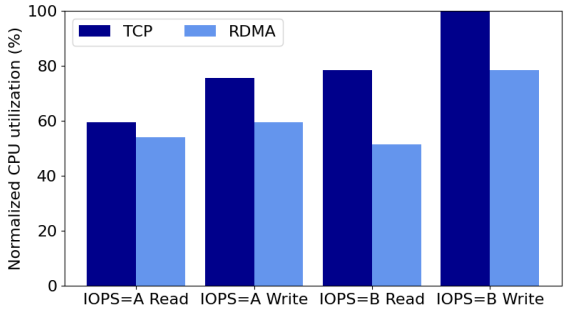


Figure 9: Average CPU usage of the host domain. We normalize results to the maximum value.

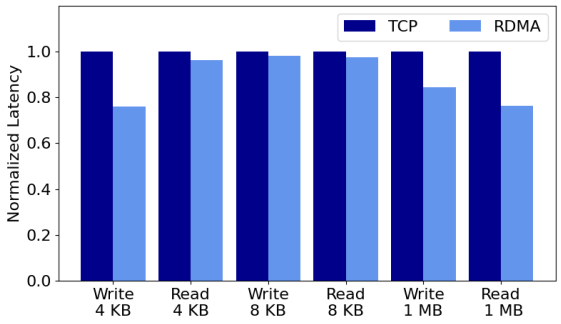


Figure 10: Average access latencies of a type of SSDs across *all* Azure public regions between February 22, 2022, and February 22, 2023. We normalize RDMA results to corresponding TCP results.

servers during two transport switches. It is worthwhile to note that CPU utilization here includes all the types of processing overhead, e.g., storage application, Azure Storage Network Protocol, and TCP/IP stack. Figure 8 gives message completion times measured in Azure Storage Network Protocol layer (Figure 4), which excludes the overhead of application processing. Compared to TCP, RDMA achieved obvious CPU saving and significantly accelerated network data transfer.

Storage frontend: Since we cannot perform large-scale A/B tests with customer traffic, we present results of an A/B test conducted in a test cluster in 2018. In this test, we used `DiskSpd` to generate read and write workloads at A IOPS and B IOPS (A < B). The I/O size was 8 KB. Figure 9 gives average CPU utilization of the host domain during the test period. Compared to TCP, RDMA could reduce the CPU utilization by up to 34.5%.

To understand the performance improvement introduced by RDMA, we leverage an always-on storage monitoring service. This service allocates some VMs in each region, uses them to periodically generate disk read and write workloads, and collects end-to-end performance results. The monitoring service covers different I/O sizes, types of disks, and transports for storage frontend traffic.

Figure 10 shows the overall average access latencies of a type of SSDs across all Azure public regions collected by the monitoring service for a year. Note that the RDMA and TCP in this figure only refer to the transport of frontend traffic generated by test VMs. We normalize RDMA results to corresponding TCP results. Compared to TCP, RDMA yielded better access latencies with every I/O size. In particular, 1 MB I/O requests benefited the most from RDMA with 23.8% and 15.6% latency reductions for read and write, respectively. This is due to the fact that large I/O requests are more sensitive to throughput than smaller I/O requests, and RDMA improves throughput drastically since it can run at line rate using a single connection without slow starts.

Congestion control: We ran stress tests in a test cluster to drive the DCQCN parameter setting that could achieve reasonable performance even under peak workloads. Figure 11 gives results of the 99th percentile message completion time, the key metric we used to guide our tuning. At the beginning, we disabled DCQCN and only tuned switch buffer parame-

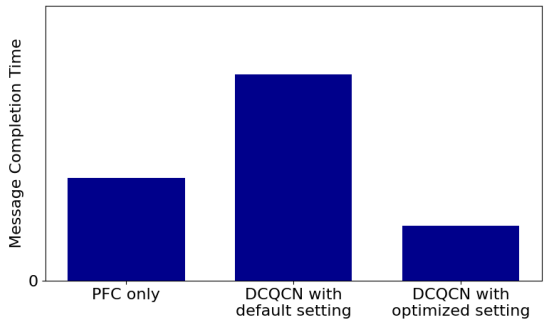


Figure 11: The 99th percentile message completion times of different schemes measured in a test cluster.

ters, e.g., the dynamic threshold of ingress lossless queues, to explore the best performance achieved by PFC only. After reaching the best performance of PFC only, we enabled DCQCN using the default parameter setting, which was derived on the lab testbed using synthetic traffic. While DCQCN reduced the number of PFC pause frames, it degraded the tail message completion time as the default setting reduced the sending rate too aggressively. Given this, we adjusted ECN marking parameters to improve DCQCN's throughput. With optimized setting, DCQCN performs better than using PFC alone. Our key takeaway from this tuning experience was that DCQCN and switch buffer should be jointly tuned to optimize the application performance, rather than PFC pause duration.

8.3 Problems Discovered and Fixed

During tests and deployments, we discovered and fixed a series of problems in NICs, switches and our RDMA applications.

FMR hidden fence: In sK-RDMA (§4.2), every I/O request from compute servers requires a `FMR` request followed by a `Send` request to the storage server, which contains the description of `FMR` registered memory and storage commands. Therefore, the send queue consists of many `FMR`/`Send` pairs.

When we deployed sK-RDMA in compute and storage clusters located in different datacenters, we found that the frontend traffic showed extremely low throughput, even though we kept many outstanding `FMR`/`Send` pairs in the send queue. To debug this problem, we used RDMA Estats to collect $T_5 - T_1$ latency for every `Send` request (§5). We found a strong correlation between $T_5 - T_1$ and inter-datacenter RTT, and noticed that there was only a single outstanding `Send` request per RTT. After we shared these findings with the NIC vendor, they identified the root cause: to simplify the implementation, NICs processed the `FMR` request only after the completions of previously posted requests. In sK-RDMA, the `FMR` request created a *hidden fence* between two `Send` requests, thus only allowing a single `Send` request in the air, which could not fill the large network pipe between datacenters. We have worked with the NIC vendor to fix this problem in the new NIC driver.

PFC and MACsec: After we enabled PFC on long-haul links between T2 and RH, many long-haul links reported high packet corruption rates, thus triggering alerts. It turned out that the MACSec standard [21] did not specify whether PFC frames should be encrypted. As a result, different vendors had no agreement on whether PFC frames sent should be encrypted and what to do with arriving encrypted PFC frames. For example, switch A may send unencrypted PFC frames to switch B, wile switch B was expecting encrypted PFC frames. As a result, switch B would treat those PFC frames as corrupted packets and report errors. We have worked with switch vendors to standardize how MACsec enabled switch ports treat PFC frames.

Congestion leaking: The problem was found in the testbed. When we enabled interoperability features (§7.2) on Gen2 NICs, we found that their throughput would be degraded. To dig into this problem, we used the water filling algorithm to calculate theoretical per-QP throughput results and compared them with actual throughput results measured from the testbed. We had two interesting observations when comparing the results. First, flows sent by a Gen2 NIC always had near identical sending rates regardless of their congestion degrees. Second, actual sending rates were very close to the theoretical sending rate of the slowest flow sent from the NIC. It seemed that all the flows from a Gen2 NIC were throttled by the slowest flow. We reported these observations to the NIC vendor, and they identified a head-of-line blocking in the NIC firmware. We have fixed this problem on all the NICs with interoperability features.

Slow receiver due to loopback RDMA: This problem was found in a test cluster. During stress tests, we found that a large number of servers sent PFC pause frames to T0 switches. However, unlike slow receivers found before, PFC watchdog was not triggered on any T0 switches. It seemed that those servers only gracefully slowed down the traffic coming from T0 switches, rather than completely blocking T0 switches for a long duration. In addition, where slow receivers were common at Azure's scale, it was very unlikely that a significant portion of servers in a cluster became "mad" simultaneously.

Based on the above observations, we suspected that these slow receivers were caused by our applications. We found that each server actually ran multiple RDMA application instances. All the inter-instance traffic ran on RDMA, regardless of their locations. Therefore, loopback traffic and external traffic coexisted on every NIC, thus creating a 2:1 congestion on PCIe lanes of the NIC. Since the NIC could not mark ECN, it could only throttle loopback traffic and external traffic through PCIe back pressure and PFC pause frames. To validate the above analysis, we disabled RDMA for loopback traffic on some servers, then these servers stopped sending PFC frames. We notice that recent work [61, 70] also found this problem.

9 Lessons and Open Problems

In this section, we summarize the lessons learned from our experience and discuss open problems for future exploration.

Failovers are very expensive for RDMA. While we have implemented failover solutions in both sU-RDMA and sK-RDMA as the last resort, we find that failovers are particularly expensive for RDMA, and should be avoided as much as possible. Cloud providers adopt RDMA to save CPU cores and then use freed CPU cores for other purposes. To move traffic away from RDMA, we need to allocate *extra* CPU cores to carry these traffic. This increases CPU utilization, and even runs out of CPU cores at high loads. Hence, it is risky to perform large-scale RDMA failovers, which we treat as serious incidents in Azure. Given the risk, only after all the tests have passed, we gradually increase the RDMA deployment scale. During the rollout, we continuously monitor network performance and immediately stop the rolltout once anomalies are detected. After unavoidable failovers, we should aggressively switch back to RDMA when possible.

Host network and physical network should be converged. In 8.3, we present a new type of slow receivers, which is essentially due to congestion inside the host. Recent work [24] also presents evidence and characterization of host congestion in production clusters. We believe this problem is just a tip of the iceberg, while many problematic behaviors between host network and physical network remain unexposed. In conventional wisdom, host network and physical network are separated entities and NIC is their border. If we look into the host, it is essentially a network connecting heterogeneous nodes (e.g., CPU, GPU, DPU) with proprietary high speed links (e.g., PCIe link and NVLink) and switches (e.g., PCIe switch and NVSwitch). Inter-host traffic can be treated as north-south traffic for the host. With the increase of the datacenter link capacity and wide adoptions of hardware offloading and device direct access technologies (e.g., GPUDirect RDMA), inter-host traffic tends to consume larger and more various resources inside the host, thus resulting in more complex interactions with intra-host traffic.

We believe that host network and physical network should be converged in the future. And we envision this converged network will be an important step towards the dis-aggregated cloud. We look forward to operating this converged network in similar ways as we manage physical network today.

Switch buffer is increasingly important and needs more innovations. The conventional wisdom [26] suggests that low latency datacenter congestion control [26, 71, 82, 112] can alleviate the need of large switch buffers as they can preserve short queues. However, we find a strong correlation between switch buffers and RDMA performance problems in production. Clusters with smaller switch buffers tend to have more performance problems. And many performance problems can be mitigated by just tuning switch buffer parameters without touching DCQCN. This is why we always tune switch buffers before touching DCQCN (§8.2). The importance of switch buffer lies in the prevalence of bursty traffic and short-lived congestion events in datacenters [108]. Conventional congestion control solutions are ill-suited for such scenarios given their reactive nature. Instead, switch buffer plays as the first resort to absorb bursts and provide fast responses.

With the increase in datacenter link speed, we believe that switch buffer is increasingly important, thus deserving more efforts and innovations. First, the buffer size per port per Gbps on pizza box switches keeps decreasing in recent years [31]. Some switch ASICs even split the packet memory into multiple partitions, thus reducing effective buffer resource. We encourage more efforts to put into the development ASICs with deeper packet buffers and more unified architectures. Second, today's commodity switch ASICs only provide buffer management mechanisms [40] designed decades ago, thus limiting the scope of solutions to handle congestion. Following the trend of programmable data plane [32], we envision that future switch ASICs would provide more programmability on buffer models and interfaces, thus enabling the implementation of more effective buffer management solutions [22].

Cloud needs unified behavior models and interfaces for network devices. The diversity in software and hardware brings significant challenges to network operation at cloud scale. Different NICs from the same vendor can even have different behaviors that cause interoperability problems, not to mention devices from different vendors. In spite of all the efforts we put into the unified switch software (§6) and NIC congestion control (§7.2), we still experienced problems due to diversity, e.g., unexpected interactions between PFC and MACsec (§8.3). We envision that more unified models and interfaces will emerge to simplify operations and accelerate innovations in the cloud. Some key areas include chassis switches, smart network appliances, and RDMA NICs. We notice that there have been some efforts on standardizing congestion control for different data paths [85] and APIs for heterogeneous smart appliances [16].

Testing new network devices is crucial and challenging. From the day one of this project, we have been making large investments in building various testing tools and running rigorous tests in both testbeds and test clusters. Despite the significant number of problems discovered during tests, we still found some problems during deployments (§8.3), mostly due to micro-behaviors and corner cases that were overlooked. Some burning questions are given as follows:

- How to precisely capture micro-behaviors of RDMA NIC implementations in various scenarios?
- Despite many endeavors to measure switches' micro-behaviors (§6.3), we still rely on domain knowledge to design test cases. How to systematically test the correctness and performance of a switch?

These questions motivate us to rethink challenges and re-

quirements of testing emerging network devices with more and more features. First, many features lack clear specifications, which is a prerequisite for systematic testing. Many seemingly simple features are actually entangled with complex interactions between software and hardware. We believe that unified behavior models and interfaces discussed above can help with this. Second, the test system should be able to interact with network devices at high speed, and precisely capture micro-behaviors. We believe programmable hardware can help on this [33, 37]. We note that there have been some recent progresses on testing RDMA NICs [69, 70] and programmable switches [37, 110].

10 Related Work

This paper focuses on RDMA for cloud storage. The literature of RDMA and storage systems is vast. Here we only discuss some closely related ideas.

Deployment experience of RDMA and storage networks: Before this project, we had deployed RDMA to support some Bing workloads and encountered many problems, such as PFC storms, PFC deadlocks, and slow receivers [50]. We learnt several lessons from this deployment. Gao et al. [46] summarized the experience of deploying intra-cluster RDMA to support storage backend traffic in Alibaba. Miao et al. [80] presented two generations of storage network stacks to carry Alibaba's storage frontend traffic: LUNA and SOLAR. LUNA is a high performance user-space TCP stack while SOLAR is a storage-oriented UDP stack implemented in proprietary DPU. Scalable Reliable Datagram (SRD) [96] is a cloud-optimized transport protocol implemented in AWS custom Nitro networking card, and used by HPC, ML, and storage applications [7]. In contrast, we use commodity hardware to enable intra-region RDMA to support both storage frontend and backend traffic.

Congestion control in datacenters: There is a large body of work on datacenter congestion control, including ECN-based [26, 27, 99, 112], delay-based [71, 72, 76, 82], INT-based [23, 75, 101], credit-based [34, 38, 45, 52, 55, 84, 86, 88] and packet scheduling [28, 30, 36, 49, 54]. Our work focuses on regional networks which have large RTT variations. We notice that some efforts [95, 107] target at similar scenarios.

Improve RDMA in datacenters: In addition to congestion control, there are many efforts to improve RDMA's reliability, security and performance in datacenters, such as deadlock mitigation [56,92,103], support of multi-path [77], resilience over lossy networks [78,83,102], security mechanisms [94,98,104], virtualization [53, 67, 89, 100], testing [69, 70], and performance isolation in multi-tenant environments [109]. Our work focuses on first party traffic in the trusted environment. Given the limited retransmission performance of our NICs, we enable RDMA over lossless networks (§2.4).

Accelerate storage systems using RDMA and other techniques: Many proposals [41, 62–66, 74, 93, 106, 111] leverage RDMA to accelerate storage systems or networked systems in general. Similar to some solutions [13, 47, 74, 90], our RDMA protocols (§4) provide socket-like interfaces to keep compatibility with legacy storage stack. In addition to RDMA, some recent proposals improve storage systems using new kernel designs [58, 59, 73] and SmartNIC [68, 81].

11 Conclusions and Future Work

In this paper, we summarize our experience in deploying intra-region RDMA to support storage workloads in Azure. The high complexity and heterogeneity of our infrastructure brings a series of new challenges. We have made several changes to our network infrastructure to address these challenges. Today, around 70% of traffic in Azure is RDMA and intra-region RDMA is supported in all Azure public regions. RDMA helps us achieve significant disk I/O performance improvements and CPU core savings.

In the future, we plan to further improve our storage systems through innovations on system architecture, hardware acceleration, and congestion control. We also plan to bring RDMA to more scenarios.

Acknowledgements

We thank our shepherd Marco Canini and the anonymous reviewers for their valuable feedback that significantly improved the final paper. Yuanwei Lu, Liang Yang and Danushka Menikkumbura also provided important feedback. Yibo Zhu made contributions to DCQCN and PFC deadlock avoidance at the early stage of this project. Ranysha Ware contributed to DCQCN tuning. Zhuolong Yu helped us measure RDMA's retransmission performance. This project represents the work of many engineers, product managers, researchers, data scientists, and leaders across Microsoft over many years, more than we can list here. We thank them all. Finally, we thank our partners: Arista Networks, Broadcom, Cisco, Dell, Keysight and NVIDIA for their technical contributions and support.

References

[1] Amazon ebs volume types. https://aws.amazon.com/ebs/volume-types/.

[2] Amazon web services region. https://aws.amazon.com/about-aws/global-infrastructure/regions_az/.

[3] Arista 7500r switch architecture ('a day in the life of a packet'). https://www.arista.com/assets/data/pdf/Whitepapers/Arista7500RSwitchArchitectureWP.pdf.

[4] Azure managed disk types. https://docs.microsoft.com/en-us/azure/virtual-machines/disks-types.

[5] Azure region. https://docs.microsoft.com/en-us/azure/availability-zones/az-overview.

[6] Cisco silicon one product family. https://www.cisco.com/c/dam/en/us/solutions/collateral/silicon-one/white-paper-sp-product-family.pdf.

[7] A decade of ever-increasing provisioned iops for amazon ebs. https://aws.amazon.com/blogs/aws/a-decade-of-ever-increasing-provisioned-iops-for-amazon-ebs/.

[8] Google cloud region. https://cloud.google.com/compute/docs/regions-zones.

[9] Keysight network test solutions. https://www.keysight.com/us/en/solutions/network-test.html.

[10] Packet testing framework (ptf). https://github.com/p4lang/ptf.

[11] Pfc watchdog in sonic. https://github.com/sonic-net/SONiC/wiki/PFC-Watchdog-Design.

[12] Priority flow control: Build reliable layer 2 infrastructure. https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/908/802.1q-Flow-Control-white_5F00_paper_5F00_c11_2D00_542809.pdf.

[13] rsocket(7) - linux man page. https://linux.die.net/man/7/rsocket.

[14] Smb direct. https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-direct.

[15] Software for open networking in the cloud (sonic). https://sonic-net.github.io/SONiC/.

[16] Sonic-dash - disaggregated api for sonic hosts. https://github.com/sonic-net/DASH.

[17] Sonic fast reboot. https://github.com/sonic-net/SONiC/blob/master/doc/fast-reboot/fastreboot.pdf.

[18] sonic-mgmt: Management and automation code used for sonic testbed deployment, tests and reporting. https://github.com/sonic-net/sonic-mgmt.

[19] Sonic warm reboot. https://github.com/sonic-net/SONiC/blob/master/doc/warm-reboot/SONiC_Warmboot.md.

[20] Switch abstraction interface (sai). https://github.com/opencomputeproject/SAI.

[21] Ieee standard for local and metropolitan area networks-media access control (mac) security. *IEEE Std 802.1AE-2018 (Revision of IEEE Std 802.1AE-2006)*, 2018.

[22] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: active buffer management in datacenters. In *SIGCOMM 2022*.

[23] Vamsi Addanki, Oliver Michel, and Stefan Schmid. Powertcp: Pushing the performance limits of datacenter networks. In *NSDI 2022*.

[24] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *HotNets 2022*.

[25] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008*.

[26] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *SIGCOMM 2010*.

[27] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI 2012*.

[28] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM 2013*.

[29] InfiniBand Trade Association. Supplement to infiniband architecture specification volume 1 release 1.2. 1 annex a17: Rocev2, 2014.

[30] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI 2015*.

[31] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. In *INFOCOM 2020*.

[32] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David

Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[33] Pietro Bressana, Noa Zilberman, and Robert Soulé. Finding hard-to-find data plane bugs with a pta. In *CoNEXT 2020*.

[34] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcpim: Near-optimal proactive datacenter transport. In *SIGCOMM 2022*.

[35] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP 2011*.

[36] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *SIGCOMM 2016*.

[37] Yanqing Chen, Bingchuan Tian, Chen Tian, Li Dai, Yu Zhou, Mengjing Ma, Ming Tang, Hao Zheng, Zhewen Yang, Guihai Chen, Dennis Cai, and Ennan Zhai. Norma: Towards practical network load testing. In *NSDI 2023*.

[38] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM 2017*.

[39] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *SIGCOMM 2018*.

[40] Abhijit K. Choudhury and Ellen L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking*, 1998.

[41] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI 2014*.

[42] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI 2018*.

[43] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1993.

[44] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma. In *ICDCS 2009*.

[45] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT 2015*.

[46] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *NSDI 2021*.

[47] Dror Goldenberg, Michael Kagan, Ran Ravid, and Michael S Tsirkin. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In *HOTI 2005*.

[48] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM 2009*.

[49] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *NSDI 2015*.

[50] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM 2016*.

[51] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM 2015*.

[52] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks

and stacks for low latency and high performance. In *SIGCOMM 2017*.

[53] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. Masq: Rdma for virtual private cloud. In *SIGCOMM 2020*.

[54] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM 2012*.

[55] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *SIGCOMM 2020*.

[56] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *CoNEXT 2017*.

[57] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *ATC 2012*.

[58] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. Tcp $\approx$ rdma: Cpu-efficient remote storage access with i10. In *NSDI 2020*.

[59] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *OSDI 2021*.

[60] IEEE. 802.11 qbb. priority based flow control. 2008.

[61] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *OSDI 2020*.

[62] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *NSDI 2019*.

[63] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance rdma systems. In *ATC 2016*.

[64] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *OSDI 2016*.

[65] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *SIGCOMM 2014*.

[66] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *SIGCOMM 2018*.

[67] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual rdma networking for containerized clouds. In *NSDI 2019*.

[68] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *SOSP 2021*.

[69] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, and Alvin R Lebeck Danyang Zhuo. Understanding rdma microarchitecture resources for performance isolation. In *NSDI 2023*.

[70] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in rdma subsystems. In *NSDI 2022*.

[71] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM 2020*.

[72] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *ATC 2015*.

[73] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs. In *ATC 2019*.

[74] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *SIGCOMM 2019*.

[75] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *SIGCOMM 2019*.

[76] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the transience-equilibrium nexus: A new approach to datacenter packet transport. In *NSDI 2021*.

[77] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for rdma in datacenters. In *NSDI 2018*.

[78] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *APNet 2017*.

[79] Matt Mathis, John Heffner, and Rajiv Raghunarayan. Tcp extended statistics mib (rfc 4898). Technical report, 2007.

[80] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in alibaba cloud. In *SIGCOMM 2022*.

[81] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *SIGCOMM 2021*.

[82] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM 2015*.

[83] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *SIGCOMM 2018*.

[84] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM 2018*.

[85] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *SIGCOMM 2018*.

[86] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *NSDI 2022*.

[87] Madhav Himanshubhai Pandya, Aaron William Ogus, Zhong Deng, and Weixiang Sun. Transport protocol and interface for efficient data transfer over rdma fabric, August 2 2022. US Patent 11,403,253.

[88] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Deverat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM 2014*.

[89] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R Gross. A hybrid i/o virtualization framework for rdma-capable network interfaces. *ACM SIGPLAN Notices*, 2015.

[90] Jim Pinkerton. Sockets direct protocol v1. 0 rdma consortium. 2003.

[91] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM 2022*.

[92] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. Gentle flow control: avoiding deadlock in lossless networks. In *SIGCOMM 2019*.

[93] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. Rdma is turing complete, we just did not know it yet! In *NSDI 2022*.

[94] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. Redmark: Bypassing rdma security mechanisms. In *USENIX Security 2021*.

[95] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *SIGCOMM 2020*.

[96] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 2020.

[97] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *NSDI 2019*.

[98] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. srdma: efficient nic-based authentication and encryption for remote direct memory access. In *ATC 2020*.

[99] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM 2012*.

[100] Dongyang Wang, Binzhang Fu, Gang Lu, Kun Tan, and Bei Hua. vsocket: virtual socket interface for rdma in public clouds. In *VEE 2019*.

[101] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, robust, and practical datacenter cc via deployable int. In *NSDI 2023*.

[102] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. Srnic: A scalable architecture for rdma nics. In *NSDI 2023*.

[103] Xinyu Crystal Wu and TS Eugene Ng. Detecting and resolving pfc deadlocks with itsy entirely in the data plane. In *INFOCOM 2022*.

[104] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable network support for secure rdma systems. In *USENIX Security 2022*.

[105] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *SYSTOR 2015*.

[106] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *FAST 2019*.

[107] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. Congestion control for cross-datacenter networks. In *ICNP 2019*.

[108] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *IMC 2017*.

[109] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *NSDI 2022*.

[110] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. Meissa: scalable network testing for programmable data planes. In *SIGCOMM 2022*.

[111] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. Octopus+: An rdma-enabled distributed persistent memory file system. *ACM Transactions on Storage*, 2021.

[112] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM 2015*.

[113] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *CoNEXT 2016*.

[114] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM 2015*.

A SONiC buffer analysis

```
"BUFFER_POOL": {
    "ingress_pool": {
        "size": "18000000",
        "type": "ingress",
        "mode": "dynamic",
        "xoff": "6000000"
    },
    "egress_lossy_pool": {
        "size": "14000000",
        "type": "egress",
        "mode": "dynamic"
    },
    "egress_lossless_pool": {
        "size": "24000000",
        "type": "egress",
        "mode": "static"
    }
}

"BUFFER_PROFILE": {
    "ingress_lossless_profile" : {
        "pool": "[BUFFER_POOL|ingress_pool]",
        "size": "1248",
        "dynamic_th": "-3",
        "xoff": "96928",
        "xon"    "1248",
        "xon_offset"    "2496"
    },
    "ingress_lossy_profile": {
        "pool": "[BUFFER_POOL|ingress_pool]",
        "size": "0",
        "static_th":"24000000"
    },
    "egress_lossless_profile": {
        "pool": "[BUFFER_POOL|egress_lossless_pool]",
        "size": "0",
        "static_th": "24000000"
    },
    "egress_lossy_profile": {
        "pool": "[BUFFER_POOL|egress_lossy_pool]",
        "size": "1664",
        "dynamic_th": "-1"
    }
}
```

Listing 1: SONiC Buffer Configuration Example

Listing 1 gives a buffer configuration example of a SONiC pizza box switch with 24 MB packet buffer. `ingress_pool` has 18 MB (`size`) shared buffer for all the ingress queues, and 6 MB (`xoff`) PFC headroom buffer exclusively for ingress lossless queues in the paused state. `egress_lossy_pool` and `egress_lossless_pool` have 14 MB and 24 MB shared buffer, respectively. It is worthwhile to notice that the sum of pool sizes can be larger than the physical buffer limit, as they are only virtual counters for admission control purposes.

Lossless packets are mapped to both ingress lossless queues (`ingress_lossless_profile`) and egress lossless queues (`egress_lossless_profile`). We use Dynamic Threshold (DT) algorithm [40] to manage the buffer occupancy of the ingress lossless queue in the 18 MB shared buffer space of `ingress_pool`. DT algorithm is controlled by a parameter called α, which is 1/8 ($2^{\texttt{dynamic_th}}$) in Listing 1. Once the ingress lossless queue hits the dynamic threshold ($\alpha\times$ remaining buffer), it will enter the paused state (send PFC pause frames) and start to use PFC headroom. All the ingress lossless queues in the paused state share a 6 MB PFC headroom pool (`xoff` of `ingress_pool`). Each ingress lossless queue can use up to 96928 bytes buffer (`xoff` of `ingress_lossless_profile`) in the PFC headroom pool. We bypass the egress admission control for lossless traffic by setting the static threshold of the egress lossless queue (`static_th` of `egress_lossless_profile`) to 24 MB, which equals to the switch buffer size.

In contrast, we only want to apply egress admission control for lossy traffic. To bypass ingress admission control for lossy traffic, we configure a sky-high static threshold 24 MB (`static_th` of `ingress_lossy_profile`) for each ingress lossy queue. Since lossy traffic can only use 18 MB shared buffer space of `ingress_pool`, the size of `egress_lossy_pool` should be no larger than 18 MB (`size` of `ingress_pool`). In Listing 1, the `size` of `egress_lossy_pool` is 14 MB. This guarantees that ingress lossless queues can exclusively use 4 MB shared buffer (`size` of `ingress_pool` - `size` of `egress_lossy_pool`) in `ingress_pool` before entering the paused state. We use DT algorithm to manage the egress lossy queue length and set α to 1/2 ($2^{\texttt{dynamic_th}}$). Once the egress lossy queue hits the dynamic threshold, its arriving packets will be dropped.

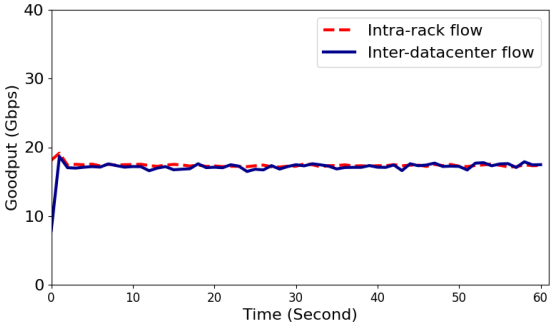


Figure 12: Goodput of two flows with different RTTs.

B DCQCN experiment results

We conduct an experiment in our lab testbed to demonstrate the RTT fairness of DCQCN. Our lab testbed uses a four-tier Clos topology like Figure 2. We use 80 km cables to interconnect T2 switches to a RH switch to emulate a region.

In this experiment, we use two hosts A and B as senders and a host C as the receiver. Each host is equipped with a Gen1 40 Gbps NIC. Host A and C are located within the same rack with $\sim$2 μs base RTT. In contrast, B is in another datacenter. The base RTT across the RH switch is $\sim$1.77 ms. On each sender, we use `ndperf` to create a QP with the receiver and keep posting 64 KB `Write` messages. Each QP can keep up to 160 in-flight `Write` messages, resulting in around 10 MB in-flight data, which is enough to saturate the large inter-datacenter pipe (40 Gbps $\times$ 1.77 ms = 8.85 MB). We set RED/ECN marking parameters K_{min}, K_{max} and P_{max} to 1 MB, 2 MB and 5%, respectively.

As shown in Figure 12, two DCQCN flows achieve similar goodput regardless of their RTTs. A flow can achieve around 17 Gbps goodput, which is close to half of the line rate. We also keep polling queue watermark counters at the congested switch and find queue watermarks oscillate around 1.36 MB, which is smaller than K_{max}. This experiment demonstrates that DCQCN does not suffer from RTT unfairness.

Transparent GPU Sharing in Container Clouds for Deep Learning Workloads

Bingyang Wu* Zili Zhang* Zhihao Bai† Xuanzhe Liu* Xin Jin*

*Peking University †Johns Hopkins University

Abstract

Containers are widely used for resource management in datacenters. A common practice to support deep learning (DL) training in container clouds is to statically bind GPUs to containers in entirety. Due to the diverse resource demands of DL jobs in production, a significant number of GPUs are underutilized. As a result, GPU clusters have low GPU utilization, which leads to a long job completion time because of queueing.

We present TGS (Transparent GPU Sharing), a system that provides transparent GPU sharing to DL training in container clouds. In stark contrast to recent application-layer solutions for GPU sharing, TGS operates at the OS layer beneath containers. Transparency allows users to use any software to develop models and run jobs in their containers. TGS leverages adaptive rate control and transparent unified memory to simultaneously achieve high GPU utilization and performance isolation. It ensures that production jobs are not greatly affected by opportunistic jobs on shared GPUs. We have built TGS and integrated it with Docker and Kubernetes. Experiments show that (i) TGS has little impact on the throughput of production jobs; (ii) TGS provides similar throughput for opportunistic jobs as the state-of-the-art application-layer solution AntMan, and improves their throughput by up to 15× compared to the existing OS-layer solution MPS.

1 Introduction

Containers [1–3] are widely used for resource management in datacenters. Containers provide lightweight virtualization, and can significantly reduce the complexity and cost of deployments and managements in datacenters.

Deep learning (DL) is an important workload in datacenters. With recent advancements in deep neural networks (DNNs) [4] and the burst of big data space, DL models have been increasingly integrated into applications and online services. Large enterprises build multi-tenant GPU clusters that are shared by many teams to develop and train DL models.

A common practice to support DL training in container clouds is to statically bind complete GPUs to containers. When a GPU is allocated to a container, the container has exclusive access to the GPU, which provides performance isolation for production jobs. But it means that other containers on the same machine cannot use the GPU when the GPU is under-utilized or is even completely idle.

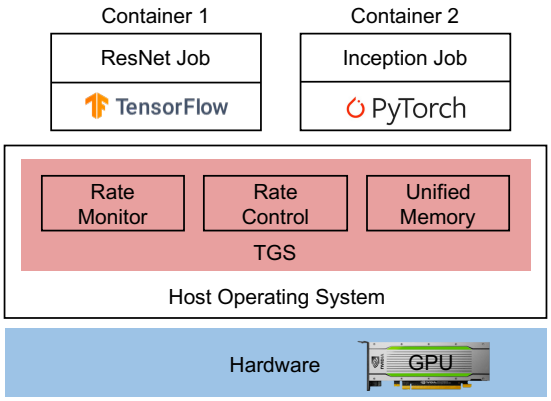


Figure 1: TGS architecture.

The major limitation of this approach is *low resource utilization*. A recent study on a production GPU cluster by Microsoft shows that the mean GPU utilization is only 52% [5]. Another measurement on a production GPU cluster at Alibaba shows even lower GPU utilization—the median GPU utilization is no more than 10% [6]. However, due to exclusive GPU allocation, incoming jobs have to wait in the queue to be scheduled even when many GPUs are not fully utilized. This causes a long job completion time for subsequent jobs.

This is a known problem in production GPU clusters [5, 6]. The problem can be addressed by GPU sharing to increase GPU utilization. In production environments [6–8], DNN training jobs are typically classified into two classes: *production* jobs, which must run without much great performance degradation caused by other jobs, and *opportunistic jobs*, which utilize spare resources. It is natural to share GPUs between the two classes of jobs to improve GPU utilization. Yet, it is critical for production environments to ensure that the impact of GPU sharing on production jobs is minimized.

GPU sharing solutions can be realized at either the application layer or the OS layer. AntMan [6] is a state-of-the-art application-layer solution. While AntMan can provide high GPU utilization and performance isolation, it modifies DL frameworks non-trivially and restricts users to use particular versions of given frameworks. NVIDIA Multiple Process

Sharing (MPS) [9] is an OS-layer solution. MPS requires application knowledge to set resource limits for performance isolation and does not support GPU sharing under GPU memory oversubscription. It merges several processes into a single CUDA context, leading to fate sharing between jobs.

We present TGS, a system that provides transparent GPU sharing to DL training in container clouds. Unlike application-layer solutions, TGS works at the OS layer and realizes the benefits of application-layer solutions at the OS layer without the limitations of existing OS-layer solutions. Transparency allows users to choose any version of any DL framework (either TensorFlow, PyTorch or a custom framework) to develop models and run jobs in containers.

The core of TGS is a lightweight indirection layer between containers and GPUs. It intercepts the system calls from containers to GPUs and regulates the GPU resource usage for concurrent jobs. TGS enables GPU sharing between the production job and the opportunistic job, but largely isolates the production job from contention.

There are two primary technical challenges in realizing an OS-layer GPU sharing solution with performance isolation. The first challenge is to share GPU compute resources between containers adaptively without application knowledge. Inaccurately setting resource limits for each container would either degrade job performance or leave resources unused. MPS and MIG require application knowledge to manually set resource limits. TGS applies an adaptive rate control approach to address this challenge without application knowledge. It monitors the performance of production jobs at runtime, and adaptively updates the resource allocation to opportunistic jobs. The control loop automatically converges to the point that opportunistic jobs utilize as many resources as possible without much affecting production jobs.

The second challenge is to enable transparent GPU memory oversubscription. GPUs have their own memory to keep the application state. MPS fails when the total GPU memory required by containers exceeds the GPU memory size. AntMan uses a custom memory management component in DL frameworks to manage memory swapping between GPU memory and host memory at the application layer. We design a transparent unified memory mechanism based on CUDA unified memory to enable unified memory at the OS layer, obviating the need to explicitly modify applications. This mechanism manages memory swapping underneath when the GPU memory is oversubscribed. TGS leverages placement preferences to ensure that GPU memory is prioritized for production jobs to protect their performance.

In summary, we make the following contributions.

- We propose TGS, a system that provides transparent GPU sharing for DL training in container clouds.
- We design adaptive rate control and transparent unified memory mechanisms to simultaneously achieve high GPU utilization and performance isolation.
- We implement TGS and integrate it with Docker and Kubernetes. Experiments show that (i) TGS has little impact on the throughput of production jobs; (ii) TGS provides similar throughput for opportunistic jobs as state-of-the-art application-layer solution AntMan and improves their throughput by up to 15× compared to existing OS-layer solution MPS.

2 Background and Motivation

In this section, we first introduce containers, deep learning training, and the current practice to support deep learning training in container clouds. Then, we show the limitations of existing solutions to motivate TGS.

2.1 Container Clouds

Containers [1–3] (e.g., Docker) are used widely to manage resources and deploy workloads in datacenters, and provide *portability* and *isolation*. A container is a standalone software package including everything needed to run an application. A containerized application can run across various environments without any modifications. Such portability enables developers to use the tools and application stacks of their choice to develop and run their applications, without worrying about deployment environments. Applications in different containers are isolated by using independent namespaces.

Containers are lightweight, compared with virtual machines. Virtual machines use a guest OS, but containers use the host OS kernel. Thus, applications can achieve bare metal performance when running in containers. Cloud operators use a container orchestration platform [10, 11] to provision, manage and update containers on many machines in a datacenter.

2.2 DL Training Workloads

DL training uses a dataset to train a DNN model. A training job contains many iterations. Each iteration uses a batch of samples from the dataset to train the DNN model. An iteration includes a forward pass and a backward pass. The forward pass uses the DNN model to compute the labels of the samples in the batch. A loss is computed based on the output labels and the actual labels using a loss function. The backward pass propagates the loss from the last layer to the first layer of the DNN model and computes the gradients for each weight. The DNN model is updated based on the gradients using an optimizer. DL training is compute-intensive, so GPUs are typically used. However, widely-adopted exclusive GPU allocation leads to low GPU utilization in production, as reported by Microsoft [5] and Alibaba [6].

2.3 Limitations of Existing Solutions

A natural way to increase GPU utilization is GPU sharing. If a single container cannot utilize all the GPU resources, a GPU can be shared by multiple containers to increase GPU utilization. However, containers on a shared GPU will compete for compute and memory resources of the GPU, and the interference can slow down the jobs.

	AntMan [6]	Salus [12]	PipeSwitch [13]	MPS [9]	MIG [14]	TGS
Transparency				✓	✓	✓
High GPU utilization	✓	✓				✓
Performance isolation	✓	✓	✓	✓	✓	✓
Fault isolation	✓		✓		✓	✓

Table 1: Comparison between TGS and existing GPU sharing solutions.

GPU sharing can be done either at the application layer or the OS layer. The primary drawback of application-layer solutions [6, 12, 13] is that they are not *transparent* to users, i.e., they require significant modifications to DL frameworks. Users are restricted to use the set of supported versions of given frameworks and have to wait for the integration if a newer version of a particular DL framework comes. This approach loses the advantage of allowing users to use any tools to develop and run applications in containers.

NVIDIA MPS [9] is an OS-layer solution for GPU sharing. It requires application knowledge to properly set the resource limit for each process to ensure performance isolation. More importantly, MPS requires the total GPU memory of the processes to fit within the GPU memory capacity and relies on applications to handle memory swapping between GPU memory and host memory. Another limitation of MPS is that it does not provide *fault isolation*. MPS merges the CUDA contexts of multiple processes into a single CUDA context to share the GPU. When a process fails, it leaves the MPS server and other processes in an undefined state and may result in process hangs, corruptions, or failures.

NVIDIA Multi-Instance GPU (MIG) [14, 15] is another OS-layer solution. MIG requires GPU hardware support and is currently only available on three high-end GPUs, i.e., NVIDIA A100, NVIDIA A30, and NVIDIA H100. MIG cannot *arbitrarily* partition a GPU based on application needs; it only supports GPU partitioning for a given set of configurations. For example, an NVIDIA A100 GPU can be partitioned into *GPU instances* with separate compute and memory resources for different DL training jobs, but MIG only provides seven fixed configurations for each GPU instance and each GPU instance cannot use more than 4/7 of the GPU compute resources or half of the GPU memory resources. Furthermore, it cannot *dynamically* change GPU resources owned by GPU instances if there are running jobs on the GPU even if the GPU usage of a container changes. Reconfiguration of MIG can only happen when the GPU is idle. MIG does not support memory oversubscription.

3 TGS Overview

TGS is a GPU sharing system for deep learning training in container clouds that is designed to meet the following goals. Table 1 compares TGS with existing GPU sharing solutions regarding these four goals.

- **Transparency.** The system should be transparent to applications so that users can use any software to develop and train DNN models in containers.
- **High GPU utilization.** The system should achieve high GPU utilization for both compute and memory resources.
- **Performance isolation.** The system should provide performance isolation for DL jobs. Production jobs should not be significantly affected by opportunistic jobs.
- **Fault isolation.** Application faults should be isolated by containers. The fault of an application in one container should not crash applications in other containers.

Architecture. Figure 1 shows that TGS is an OS-layer approach: it sits between containers and GPUs. Containers and applications are unaware of TGS. Users can use any custom framework to develop and train DNN models. A GPU is exposed as a regular GPU to the containers. The processes in the containers issue GPU kernels, i.e. functions executed on the GPU, to the GPU as they do with a dedicated GPU. TGS uses a lightweight indirection layer to share the GPU between workloads of several containers. The indirection layer intercepts the GPU kernels from containers and regulates these GPU kernels to control the resource usage of each container.

Key ideas. TGS leverages an adaptive rate control mechanism and a transparent unified memory mechanism to tackle two challenges in providing transparent GPU sharing at OS layer. The first challenge is to adaptively share GPU compute resources between containers without application knowledge. To address this challenge, the rate monitor of TGS monitors the performance of each container, and provides the number of *CUDA blocks* (a basic scheduling and execution unit on the GPU) as a real-time signal for the control loop. Based on the signal, the rate control of TGS adaptively controls the rate of sending GPU kernels to the GPU for each container. The control loop automatically converges to the point that opportunistic jobs utilize as many remaining resources as possible to achieve high GPU utilization without greatly affecting the performance of production jobs.

The second challenge is to enable transparent GPU memory oversubscription. AntMan [6] modifies DL frameworks to swap GPU memory when GPU memory is oversubscribed. OS-layer solution MPS does not support GPU memory oversubscription, and relies on applications to handle memory swapping. These approaches are not transparent. To address this challenge, TGS exploits CUDA unified memory [16] which unifies GPU memory and host memory in a single

memory space. TGS intercepts and redirects GPU memory allocation calls from containers to the CUDA unified memory space. When the GPU memory is oversubscribed, TGS can automatically evict some data of opportunistic jobs to the host memory, and change the mapping of the corresponding virtual addresses to the new data locations in the host memory. The entire process is transparent to applications. To ensure performance isolation, TGS uses memory placement preferences to prioritize allocating GPU memory for production jobs over opportunistic jobs.

The design of TGS has two other benefits. First, the architecture is *lightweight*. TGS has low overhead and conforms with the principle of containers. Second, TGS provides the same *fault isolation* property as regular containers. The containers in TGS use separate GPU contexts, as opposed to MPS which merges the CUDA contexts of the containers into one. Therefore, an application fault in one container does not affect or terminate other containers.

4 TGS Design

In this section, we present the design of TGS. We first describe the adaptive rate control mechanism to share GPU compute resources. Then we describe the unified memory mechanism to share GPU memory resources.

4.1 Sharing GPU Compute Resources

Application code is encapsulated into functions to be executed on a GPU, which are known as GPU kernels. GPU kernels are highly optimized based on the particular architecture and execution model of the GPU. A small DNN training job may not use all the compute resources of a GPU. In this case, the GPU has low utilization if it is exclusively allocated to the container of the job. TGS improves GPU utilization by GPU sharing. In TGS, a GPU can be exposed to and shared by multiple containers to increase GPU utilization.

TGS ensures the performance of production jobs is not greatly affected by opportunistic jobs. Opportunistic jobs use no more than the resources left by production jobs. To achieve this, we need to solve two problems. First, we need to estimate how many resources are left by production jobs. Second, we need to control opportunistic jobs to use no more than the remaining resources.

Strawman solution: priority scheduling. A strawman solution is priority scheduling. It intercepts the GPU kernels from containers and puts them into a production queue and an opportunistic queue based on the priority of the job. The kernels in the opportunistic queue are only scheduled to the GPU when the production queue is empty. In this solution, whether there are remaining resources is estimated by checking whether the production queue is empty, and controlling the resource usage of opportunistic jobs is achieved by prioritizing the scheduling of the kernels in the production queue. This is a canonical solution to performance isolation and high utilization, and has been widely used in computer systems.

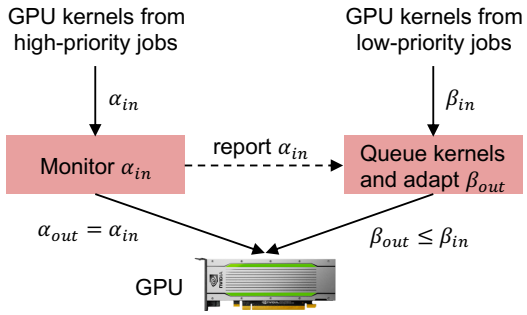


Figure 2: Adaptive rate control.

However, this solution is not suitable for GPU sharing. An empty production queue for GPU jobs does not mean production jobs are not using the GPU. A GPU kernel is an optimized GPU function that runs for some time. The GPU kernels scheduled in the past may still be running on the GPU, while the production queue is empty. Similarly, an empty queue also cannot tell how many resources on left on the GPU. Therefore, if the kernels in the opportunistic queue are sent to the GPU and the production jobs are using most of the GPU resources, then the GPU kernels from both jobs would contend with each other, which incurs large overhead for production jobs. Keeping track of GPU kernels running on the GPU is also not feasible, because the state of the GPU is not fully visible.

It may be possible to implement a priority scheduler into the GPU device driver, so that the scheduler can have full visibility of the resource usage and can perform fine-grained control. This solution is not general. It is tightly tied to the low-level GPU specifics and requires deep integration with each type of GPU based on their architecture and execution model. Some GPUs are blackboxes and do not expose such control to the OS.

Our solution: adaptive rate control. TGS uses an adaptive rate control approach (Figure 2). The main idea is to carefully control the dequeuing rate of the kernels in the opportunistic queue based on the kernel arrival rate, so that opportunistic jobs can use up the remaining compute resources without greatly affecting the production job. This is a general OS-layer approach: it is decoupled from low-level GPU specifics and does not require access to GPU internal control.

This approach requires a feedback signal to tell the control loop whether the dequeuing rate of the opportunistic queue can be increased to use more resources or should be decreased to avoid degrading production jobs. Ideally, we want to use the application performance, i.e., the training throughput for DL training workloads, as the feedback signal, because this is the metric we ultimately care about. However, we cannot directly obtain the training throughput, because this requires application knowledge, and we aim to design an OS-layer solution that is transparent to applications.

One choice of the signal is GPU utilization, i.e., increase the rate if the GPU utilization is below 100%. While this choice

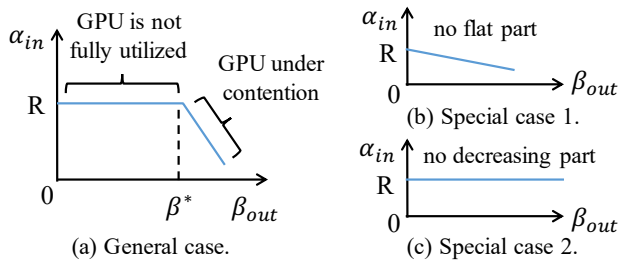


Figure 3: Relationship between the rates of production and opportunistic jobs.

seems natural, it has two drawbacks. First, the definition of GPU utilization is hardware-specific and is often vague [17]. Today's GPUs contain different types of compute units on a single chip, e.g., Tensor cores and CUDA cores for different data types on NVIDIA GPUs. GPU utilization reported by GPU drivers (if supported) often lacks a precise definition. Even if it does (e.g., the percentage of stream processors that are used), it is unclear what a single utilization value actually means for a GPU with several types of compute units. Second, GPU utilization is only loosely coupled with the application performance. Even when the reported GPU utilization is below 100%, it does not mean we can increase the dequeuing rate of the opportunistic queue without slowing down production jobs. For example, a production job and an opportunistic job may compete for the same type of compute units that are already used up by the production job alone, though there are other types of compute units that are idle; two jobs may also compete for other resources than the one captured by GPU utilization.

In TGS, we use the kernel arrival rate of production jobs (i.e., the rate that TGS receives kernels from the containers) as the feedback signal. A DL training job constructs a compute graph based on the DNN model for its training process. It uses the compute graph to generate and send kernels to the GPU to perform training. The compute graph captures the dependencies between the kernels. The kernel arrival rate directly corresponds to the training throughput. If the training is slowed down, the kernels are finished slower, the dependencies are satisfied slower, and the kernel arrival rate drops. Therefore, TGS uses a rate monitoring module to monitor the kernel arrival rate of production jobs, and uses it as the feedback signal to control the kernel dequeuing rate of opportunistic jobs. Note that any contention between production jobs and opportunistic jobs can be captured by this kernel arrival rate, including GPU cache contention, CPU contention and network contention. Some of them are beyond what a GPU hardware design can control, and TGS uses rate control as a knob to control all of them. Since there can be a small variance in the kernel arrival rate, TGS uses a moving average to smooth the estimation of the kernel arrival rate. For the kernels from production jobs, TGS only performs a simple counting operation to estimate the kernel arrival rate. It does not queue the kernels and directly passes them to the GPU, to minimize the impact on the performance of production jobs.

Rate adaptation algorithm. The rate adaptation algorithm controls the kernel dequeuing rate of the opportunistic queue, so that the kernel arrival rate of production jobs is not greatly affected and the kernel dequeuing rate of opportunistic jobs is maximized. Formally, let α_{in} and α_{out} be the rates that the kernels of production jobs arrive at and departure from TGS respectively, and β_{in} and β_{out} be those of the opportunistic jobs. TGS only monitors, but does not limit the rate of production jobs. So $\alpha_{in} = \alpha_{out}$. Let the kernel arrival rate of production jobs when the GPU is not shared be R. The rate control algorithm is to maximize β_{out} so that $\alpha_{in} = R$. In the formulation, β_{out} is the variable controlled by the algorithm and α_{in} is dependent on β_{out}. Let f be the function that captures the relationship between α_{in} and β_{out}, i.e., $\alpha_{in} = f(\beta_{out})$. Then the algorithm has to solve the following optimization problem.

$$max \quad \beta_{out} \tag{1}$$
$$s.t. \quad \alpha_{in} = f(\beta_{out}) \geq R \tag{2}$$
$$\beta_{out} \geq 0 \tag{3}$$

The exact shape of $f(\beta_{out})$ is unknown, but we know its rough shape by the nature of the problem. Specifically, $f(\beta_{out})$ is flat and is equal to R when β_{out} is small, and is monotonically decreasing when β_{out} is large, as illustrated in Figure 3(a). The intuition is that when β_{out} is small, the GPU is not fully utilized and executing the kernels of opportunistic jobs does not affect the performance of production jobs, resulting in a flat line; after the tipping point β^*, opportunistic jobs start to compete with production jobs for GPU resources, causing the performance of production jobs to drop. Note that the monotonically decreasing part is not necessarily linear; Figure 3(a) illustrates the general trend that α_{in} decreases when β_{out} increases. The goal of the algorithm is to find the tipping point β^* from which $f(\beta_{out})$ starts to decrease.

Figure 3(a) is the general case. There are two special cases. Figure 3(b) is the special case where the GPU is already fully utilized by production jobs, so that even executing a small number of kernels for opportunistic jobs would degrade the performance of production jobs. In this case, the line does not have a flat part. Figure 3(c) is the special case where the demand of opportunistic jobs is very small, so that even when the dequeuing rate is not limited, the performance of production jobs is not affected. In this case, the line does not have a monotonically decreasing part.

To approximate the optimal β_{out}, we use the canonical additive increase multiplicative decrease (AIMD) method to control the rate β_{out}, as shown in Algorithm 1. Specifically, TGS first measures the rate R of a production job on a GPU before it adds an opportunistic job to the GPU for sharing (line $1-3$). After the opportunistic job is added, TGS additively increases β_{out}, if α_{in} is greater than or equal to R (line $24-$

25), or multiplicatively decreases β_{out}, if α_{in} is below R (line $29-30$). AIMD ensures that β_{out} can approximately converge to the tipping point β^*. To accelerate convergence, a slow start phase is adopted (line $17-22$). Experiments in §6 shows that the convergence is fast. When the production job changes its resource usage pattern, TGS detects that the variance of R is beyond a threshold. In this case, the rate control module suspends the opportunistic job and measures new R (line $26-28$). When R becomes stable, the rate control module uses AIMD to adjust β_{out} to the tipping point. We have the following theorem to ensure the convergence of the adaptive rate control algorithm at most cases.

Theorem 1 *Assuming DL jobs are stable during the profiling phase and the convergence phase, the adaptive rate control algorithm converges in $O(B\log B)$ function calls, where B is the throughput limit of jobs in the GPU.*

The proof of the theorem is in Appendix A. The proof is based on the stability of the deep learning training workload. For readers familiar with congestion control in computer networking, our problem resembles the bandwidth allocation problem when multiple flows compete for the bandwidth resources of a shared link. In bandwidth allocation, each flow uses a congestion control algorithm to control its own rate, and after the system converges, each flow gets a fair share of the link bandwidth. Our problem is subtly different from bandwidth allocation in that we do not limit the rate of production jobs, and only control the rate of opportunistic jobs to ensure that the performance of production jobs is not greatly affected by resource sharing.

4.2 Sharing GPU Memory Resources

GPUs have GPU memory that is separated from the host memory. The memory size in modern GPUs ranges from a few GB to tens of GB. GPU memory stores the state and data needed by applications to perform their computation on the GPU. The compute units in the GPU can access the GPU memory much faster than the host memory. The GPU device driver exposes the GPU memory to users with an API, which is similar to the memory management API for the host memory. Users use the API to allocate and manage GPU memory for their GPU programs, e.g., `cudaMalloc` for GPU memory allocation on NVIDIA GPUs. Similar to GPU compute resources, the GPU memory can be shared by multiple containers when a single container cannot utilize all the GPU memory resources.

Strawman solution: pass-through allocation. A strawman solution is to directly pass the GPU memory allocation calls from containers to the GPU. In this way, the GPU memory is fully utilized as long as there is enough demand from containers. The major limitation of this solution is that it has large overhead for production jobs. In this solution, when production jobs do not use all the GPU memory, opportunistic jobs can obtain the remaining memory. Later, if the production job wants to allocate more GPU memory, they would not be able to do so because the remaining memory has been allocated to opportunistic jobs. Without sufficient GPU memory, production jobs may run at a lower speed, or even fail, which violates fault isolation.

Algorithm 1 Adaptive Rate Control Algorithm

```
1:  procedure INIT
2:      R = measure_high_prio_job_rate()
3:      β_out = 0
4:      state = SLOW_START
5:
6:  procedure UPDATE_HIGH_RATE
7:      R_avg = avg(high_rate_window)
8:      dR = |R − R_avg|/R
9:      if dR < R_threshold then
10:         R = max(R, R_avg)
11:     else
12:         R = measure_high_prio_job_rate()
13:
14: procedure UPDATE_LOW_RATE_LIMIT
15:     dα = |R − α_in|/R
16:     switch state do
17:         case SLOW_START :
18:             if dα < threshold_slow_start then
19:                 β_out *= δ_SS
20:             else
21:                 β_out /= δ_SS
22:                 state = CA
23:         case CA :
24:             if dα < threshold_1 then
25:                 β_out += δ_AI
26:             else if dα > threshold_2 then
27:                 β_out = 0
28:                 state = SLOW_START
29:             else
30:                 β_out *= δ_MD
```

Another limitation of this solution is that it does not consider the characteristics of DL frameworks. When starting a job, some DL frameworks (e.g., TensorFlow) claim all the available GPU memory even if the training job does not request that much memory. These DL frameworks typically have a memory pool that caches all the allocated memory, and give the memory to the training job on demand. They do not free and return the allocated memory back to the GPU when some memory is not used. This is an optimization in these DL frameworks to avoid the overhead of frequently calling GPU memory to allocate and release during a job.

This optimization introduces challenges to sharing the GPU memory. Application-layer solutions like AntMan [6] can directly modify DL frameworks to obtain the memory usage of training jobs and disable unnecessary memory caching to return unused GPU memory back to the GPU. However, to design a transparent OS-layer solution, modifications on DL frameworks or applications are not allowed.

Our solution: unified GPU and host memory. Modern GPUs provide a feature called *unified memory* which unifies GPU memory and host memory in a single address space. Unified memory is traditionally used by applications to simplify GPU memory management. TGS applies CUDA unified

memory [16] in a novel way: it uses CUDA unified memory allocation as an *indirection* of GPU memory allocation, in order to achieve transparency and performance isolation for GPU memory sharing. Specifically, TGS exposes CUDA unified memory as pseudo GPU memory to containers. When a container issues a GPU memory allocation call, whether the call is for regular GPU memory or CUDA unified memory, TGS intercepts this call and allocates the memory requested by the call in the CUDA unified memory space. When production jobs do not use up the GPU memory, opportunistic jobs can obtain the remaining GPU memory.

Pseudo GPU memory refers to that the allocated memory appears to be normal GPU memory to containers and applications, while it can actually come from either GPU memory or host memory depending on availability. Note that we do not change the virtual memory system. Pseudo memory is still virtual memory, and applications use virtual memory addresses to access allocated pseudo memory. A GPU/host virtual memory address is translated to a GPU/host physical memory address by the GPU/host memory management unit.

The transparent unified memory in TGS is different from the original CUDA unified memory in two aspects, which are (i) performance isolation and (ii) transparent oversubscription of GPU memory. To provide performance isolation, TGS uses placement preferences in CUDA unified memory to prioritize the allocation of GPU memory to production jobs. When the GPU memory is not full, the memory allocation requests from any job get the GPU memory. When the GPU memory is full, TGS tries to place the blocks of production jobs in the GPU memory, and evict the blocks of opportunistic jobs to the host if necessary. This is transparent to the containers, as the containers still use the same virtual memory addresses to access their allocated memory space. The virtual memory addresses are translated to physical memory addresses at different locations. This mechanism also does not introduce additional out-of-memory (OOM) faults, because in the view of DL training jobs, the GPU memory capacity is the same as the size of the original GPU memory.

The transparent unified memory in TGS also addresses the issue of overclaiming the GPU memory in existing DL frameworks, without modifications to DL frameworks. When the DL framework claims all the available GPU memory, TGS allocates the requested amount of memory from the CUDA unified memory space. The actually used memory would trigger GPU page faults and be swapped to the GPU memory when it is used for the first time, and then would reside in the GPU memory. Consequently, only the portion actively used by the training job is in the GPU memory; the remaining portion is in the host memory. This allows opportunistic jobs to efficiently share the GPU memory.

5 Implementation

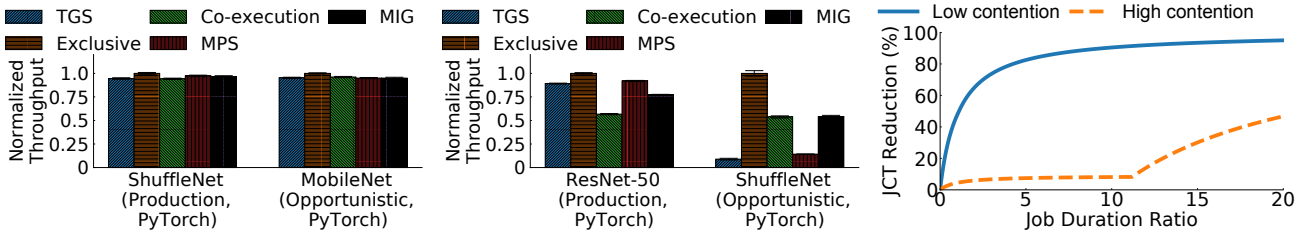
We have implemented a system prototype for TGS with ∼3000 lines of code in C++ and Python, and integrated it with Docker and Kubernetes. A coordinator process takes charge of resource management and leverages the indirection layer of TGS to enable GPU sharing between containers. Specifically, the adaptive rate control and the transparent unified memory provided by TGS are used for GPU sharing. The code of TGS is open-source and is publicly available at https://github.com/pkusys/TGS.

Adaptive rate control. TGS intercepts CUDA driver API calls related to CUDA kernel launch from containers for rate monitoring and rate control. Because CUDA kernel launch may be evoked by multiple threads in the container, TGS uses a global counter to record the number of CUDA blocks launched in a given time period. A CUDA block is a group of threads that must execute in the same SM (Streaming Multiprocessor) and different CUDA blocks can run independently in parallel. As the number of a CUDA block that a kernel contains is specified in the CUDA driver API call, the number of pending CUDA blocks can be treated as a real-time signal to estimate the performance of production jobs. For a production container, a standalone thread serves as the rate monitor, which reads this counter of the TGS periodically and sends the value to the rate-control component of the opportunistic container on the same GPU. For an opportunistic container, a rate control thread is created when the CUDA driver starts to work. The rate control thread adjusts the rate limit of the opportunistic container according to the received statistics. To keep the kernel launch rate of the opportunistic container at a desirable value, all CUDA kernel launch API calls are redirected to the rate control component first. The rate control component accesses statistics generated by the rate monitor to examine whether the rate limit is satisfied and defers the kernel launch if the rate of the opportunistic container exceeds the rate limit.

Unified memory management. To implement transparent memory sharing, TGS intercepts CUDA driver API calls related to GPU memory allocation, such as `cuMemAlloc`, and replaces these calls with unified memory allocation calls using `cuMemAllocManaged`. We use `cuMemAdvise` to prioritize the allocation of GPU memory for production containers. Specifically, we use `cuMemAdvise` to set the preferred location of memory allocation as the current GPU to avoid eviction for production containers. When the production container finishes, the indirection layer in the opportunistic container would use CUDA driver API `cuMemPrefetchAsync` to prefetch memory located in the host memory transparently.

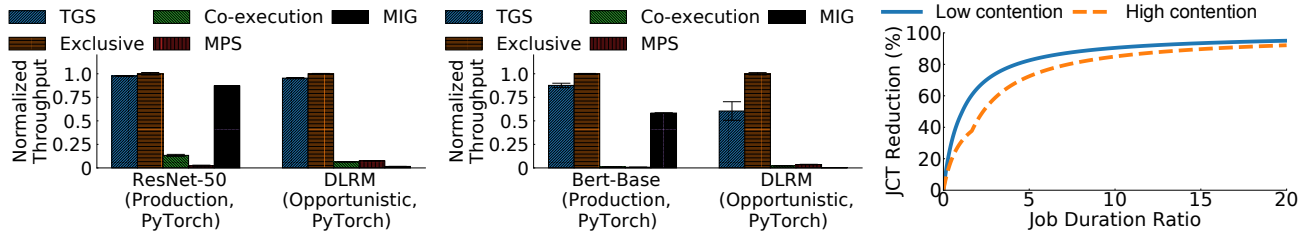
6 Evaluation

Setup. Most experiments are conducted on a server machine configured with an Intel Xeon Silver 4210R CPU, two NVIDIA A100 40 GB PCIe GPUs and 126 GB host memory. AntMan [6] only open-sourced one particular version based on TensorFlow 1.15.4 and the version is not compatible



(a) Low-contention scenario. (b) High-contention scenario. (c) JCT reduction of TGS over Exclusive.

Figure 4: Throughput of production and opportunistic jobs for different model pairs when GPU memory is sufficient.



(a) Low-contention scenario. (b) High-contention scenario. (c) JCT reduction of TGS over Exclusive.

Figure 5: Throughput of production and opportunistic jobs for different model pairs under GPU memory oversubscription.

with A100. Therefore, all experiments involved in TensorFlow are conducted on an AWS p3.2xlarge instance which is configured with eight Intel Xeon Scalable (Skylake) vCPUs, one NVIDIA V100 16 GB Tensor Core GPU and 61 GB host memory. The software environment includes NVIDIA driver 460.91.03, CUDA 11.2, Docker 20.10.5, PyTorch 1.9.0, TensorFlow 1.15.4, torchvision 0.10.0 and scipy 1.6.3.

Workloads. We use various models for evaluation. The models include ShuffleNet, MobileNet, GCN (Graph Convolutional Network), ResNet-50, BERT-Base, DLRM (Deep Learning Recommendation Model) and ESPnet2. These models are representative and widely-used, and are standard benchmarks for evaluating DL systems. They vary in terms of GPU resource usage, which allows us to evaluate TGS under different levels of GPU resource contention.

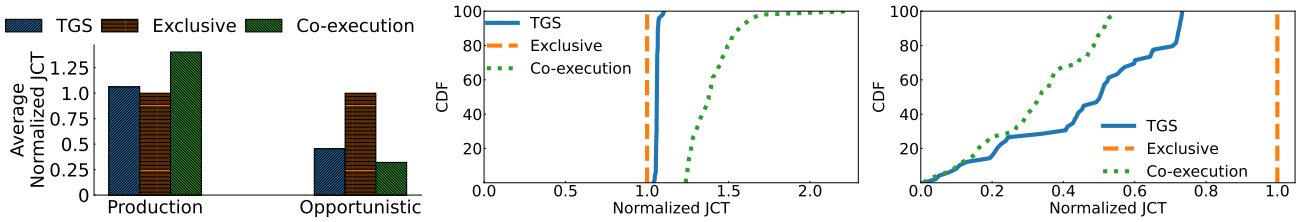
Comparison. To demonstrate the benefits of TGS, we compare the following mechanisms in the experiments. Each job runs in a separate container. We use throughput (iterations per second) as the main metric to evaluate the performance of different mechanisms, because it is a direct metric of a job's speed. We run at least 100 seconds for each case to measure the variance of the throughput, which typically includes 2000 iterations of a DL training job. Because a DL training job performs the same computation for each iteration (only the input data is different), the variance is low. We also use job completion time (JCT), but it depends both on the throughput and the number of iterations. The latter is configured by the user and varies from job to job.

- **TGS.** This is the proposed system.
- **Exclusive.** The production and opportunistic jobs are given exclusive access to a GPU when they run.
- **Co-execution.** The production job and the opportunistic job are executed concurrently without TGS.
- **NVIDIA MPS.** The production job and the opportunistic job run concurrently with NVIDIA MPS. We manually find the appropriate resource limit to set for each job in MPS to ensure that the performance of the production job is not affected by the opportunistic job.
- **NVIDIA MIG.** We manually set the best configuration to partition GPUs into different GPU instances so that the performance degradation of the production job brought by the opportunistic job is minimal.

Due to the compatibility issue of AntMan [6], we compare it with TGS in §6.7.

6.1 Adaptive Rate Control

TGS uses an adaptive rate control approach to allocate GPU compute resources between containers in order to simultaneously achieve high GPU utilization and performance isolation. In this experiment, we show that TGS packs an opportunistic job with a production job on a GPU to increase GPU utilization when the production job cannot use up all the GPU resources, and that the overhead of the production job is 5% to 10.8%. We use two different pairs of DNN models for the production job and the opportunistic job to evaluate TGS under different scenarios of resource contention. In this experiment, the total required GPU memory of the two jobs does not exceed the GPU memory capacity. This allows us to focus on evaluating the effectiveness of adaptive rate control. In the experiment, the two jobs arrive at the same time, and we measure the throughput for each job. To clearly show the difference between the five mechanisms, we normalize the throughput of each mechanism to that of Exclusive.



(a) Average JCT. (b) CDF of production jobs. (c) CDF of opportunistic jobs.

Figure 6: Performance comparison under a mixed workload job stream.

Figure 4a compares the performance of the five mechanisms when the production job trains ShuffleNet with batch size 4 and the opportunistic job trains MobileNet with batch size 4. These two models are small, so this case has low resource contention, and the throughput of the production job and the opportunistic job is almost the same for the five mechanisms. The overhead of TGS is 5%.

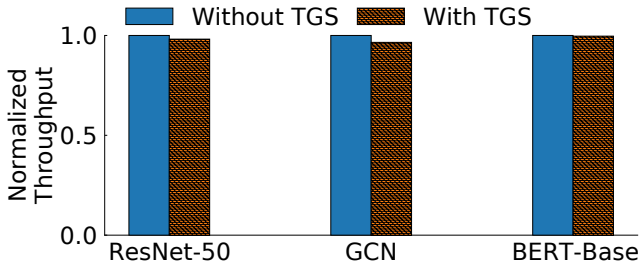
Figure 4b shows the results when the production job trains ResNet-50 with batch size 24 and the opportunistic job trains ShuffleNet with batch size 64. Both models are more computation-intensive than the models in Figure 4a. Thus, this case has a higher resource contention. TGS and MPS provide higher performance of the production job compared to Co-execution, because TGS and MPS control the resource allocation. Co-execution does not provide performance isolation, so the contention with the opportunistic job causes the throughput of the production job to reduce to 57% of that under Exclusive. The opportunistic job gets more resources than it should get by contending with the production job under co-execution. Thus the throughput of the opportunistic job under co-execution is high. TGS incurs 10.8% overhead for the production job although the resource contention is high. The performance provided by MPS is also comparable with TGS, although MPS sacrifices fault isolation. MIG only provides limited configurations for each GPU instance. On an NVIDIA A100 GPU, each GPU instance can only use at most one half of the total GPU memory and 4/7 of total SMs for GPU computation when a GPU is partitioned into two instances. In the high contention scenario, when the production job needs more GPU SMs than 4/7 for computation, the performance of the production job suffers, and is reduced to 77% of that under Exclusive. The opportunistic job gets more resources than it should, so its throughput is quite high.

While TGS protects production jobs from high contention caused by the opportunistic job, some sharing overhead is inevitable. In terms of throughput, Exclusive slightly outperforms TGS, because Exclusive runs DL models exclusively on the GPU. However, in this case, opportunistic jobs have to wait until the completion of the production job before execution. This leads to longer JCT for opportunistic jobs. Figure 4c shows that as the ratio of the job duration of the production job to that of the opportunistic job becomes larger, TGS can significantly reduce the queuing delay and thus speed up the opportunistic job over Exclusive. When the ratio is 20, TGS can reduce the JCT of the opportunistic job by 95% than Exclusive at the low-contention scenario and by 47% at the high-contention scenario.

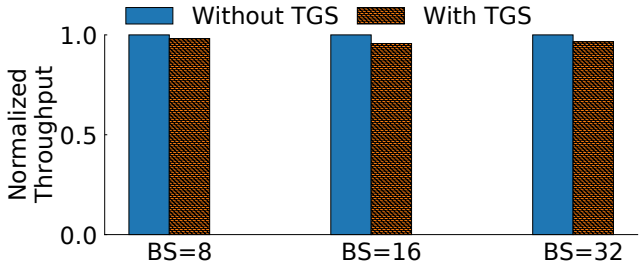
6.2 Unified Memory Management

In this experiment, we show that TGS provides high GPU utilization and performance isolation for GPU sharing even when the GPU memory is oversubscribed. We use two different pairs of DNN models to evaluate TGS under different scenarios. To oversubscribe the GPU memory, we use DLRM as the model of the opportunistic job for both pairs. DLRM is a large recommendation model with high GPU memory consumption. Similar to previous experiments, two jobs arrive at the same time, and we measure the throughput of each job. To clearly show the differences between the five mechanisms, we normalize the throughput of each mechanism to that of Exclusive for each job. Because MPS and Co-execution do not support GPU memory oversubscription, we modify the DL frameworks to use unified memory to evaluate them.

Figure 5a compares the performance of the five mechanisms when the production job trains ResNet-50 with batch size 16 and the opportunistic job trains DLRM with batch size 2048. The overhead of TGS is 2.3% compared to Exclusive. Co-execution has lower throughput due to resource contention. While MPS can set resource limits for SM usage, it cannot prioritize GPU memory allocation, and the two jobs contend for GPU memory resources when the GPU memory is oversubscribed. This causes significant memory swapping between GPU memory and host memory for both jobs, which degrades the performance of the production job under GPU memory oversubscription. MIG can partition the GPU memory resources, but it cannot provide sufficient GPU SMs with the production job due to the configuration constraints. Therefore, the performance of the production job under MIG is lower than that of Exclusive and TGS. In terms of the opportunistic job, Co-execution and MPS have lower throughput due to GPU memory contention. TGS improves the throughput by 7.8× over MPS for the opportunistic job by prioritizing memory allocation. MIG cannot partition GPU memory flexibly. The GPU instance of the opportunistic job can only use one half of the GPU memory to maintain performance of the production job. Therefore, the throughput of the opportunistic



(a) Different DNN models.



(b) Different batch sizes.

Figure 7: System overhead of TGS.

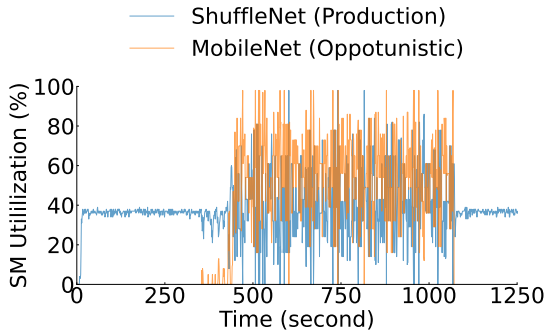
job under MIG is even lower than that of Co-execution and MPS.

Figure 5b shows the results when the production job trains BERT-Base with batch size 4 and the opportunistic job trains DLRM with batch size 256. BERT-Base is more computation-intensive than ResNet-50, and thus there is heavier contention. TGS maintains the performance as Exclusive with 12.3% overhead for the production job. Due to heavier contention, Co-execution and MPS perform worse for the production job. Due to more GPU compute resource demand, MIG performs also worse. TGS improves the throughput by 36× over Co-execution, 72× over MPS, and 1.5× over MIG for the production job. TGS also performs the best for the opportunistic job compared to MIG, MPS and Co-execution. They are slower due to resource contention and simply use unified memory without leveraging priority information. For the opportunistic job, TGS improves the throughput by 24×, 15× and 259×, compared to co-execution, MPS, and MIG, respectively.

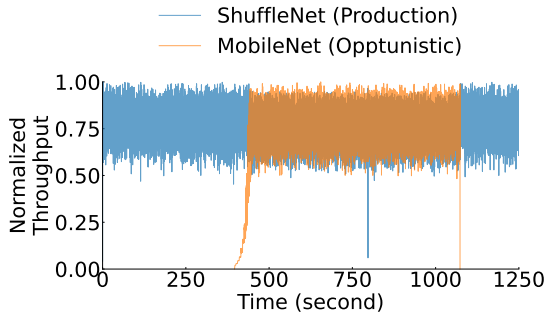
Exclusive provides all GPU resources to the production job, even though GPU resources are not fully utilized. As a result, the opportunistic job has a long queuing time—it has to wait for the production job to finish before it can be executed. As shown in figure 5c, when the ratio of the job duration of the production job to that of the opportunistic job reaches 20, TGS reduces the JCT of the opportunistic job by 95% over Exclusive at the low-contention scenario and by 92% at the high-contention scenario.

6.3 Mixed Workload Job Stream

In this experiment, we compare TGS with Exclusive and Co-execution when sharing a GPU between a mixed workload job stream. The DNN models used in the trace are consistent with previous experiments, including ResNet-50, MobileNet,



(a) GPU utilization.



(b) Training throughput.

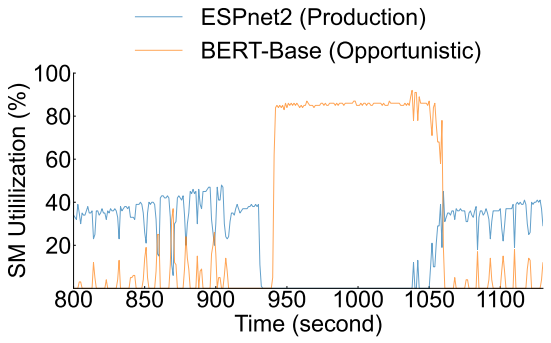
Figure 8: Convergence under dynamic job arrival.

ShuffleNet, GCN, BERT-Base, and DLRM. The running time of the jobs are from a production DL training job trace of Microsoft [5]. The job stream contains 100 jobs, where half are production jobs and the other half are opportunistic jobs. We use fast-forwarding [18] to speed up the experiment. NVIDIA MIG and NVIDIA MPS cannot dynamically change GPU resources allocated to a DL training job, so we do not compare them in this experiment.

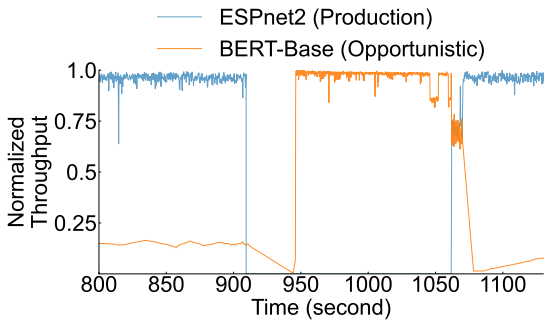
Figure 6a shows the average JCT when executing the trace. For fair comparison, we normalize the JCT of each mechanism to that of Exclusive for each job. As shown in figure 6b, because Co-execution cannot protect production jobs from contention caused by GPU sharing, the average normalized JCT of production jobs under Co-execution is 135% of that under Exclusive, while TGS only incurs 6% overhead. Compared to Exclusive, Figure 6c shows that TGS can significantly reduce the JCT of opportunistic jobs. This is because TGS can reduce the queueing time of opportunistic jobs, as they can use remaining GPU resources not used by production jobs, instead of waiting for production jobs to complete. TGS reduces the average normalized JCT of opportunistic jobs to 48% of that under Exclusive.

6.4 System Overhead

TGS monitors the rate of production jobs, and relies on the monitoring to decide whether a GPU can be shared and how many resources can be allocated to opportunistic jobs. When a GPU is shared, experiments in previous sections have demonstrated that opportunistic jobs do not greatly affect production

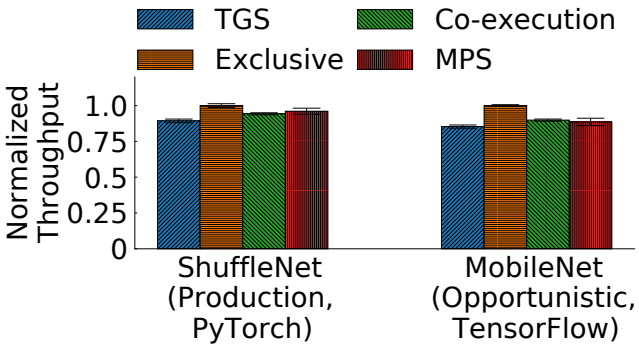


(a) GPU utilization.

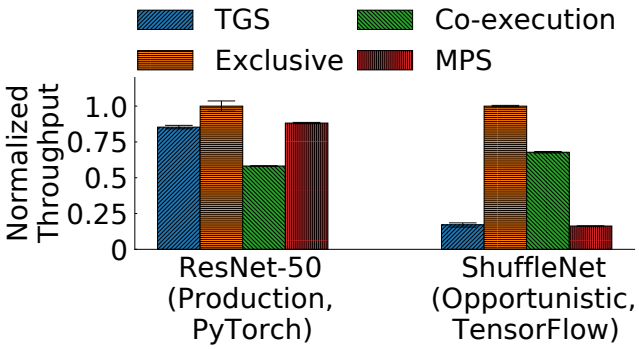


(b) Training throughput.

Figure 9: Convergence under dynamic resource usage.



(a) Low-contention scenario.



(b) High-contention scenario.

Figure 10: GPU sharing between different DL frameworks.

jobs. In this experiment, we explore the system overhead of the rate monitoring component in TGS. We measure the throughput of a job with and without TGS for different configurations, and normalize the throughput to that without TGS.

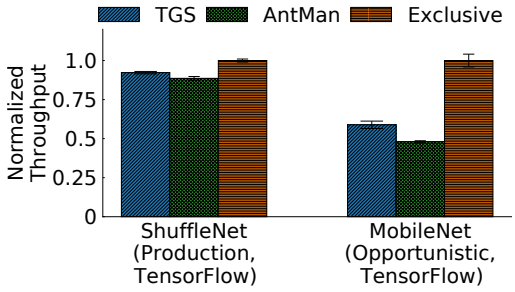
Figure 7a shows the throughput under different DNN models. The throughput is almost the same with and without TGS for ResNet-50, GCN and BERT-Base. Figure 7b shows the throughput under different batch sizes. We use ResNet-50 as the DNN model. Similarly, the JCT is almost the same with and without TGS for batch size 8, 16 and 32. The results demonstrate that the rate monitoring component of TGS incurs 0.3% to 5% overhead for production jobs.

6.5 Convergence

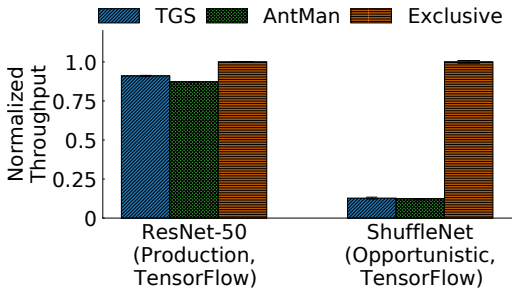
We evaluate the convergence of TGS in different scenarios. The first scenario evaluates the convergence under *dynamic job arrivals*, i.e., a job arrives in the middle to share the GPU with an existing job. In this scenario, the production job training ShuffleNet with batch size 4 is running in the beginning. The opportunistic job training MobileNet with batch size 4 is started after 350 seconds and runs for 240 seconds before it finishes. Figure 8a and Figure 8b show the time series of the GPU utilization and normalized throughput, respectively. As shown in Figure 8a, there are still idle GPU resources when the production job runs, so the total GPU utilization increases when the two jobs run concurrently and share the GPU. Figure 8b shows that the throughput of the opportunistic job increases when it is launched at 350 seconds. At the same time, GPU sharing does not affect the throughput of the production job.

The second scenario evaluates the convergence under *dynamic resource usage*, i.e., a job dynamically switches between high and low GPU utilization, and the other job utilizes the unused GPU resources. In this scenario, the production job trains ESPnet2 with batch size 1 and the opportunistic job trains BERT-Base with batch size 16. ESPnet2 has several phases, so it changes GPU utilization periodically. Figure 9a and Figure 9b show the time series of the GPU utilization and normalized throughput during a transition, respectively. When ESPnet2 needs more GPU resources, the production job keeps its maximum throughput. Between 910 and 940 seconds, ESPnet2 does not train, but runs validation in the GPU. Thus ESPnet2 still utilizes GPU but the throughput is zero. After 940 seconds, ESPnet2 runs into a phase that primarily uses CPU, and Figure 9a shows that the GPU utilization of ESPnet2 decreases to 0. TGS detects the change and dynamically allocates more GPU resources to the opportunistic job. After 1060 seconds, the production job starts using GPU again and reclaims all GPU resources. TGS ensures that the production job is not greatly affected by the opportunistic job.

In summary, these experiments demonstrate that TGS can converge in different scenarios. On the contrary, MIG cannot change GPU resource allocation to each GPU instance whenever there is a job running on the GPU, and MPS cannot change GPU resources allocated to a job after the job begins.



(a) Low-contention scenario.



(b) High-contention scenario.

Figure 11: Comparison between TGS and AntMan.

6.6 Supporting Different DL Frameworks

The experiments in previous sections are based on PyTorch, because TensorFlow-like frameworks claim all GPU memory by default when DL models start and the baselines cannot be directly used for GPU sharing for these frameworks. Specifically, Co-execution does not support GPU memory oversubscription or GPU memory allocation on demand. When one job claims all GPU memory, another job cannot use any GPU memory and would be aborted under Co-execution. MPS also suffers from this behavior. To compare TGS with them, we modify DL frameworks to use CUDA unified memory and enable dynamic GPU memory allocation.

Figure 10a compares the performance of the four mechanisms when the production job trains ShuffleNet with batch size 4 on PyTorch and the opportunistic job trains MobileNet with batch size 4 on TensorFlow. The result is similar to that of Figure 4a.

Figure 10b compares the performance of the four mechanisms in the high contention scenario. The production job trains ResNet-50 with batch size 16 and the opportunistic job trains ShuffleNet with batch size 32. Similar to figure 4b, TGS reduces the throughput of the production job by 14% compared to Exclusive, while Co-execution reduces the throughput by 41%. MPS achieves comparable performance, but it has to be manually tuned and breaks fault isolation.

6.7 Comparison with AntMan

In this experiment, we compare TGS with AntMan [6], which is a state-of-the-art application-layer solution for GPU sharing. AntMan is closely coupled with DL frameworks and uses an application-layer metric, iteration time, to control the opportunistic job. The open-sourced GitHub repository of AntMan is not fully functional. It does not include the logic to dynamically allocate resources to jobs. We contacted the authors of AntMan and followed their instructions to add necessary code in order to run AntMan. Figure 11a and 11b show the comparison under low-contention (ShuffleNet with batch size 4 and MobileNet with batch size 4) and high-contention scenarios (ResNet-50 with batch size 8 and ShuffleNet with batch size 4), respectively. Although AntMan uses application-layer knowledge and controls the jobs at the application layer, TGS still achieves similar performance to AntMan. The throughput of the production job under TGS is 104.1% to 104.3% than that under AntMan, while the throughput of the opportunistic job using TGS is 103% to 122% than that of AntMan. Compared to AntMan, TGS provides the same benefit of GPU sharing and is transparent to DL frameworks.

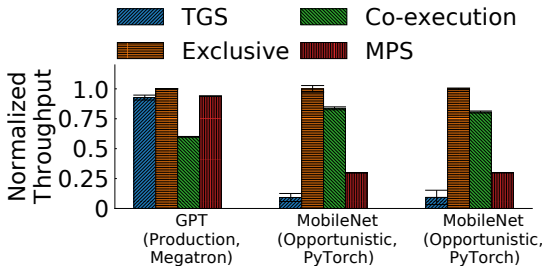


Figure 12: GPU sharing with the large model.

6.8 GPU Sharing for Large Model Training

In §6.2, we have shown that even if a large model (e.g., DLRM) with large batch size (e.g., 2048) and large memory consumption (e.g., 38 GB) runs on a GPU, TGS can still mostly maintain the performance of the production job, while providing the remaining GPU resources to the opportunistic job. In this experiment, we show that although it is not a common scenario, TGS can provide GPU sharing capability when training a bigger model (e.g., GPT). We train a GPT with batch size 32 using two NVIDIA A100 GPUs as the production job, while running two single-GPU opportunistic jobs training MobileNet with batch size 4. Figure 12 shows that TGS still can achieve comparable performance compared to MPS, while MPS breaks fault isolation and Co-execution breaks performance isolation. NVIDIA MIG does not support multi-GPU jobs when a GPU is partitioned into several GPU instances, so it is not evaluated in this case.

7 Discussion

Distributed training. Many solutions have been proposed to achieve high GPU utilization for distributed training jobs [19–23]. With these solutions, it is unlikely that a distributed training job would leave substantial GPU resources unused; otherwise, the job should reduce its GPUs. Therefore, there is little need for TGS. It is most suitable for sharing GPUs between single-GPU jobs, which is also how GPU sharing is used in previous solutions [6, 12, 13]. Yet, TGS

can be applied to increase GPU utilization for unoptimized distributed jobs, by controlling the GPU resource usage of an opportunistic job on each GPU as for single-GPU jobs.

GPU cluster scheduling. Many solutions [7, 18, 24–28] have been proposed to minimize job completion time and provide fairness for a GPU cluster. GPU cluster scheduling is orthogonal and complementary to TGS. TGS provides the mechanism for transparent GPU sharing, which can be used by cluster schedulers when they schedule and place jobs. We note that some schedulers [18, 28] pack multiple jobs on a GPU, which are at the application level and require modifications to DL frameworks. Also, they do not support GPU memory oversubscription. These schedulers can benefit from TGS.

Space sharing and time sharing. The concepts of GPU compute sharing and memory sharing are orthogonal to space sharing and time sharing. Sharing GPU compute resources can be done either in space sharing or in time sharing. The adaptive rate control mechanism and transparent unified memory mechanism of TGS can be used either in space sharing or in time sharing. GPU space sharing needs hardware support and is not well supported. Current space sharing solutions reduce performance isolation (e.g. MIG) or fault isolation (e.g. MPS). Therefore, TGS currently uses time sharing.

8 Related Work

Deep learning systems. Many DL frameworks have been proposed for developing and running DNN models [29–36]. Some works optimize communication to improve distributed training performance [19–23]. Some works use memory swapping to handle the GPU memory problem for training large DNN models [16, 37–39]. They focus on improving the performance of a single training job, while TGS provides a solution for improving the GPU utilization of running many jobs in a cluster. Some works [40, 41] propose algorithms for inter-job GPU memory management, but they are not transparent to applications and require modifications to DL frameworks. GPUswap [42] proposes a transparent GPU memory swapping system, but it needs to modify GPU drivers. However, most current commercial GPU drivers, such as NVIDIA GPU drivers, are not open-source. Open-source GPU drivers are not as high performance as the commercial ones, so they are not widely used for DL training workloads. MIG-Serving [43] tries to find better configurations to use MIG for GPU sharing. However, MIG itself has limitations as described above. We compare MIG with the best configuration and TGS in the evaluation section, and show the benefits of TGS. There are many solutions for optimizing DL inference workloads [44, 45]. We focus on GPU clusters for training workloads in this paper. Several scheduling algorithms have been designed to schedule DL training jobs in a GPU cluster [7, 18, 24–28]. These works are orthogonal to TGS.

Containers. Containers provide lightweight virtualization for applications. Due to the benefits of portability, isolation and performance, containers are widely used in datacenters. Major public cloud services, such as AWS, Microsoft Azure and Google Cloud, offer containers as a service [46–48]. Many container runtimes (e.g., Docker) and orchestration systems (e.g., Kubernetes) are developed and deployed [1–3, 10, 11, 49, 50]. Some work is proposed to provide high-performance networking with isolation [51–56]. These solutions are orthogonal to TGS, which focuses on improving GPU utilization.

GPU sharing. Several solutions have been proposed for GPU sharing. Early solutions [57–65] explored OS-layer techniques like driver call interception and application-layer techniques like introducing new programming APIs, for sharing GPU between applications. They focus on jobs with a few kernels, and are not specifically designed for DL training that typically has hundreds of kernels. With the emergence of DL applications, recent solutions [6, 12, 13] have been designed for GPU sharing of DL training. AntMan [6] is the state-of-the-art application-layer solution for GPU sharing. Salus [12] uses centralized GPU memory management and kernel scheduling for GPU sharing. It requires all the applications to fit in the GPU memory. PipeSwitch [13] provides fast context switching for DNN jobs, but only one job can run at each time. They all modify DL frameworks. MPS [9] is an OS-layer solution, but it requires application knowledge to correctly set resource limits, does not support GPU memory oversubscription and does not provide fault isolation. Planaria [66] is an accelerator designed for the multi-tenant scenario. In comparison, TGS is a software solution that can be used for sharing a variety of hardware.

9 Conclusion

We have presented TGS, a system that transparently shares GPUs for DL workloads to improve GPU utilization in container clouds. TGS is distinguished from state-of-the-art application-layer solutions in that it enables users to use any DL framework and library to develop and train DNN models in containers. Shared GPUs are exposed to containers as regular GPU devices, and TGS transparently runs multiple containers on a GPU when a single container cannot utilize all GPU resources. TGS achieves both high utilization and decent performance isolation.

Acknowledgments. We sincerely thank our shepherd John Wilkes and the anonymous reviewers for their valuable feedback. This work was supported by the National Key Research and Development Program of China under the grant number 2020YFB2104100, the National Natural Science Foundation of China under the grant number 62172008 and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xin Jin is the corresponding author. Bingyang Wu, Zili Zhang, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] "containerd." https://containerd.io/.

[2] "cri-o." https://cri-o.io/.

[3] "Docker." https://www.docker.com/.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, 2015.

[5] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX ATC*, 2019.

[6] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on GPU clusters for deep learning," in *USENIX OSDI*, 2020.

[7] W. Qizhen, X. Wencong, Y. Yinghao, W. Wei, W. Cheng, H. Jian, L. Yong, Z. Liping, L. Wei, and D. Yu, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *USENIX NSDI*, 2022.

[8] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang, "HiveD: Sharing a GPU cluster for deep learning with guarantees," in *USENIX OSDI*, 2020.

[9] "CUDA Multi-Process Service." https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[10] "Kubernetes." https://kubernetes.io/.

[11] "Docker Swarm." https://docs.docker.com/engine/swarm/.

[12] P. Yu and M. Chowdhury, "Salus: Fine-grained GPU sharing primitives for deep learning applications," in *Conference on Machine Learning and Systems*, 2020.

[13] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *USENIX OSDI*, 2020.

[14] "Nvidia multi-instance GPU (MIG)." https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[15] "Nvidia multi-instance GPU user guide." https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[16] "CUDA Unified Memory." https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[17] J. Gleeson, S. Krishnan, M. Gabel, V. J. Reddi, E. de Lara, and G. Pekhimenko, "RL-Scope: Cross-stack profiling for deep reinforcement learning workloads," in *Conference on Machine Learning and Systems*, 2021.

[18] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX OSDI*, 2018.

[19] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[20] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *ACM SOSP*, 2019.

[21] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *USENIX OSDI*, 2020.

[22] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *ACM SOSP*, 2019.

[23] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and generic collectives for distributed ML," in *Conference on Machine Learning and Systems*, 2020.

[24] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *USENIX NSDI*, 2019.

[25] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *ACM Symposium on Cloud Computing*, 2017.

[26] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *EuroSys*, 2018.

[27] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient GPU cluster scheduling," in *USENIX NSDI*, 2020.

[28] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *USENIX OSDI*, 2020.

[29] "TensorFlow." https://www.tensorflow.org/.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *USENIX OSDI*, 2016.

[31] "PyTorch." https://pytorch.org/.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019.

[33] "MXNet." https://mxnet.apache.org/.

[34] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *LearningSys at Neural Information Processing Systems*, 2015.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *USENIX OSDI*, 2014.

[36] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012.

[37] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[38] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *ACM ASPLOS*, 2020.

[39] G. Wang, K. Wang, K. Jiang, X. LI, and I. Stoica, "Wavelet: Efficient dnn training with tick-tock scheduling," in *Conference on Machine Learning and Systems*, 2021.

[40] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based gpu memory management for deep learning," in *ACM ASPLOS*, 2020.

[41] G. Lim, J. Ahn, W. Xiao, Y. Kwon, and M. Jeon, "Zico: Efficient GPU memory sharing for concurrent DNN training," in *USENIX ATC*, 2021.

[42] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of gpu memory through transparent swapping," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[43] C. Tan, Z. Li, J. Zhang, Y. Cao, S. Qi, Z. Liu, Y. Zhu, and C. Guo, "Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem," *arXiv preprint arXiv:2109.11067*, 2021.

[44] J. Kosaian, K. V. Rashmi, and S. Venkataraman, "Parity models: Erasure-coded resilience for prediction serving systems," in *ACM SOSP*, 2019.

[45] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *ACM SOSP*, 2019.

[46] "AWS containers." https://aws.amazon.com/containers/.

[47] "Microsoft azure containers." https://azure.microsoft.com/en-us/product-categories/containers/.

[48] "Google cloud containers." https://cloud.google.com/containers.

[49] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*, 2015.

[50] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys*, 2013.

[51] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "Freeflow: Software-based virtual RDMA networking for containerized clouds," in *USENIX NSDI*, 2019.

[52] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a low-overhead container overlay network," in *USENIX NSDI*, 2019.

[53] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang, "Socksdirect: Datacenter sockets can be fast and compatible," in *ACM SIGCOMM*, 2019.

[54] Z. He, D. Wang, B. Fu, K. Tan, B. Hua, Z.-L. Zhang, and K. Zheng, "Masq: RDMA for virtual private cloud," in *ACM SIGCOMM*, 2020.

[55] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: a microkernel approach to host networking," in *ACM SOSP*, 2019.

[56] A. Narayan, A. Panda, M. Alizadeh, H. Balakrishnan, A. Krishnamurthy, and S. Shenker, "Bertha: Tunneling through the network API," in *ACM SIGCOMM HotNets Workshop*, 2020.

[57] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *European Conference on Parallel Processing*, 2010.

[58] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *ACM Workshop on System-level Virtualization for High Performance Computing*, 2009.

[59] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *International Conference on High Performance Computing & Simulation*, 2010.

[60] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *IEEE HPDC*, 2011.

[61] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, 2011.

[62] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *ACM ASPLOS*, 2013.

[63] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *ACM ASPLOS*, 2015.

[64] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-grained GPU resource virtualization for narrow tasks," in *ACM PPoPP*, 2017.

[65] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-NET: Effective GPU sharing in NFV systems," in *USENIX NSDI*, 2018.

[66] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *IEEE/ACM MICRO*, 2020.

A Convergence of Adaptive Rate Control Algorithm

We assume each GPU has an unknown constant throughput limit B. The TGS's goal is to maximize throughput of the opportunistic job without affecting the production job very much. We assume throughput of the production job is relatively stable. Therefore, the adaptive rate control algorithm can accurately measure the throughput of the production job, i.e. α_{in}. When throughput of the production job is unstable beyond a manual tuned threshold, TGS re-estimates α_{in}. In this context, we define that a *cycle* is a phase starting after TGS detects contention and ending when TGS detects contention again. A *step* is defined as an invocation of the rate control component to adjust rate limit of the opportunistic job, such as an additive increase or a multiplicative decrease. Hence, a *cycle* consists of one multiplicative decrease *step* and multiple continous additive increase *step*s. Let the initial value of β_{out} be β_0 ($\beta_0 \leq B$). The simplified convergence of the rate adaptive control algorithm is shown as follow:

Opportunistic Job	production Job
β_0	$\min(R, B-\beta_0)$
$\beta_1 = \beta_0 + \delta_{AI}$	$\min(R, B-\beta_1)$
$\beta_2 = \beta_0 + \delta_{AI} + \delta_{AI}$	$\min(R, B-\beta_2)$
$\vdots$	$\vdots$
$\beta_k = \beta_0 + \underbrace{\delta_{AI} + \cdots + \delta_{AI}}_{k}$	$\min(R, B-\beta_k)$
Detect Contention:	$R + \beta_0 + k\delta_{AI} \geq B$
Action:	Multiplicative Decrease
$\beta_{k+1} = \frac{\beta_0 + k\delta_{AI}}{\delta_{MD}}$	$\min(R, B-\beta_{k+1})$
$\beta_{k+2} = \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + \delta_{AI}$	$\min(R, B-\beta_{k+2})$
$\beta_{k+3} = \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + \delta_{AI} + \delta_{AI}$	$\min(R, B-\beta_{k+3})$
$\vdots$	$\vdots$
$\beta_{k+l+1} = \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + \underbrace{\delta_{AI} + \cdots + \delta_{AI}}_{l}$	$\min(R, B-\beta_{k+l+1})$
Detect Contention:	$R + \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + l\delta_{AI} \geq B$
Action:	Multiplicative Decrease
$\beta_{k+l+2} = \frac{\beta_0}{\delta_{MD}^2} + \frac{k\delta_{AI}}{\delta_{MD}^2} + \frac{l\delta_{AI}}{\delta_{MD}}$	$\min(R, B-\beta_{k+l+2})$
$\vdots$	$\vdots$
$\beta^* = \frac{\beta_0}{\delta_{MD}^{\log \beta_0}} + \frac{k\delta_{AI}}{\delta_{MD}^{\log \beta_0}} + \frac{l\delta_{AI}}{\delta_{MD}^{\log \frac{\beta_0}{2}}} + \cdots + m$	$\min(R, B-\beta^*)$

We assume the unit of bandwith is indivisible. As shown above, the adaptive rate control algorithm converge in $O(\log \beta_0)$ *cycle*s, because the unknown term β_0 decreases to zero in $O(1 + \log \beta_0)$ *cycle*s, i.e. $O(B \log B)$ *step*s. Therefore, the complexity of the adaptive rate control algorithm is $O(B \log B)$.

ARK: GPU-driven Code Execution for Distributed Deep Learning

Changho Hwang[1,2], KyoungSoo Park[1], Ran Shu[2], Xinyuan Qu[2,†], Peng Cheng[2], and Yongqiang Xiong[2]

[1]*KAIST* [2]*Microsoft Research*

Abstract

Modern state-of-the-art deep learning (DL) applications tend to scale out to a large number of parallel GPUs. Unfortunately, we observe that the collective communication overhead across GPUs is often the key limiting factor of performance for distributed DL. It under-utilizes the networking bandwidth by frequent transfers of small data chunks, which also incurs a substantial I/O overhead on GPU that interferes with computation on GPU. The root cause lies in the inefficiency of CPU-based communication event handling as well as the inability to control the GPU's internal DMA engine with GPU threads.

To address the problem, we propose a ***GPU-driven*** code execution system that leverages a GPU-controlled hardware DMA engine for I/O offloading. Our custom DMA engine pipelines multiple DMA requests to support efficient small data transfer while it eliminates the I/O overhead on GPU cores. Unlike existing GPU DMA engines initiated only by CPU, we let GPU threads directly control DMA operations, which leads to a highly efficient system where GPUs drive their own execution flow and handle communication events autonomously without CPU intervention. Our prototype DMA engine achieves a line-rate from a message size as small as 8KB (3.9x better throughput) with only 4.3µs of communication latency (9.1x faster) while it incurs little interference with computation on GPU, achieving 1.8x higher all-reduce throughput in a real training workload.

1 Introduction

Modern machine learning (ML) applications tend to harness an increasingly larger number of accelerators (especially GPUs in this work) [19, 26]. State-of-the-art deep learning (DL) algorithms often need to scale out to thousands of GPUs for higher throughput and accuracy [26]. Unfortunately, this poses a substantial communication overhead to the entire system, which harms GPU utilization by delaying or interfering with numeric computation.

The communication overhead mainly arises in two different aspects. First, collective communication (e.g., all-reduce, split-and-gather, all-to-all, etc.), which is widely adopted in most of popular DL algorithms, often splits the data for transfer into multiple small chunks for pipelining or for sending to multiple different destinations. The chunk size tends to get smaller as we scale out, which is detrimental to efficient utilization of networking bandwidth. Second, popular communication libraries for GPUs such as NCCL [32] and RCCL [5] often incur a severe I/O overhead on GPU. This is because they commonly leverage memory-mapped I/O (MMIO) for data copies between GPUs, which consumes a substantial amount of GPU resources (i.e., core cycles and L2 cache/DRAM bandwidth). We observe that concurrent execution of collective communication and numeric computation on GPU heavily interferes with each other – in our training experiment with BERT-Large [10], the throughput of parallel computation drops by 45% while it achieves only 53.6% of the peak communication throughput (see details in Section 2.3).

Unfortunately, it is challenging for existing systems to address both issues (i.e., large transfer delay for small chunks and I/O overhead on GPU) at the same time. One may avoid the I/O overhead by offloading the I/O to a hardware DMA engine instead of employing MMIO with GPU threads. However, the current DMA engine on commodity GPU is initiated only by CPU threads, which often enrolls CPU's control on the critical path of communication. This incurs the CPU-GPU synchronization overhead that bloats up the communication latency, especially detrimental to the throughput of small data chunk transfer. In fact, one can observe hundreds of µs of communication latency in a popular DL framework as it leverages the DMA engine. Similarly, if one does not employ the DMA engine for communication of data chunks, the communication would suffer from high I/O overhead on GPU.

This paper proposes the *GPU-driven* system named *ARK*, a communication-motivated DL system design. The key idea of the GPU-driven system lies in autonomous execution control of GPU code without any control by external devices. This regime tightly connects computational power of every GPU

[†] Now at Horizon Robotics.

core across machines by allowing GPU threads to communicate directly with remote GPUs without any external control signals, which ends up achieving low-latency communication. At the same time, to avoid the I/O overhead on GPU, we design a GPU-controlled DMA engine. Specifically, our custom DMA engine is directly initiated by GPU threads, which avoids the heavy MMIO without CPU intervention.

Our evaluation shows that our DMA engine prototype is especially beneficial for small messages, achieving a high communication throughput (3.87x over `cudaMemcpy` with 8KB messages) at low latency (9.1x faster over CPU intervention). Furthermore, it does not interfere with computation on GPU, which delivers both computation and communication throughput gains over using MMIO-based libraries [5,32] (1.8x faster all-reduce in BERT-Large [10] training, see Section 5.3).

To realize the GPU-driven system, we also present an efficient scheduler of autonomous execution on GPU. Our key observation is that online dynamic scheduling is unnecessary as DL workloads are typically deterministic at runtime. Instead, we present the *virtual Cooperative Thread Array* (vCTA) framework that abstracts *offline* GPU scheduling. Offline scheduling allows eliminating the runtime scheduling overhead at the back-end, while reusing the existing front-end interface and GPU kernel implementations.

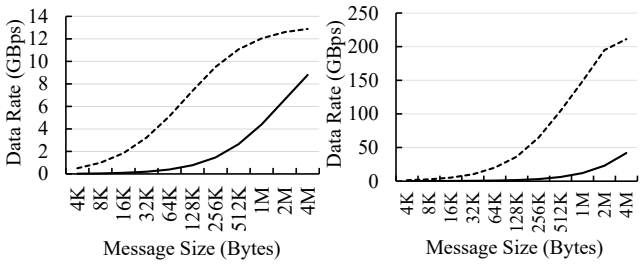
ARK supports efficient and flexible parallel execution models for data-, tensor-, and pipeline-parallelisms. Our evaluation demonstrates that ARK delivers substantial performance gains both in training and inference, achieving 2.5x and 3.6x throughput improvement, respectively.

2 Background & Motivation

This section explains existing inter-GPU communication technologies and their limitations.

2.1 Small Data Transfer in Distributed DL

Collective communication consists of several communication primitives that concurrently exchange the data across multiple GPUs, which is widely adopted to implement various parallelism methods in distributed DL. Popular use cases include *all-reduce* for data-parallelism, *split-and-gather* for tensor-parallelism [22,40], and *all-to-all* for expert-parallelism [11]. As the number of employed GPUs gets larger, the size of unit data transfer in collective communication becomes smaller as it splits the local data into multiple pieces to be delivered to different GPUs. This small transfer size makes the overall performance of collective communication highly dependent on the control plane overhead before and after each data transfer. Unfortunately, we observe that the control plane overhead either with *CPU-controlled* or even *GPU-controlled* communication is pretty substantial (See Section 2.2 and Section 2.3). Also, existing workarounds (e.g., *tensor fusion* [39]) that batch a large amount of data to avoid small transfers would not completely address the problem as they trade off computational throughput by intentionally delaying data transfer.



(a) PCIe v3. (b) NVLink v3.

Figure 1: Data dependency between GPUs decreases the inter-GPU data rate due to event handling delays. Solid lines refer to actual data rate (for sending one message at a time) in TensorFlow's CPU-controlled communication crossing a PCIe v3 or a NVLink v3 switch while dashed lines indicate the ideal data rate without event handling delays.

2.2 External Execution Control Overhead

Existing GPU program execution heavily relies on an external processor (i.e., CPU) to submit GPU commands for kernel execution or data transfer. Unfortunately, this model often incurs a large overhead due to the delay for command delivery from the host side to GPU hardware queue (i.e., *stream*). One can use the conventional GPU event interface (i.e., `cudaEvent`) to hide the delay, but it would also suffer from substantial delay for event handling. When adopted to inter-GPU communication, which we call *CPU-controlled* communication (in contrast to *GPU-controlled* communication by NCCL [32]), we observe that event handling becomes the primary cause for large communication delay beyond the data transfer itself.

We consider a common communication scenario where two GPUs have a data dependency – one GPU receives computation results of another GPU to feed them as input to its own computation. In every data transfer, event handling is needed to check the dependency between the copy and the GPU commands around the copy operation, which reduces the actual data rate between GPUs. Figure 1 compares the ideal inter-GPU data rate (`cudaMemcpy` throughput) with the actual data rate in TensorFlow's CPU-controlled communication, which is still used along with NCCL especially for model-parallelism implementations. We see that the event handling overhead with `cudaMemcpy` drastically lowers the data rate both in the PCIe and NVLink interfaces. We explain two implementations when GPU A sends data to GPU B.

2.2.1 Runtime Intervention for the Control

CPU can serve as an intermediary to deliver an event between two communicating GPUs. In fact, if GPUs are located in different NUMA nodes or on different machines, the runtime intervention by CPU is required for communication. Also, some frameworks like TensorFlow implement a generic interface that always uses CPU for GPU event handling regardless of the placement. Figure 2 illustrates the event handling overhead due to CPU intervention when GPU A sends its data to

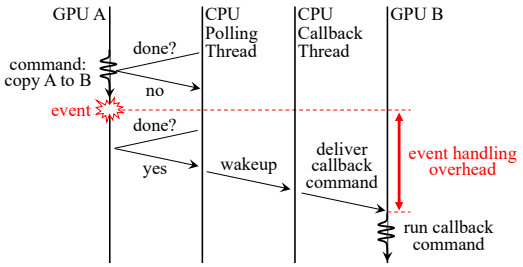


Figure 2: CPU intervention in inter-GPU event handling.

Overhead Detail	Delay (µs)
Initiation	
Trigger send ready event on the GPU	3.8
Sync comp. stream and comm. stream	11.6
Completion Check	
Event polling gap	58.3
Delay of pthread mutex lock	58.7
GPU kernel launch overhead	19.2
Total	**151.6**

Table 1: Breakdown of the constant overhead of inter-GPU data transfer using TensorFlow in Figure 1.

GPU B that plans to run the next command with the data.

We notice three places for the overhead. First, it is inefficient for a CPU thread to poll GPU events because the event interface disallows the CPU thread to monitor multiple events at the same time. While it takes only ∼3µs for a dedicated busy-waiting CPU thread to be notified of a triggered GPU event,[2] this approach does not scale when an application has to run many parallel tasks, which will run many polling threads. Instead, the event polling loop of TensorFlow uses only one CPU thread, which incurs a ∼58.3µs of polling gap on average (see Table 1). Second, it takes time to wake up the CPU thread that invokes the callback function of the triggered event. In TensorFlow, it takes ∼58.7µs for the callback thread to acquire the mutex lock from when it is released by the polling thread. This delay could be reduced to as low as 5µs if both threads are running on the same CPU core, but co-locating the threads or even merging them into a single one would increase the event polling interval as well as the overall processing time. Lastly, it is inefficient for the callback thread to deliver the computation command to GPU B. Delivering the event signal to GPU B would take only 2∼3µs if implemented efficiently,[3] but we need to deliver the callback command binary as well. We can avoid the extra delay if we deliver the GPU command in advance and trigger it later on the CPU side, but this is not supported by commodity GPU.

2.2.2 Asynchronous Control

If the GPUs are under the same NUMA node, CPU can reserve a GPU event to be triggered asynchronously so that GPUs can directly communicate with each other when the event occurs. In this case, one can deliver the callback command to GPU B before the actual event and use the conventional GPU event interface (i.e. `cudaEvent` or a higher-level wrapper such as CUDA Graphs [27]) to trigger the callback command on GPU B with GPU A's event. Ideally, this should take as short as sending a single bit from GPU A to GPU B. However, we find that triggering a GPU event (∼4µs) and waking up a dependent GPU command (10∼20µs) are disappointingly slow – it ends up taking as much as sending the command binary to the GPU at runtime. We suspect that this is due to inefficient hardware implementation on GPU for event handling. In TensorFlow, this overhead contributes to the delay for initiating a transfer that depends on GPU computation as shown in Table 1.

2.3 I/O Overhead of GPU-side Control

Since CPU intervention incurs a large overhead, how about managing the communication with GPU itself? NCCL [32] [4] leverages *GPUDirect* [31] to enable this approach, which exposes the GPU memory space for peer-to-peer access so that GPU threads can read/write data to/from another GPU.[5] As GPU threads can directly invoke data copy, they can handle communication events efficiently without the involvement of CPU. Since commodity GPU hardware disallows GPU threads to initiate its own DMA engine, GPU-controlled communication leverages MMIO, which will implicitly conduct DMA when GPU threads write data on the mapping. Figure 3 compares CPU-controlled and GPU-controlled communication. The former one (Figure 3a) takes the following steps: ① CPU is notified when the data is ready, ② CPU initiates the DMA engine, and ③ DMA copies the data. On the other hand, GPU-controlled communication with MMIO (Figure 3b) follows ① CPU creates a memory map (mmap) of the destination GPU's address space prior to runtime execution, ② the data is ready at runtime, and ③ GPU threads copy the data into the mmap, which implicitly conducts DMA copy.

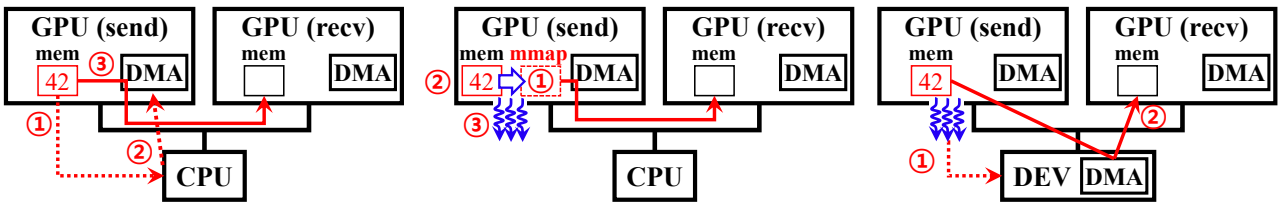
Unfortunately, data copying by GPU threads often heavily interferes with parallel kernel computation, especially due to L2 cache pollution and warp scheduler operations. Specifically, a data-copy GPU thread needs to load the data onto its register file for data transfer, but this pollutes the L2 cache as one cannot bypass the L2 cache when reading from DRAM on commodity GPU [34]. It leads to severe performance degradation over initiating DMA directly, as the latter copies the data on DRAM directly to the I/O bus (PCIe or NVLink). Additionally, the copying threads frequently issue 'load/store'

[2] Please refer to the experiment setup in Section 5.

[3] This is roughly estimated based on that it takes ∼2µs for a GPU thread to read a 4-byte data on the host DRAM and it takes ∼3µs for a busy-waiting CPU to read a GPU event.

[4] Equally applied to RCCL [5] on AMD GPU as well. For convenience, we borrow the terms from CUDA or NVIDIA GPUs, which can be easily converted into corresponding terms in OpenCL or AMD GPUs.

[5] CPU-controlled communication also leverages GPUDirect for efficient `cudaMemcpy` between peer GPUs without crossing the root complex, but its execution path is different from that of GPU-controlled communication.



(a) CPU-controlled. (b) GPU-controlled MMIO. (c) GPU-controlled DMA (this work).

Figure 3: Comparison between CPU-controlled and GPU-controlled communication – the latter has two different approaches, which leverage (b) MMIO (like NCCL) or (c) directly initiated DMA (this work). DEV refers to any kinds of devices that can implement our DMA engine.

instructions that drive warp schedulers busy, which makes other threads for parallel computation yield their clock cycles. Although the affected computation threads are limited to those that co-run warp schedulers with data-copy threads, they delay the entire kernel by falling behind the other threads.

To analyze the impact of the contention, we measure the slowdown of two different GPU kernels that heavily access only a specific type of GPU resources each: L2 cache (1.96 TBps read) and warp schedulers (2.02 IPC),[6] respectively (all numbers measured on a V100 GPU), while running concurrently with NCCL (v2.11.4) 64 MB all-gather[7] kernels using 8x V100 GPUs. We leverage NVIDIA Visual Profiler (NVVP) and Nsight Compute to verify that (1) the L2 cache kernel shows near-zero DRAM access and L1 data cache hit rate and (2) the warp schedulers kernel shows near-zero L2 cache/DRAM throughput. We have also verified the concurrency of computation and all-gather kernels and no other CPU/GPU activities during the experiment. In this experiment, the slowdowns due to L2 cache and warp schedulers contention are up to 2.4x and 2.0x, respectively, where it slows down either the computation or the concurrent NCCL communication (when one side is degraded less, the other side tends to be impacted more). This result shows that heavy contention could arise depending on the GPU resource usage of concurrent computation kernels.

We run a microbenchmark to evaluate the contention of NCCL all-reduce during data-parallel training of a BERT-Large [10] model. This model performs 32 MB of all-reduce at a time, which issues 4 MB data transfer in parallel with eight GPU workers. On a server with 8x V100 GPUs (connected with a single PCIe switch (16x PCIe v3)), the parallel computation throughput drops by 45.0% while all-reduce achieves only 5.0 GBps on average, degraded to 53.6% of the peak throughput without the interference. On a server with 8x A100 GPUs (connected with an NVSwitch (NVLink v3)), the slowdown of all-reduce is even worse – the parallel computation throughput drops by 14.3% while the NCCL all-reduce achieves only 30.9% of the peak throughput (49.0 GBps).

[6] Heavy usage of warp schedulers means frequent instruction fetches, i.e. large instructions per cycle (IPC). $>$ 99.2% of instructions are FFMA.

[7] We use all-gather as it only performs communication without any extra computation such as reduction in all-reduce.

3 ARK Framework Design

In this section, we present the design of ARK, our approach with the *GPU-driven* code execution that avoids the communication overhead on GPU without CPU intervention.

3.1 GPU-controlled DMA Engine

We claim that a GPU-controlled DMA engine (Figure 3c) can eliminate the communication overhead, which in turn serves as the basis of our GPU-driven system. The GPU-controlled DMA engine enables a GPU thread to directly initiate DMA operations when the data is ready (①), which will immediately push the data into the I/O bus without wasting GPU cycles (②). We leverage existing GPUDirect techniques to expose the GPU's physical address space to our DMA engine.

While GPU-controlled DMA would deliver low-latency communication without the MMIO overhead, it is non-trivial to realize this feature. In fact, an ideal implementation would be to modify the existing DMA engine on GPU to support GPU-controlled DMA, but it is infeasible as we cannot update the GPU hardware. Instead, we consider employing an external device as illustrated in Figure 3c at the cost of extra communication latency from GPU threads.

Despite of performance benefits, adopting new hardware for GPU-controlled DMA engine might be costly in many existing systems. To provide an interim solution, we pursue a general DMA engine design that can be implemented as either software or hardware on any hardware platforms (e.g., CPU, GPU, SmartNIC, FPGA, etc.) or I/O bus types (PCIe, NVLink [33], or Infinity Fabric Link (xGMI) [3]). Regardless of the platform, all implementations need to share the same runtime interface for GPU kernels. Also, the DMA interface should support low latency and flexibility while meeting the different requirements of software and hardware engines.

In this paper, we present both a software implementation and a hardware prototype of GPU-controlled DMA engine. Our software engine works over any existing systems without additional hardware as it leverages host CPU cores – busy-waiting CPU threads read DMA requests from GPU and initiate DMA accordingly. This design is aligned with the principle of GPU-driven system as GPU threads directly initiate the data transfer, while CPU threads only mechanically initiate

data copies without any GPU event handling or GPU resource consumption. Our hardware engine prototype is implemented on FPGA, which we present to show the potential benefit of hardware deployment over the software engine. We explain the details of DMA engine implementations in Section 4.1.

3.2 Loop Kernel & Virtual CTA

GPU-controlled DMA engines would be easily adopted by existing systems, e.g., NCCL can replace its MMIO with initiating our DMA engines. However, existing systems would not fully exploit the benefit of GPU-controlled communication as the communication APIs are launched by CPU – the CPU intervention barrier still remains between computation and communication.

To remove this barrier, we propose a GPU-driven code execution system that runs an entire DL application in a single kernel, called a *loop kernel*. Our key observation is that online dynamic scheduling is unnecessary as DL workloads are typically deterministic at runtime. Instead of dynamically launching GPU kernels with CPU at runtime, our GPU-driven system automatically merges all kernels into a loop kernel (one for each GPU) at compile time and launches it only once at application start. Then, the loop kernel runs continuously during the entire lifetime of the application. A loop kernel is generated by our code generator that reads an operational graph of a DL application and automatically assembles corresponding code snippets of GPU operators to build loop kernel code. We call this code generation as *offline scheduling* as all GPU operators are statically distributed across GPU cores, or *Streaming Multiprocessors* (SMs), by the code. Offline scheduling lets GPUs efficiently control the application, which would minimize the event handling overhead for inter-GPU communication. We discuss several technical details of the loop kernel in Section 4.2.

Figure 4 shows that the loop kernel design deviates from the conventional framework for declaring, scheduling, and executing GPU tasks. In both CPU- and GPU-driven systems, a GPU operator is commonly defined as a set of multiple unit operators that each computes a part of the entire output in the SIMD manner. Meanwhile, both systems declare the operator differently in the GPU code. The CPU-driven system declares each unit operator as a Cooperative Thread Array (CTA)[8] and the entire operator as a separate kernel, which requires launching multiple kernels for multiple operators. In contrast, our GPU-driven system disallows multiple kernels as it executes all operators in the single loop kernel. Instead, it exploits intermediate declaration of unit operators that are scheduled as part of the CTAs of the loop kernel, which we call *virtual CTAs* (vCTAs).

vCTA provides the key abstraction for offline scheduling in ARK, which enables *software-defined* SM scheduling. A vCTA declares the code for a unit operator that is affinitized

[8]CTA is conceptually and functionally the same as a thread block in CUDA or a workgroup in OpenCL.

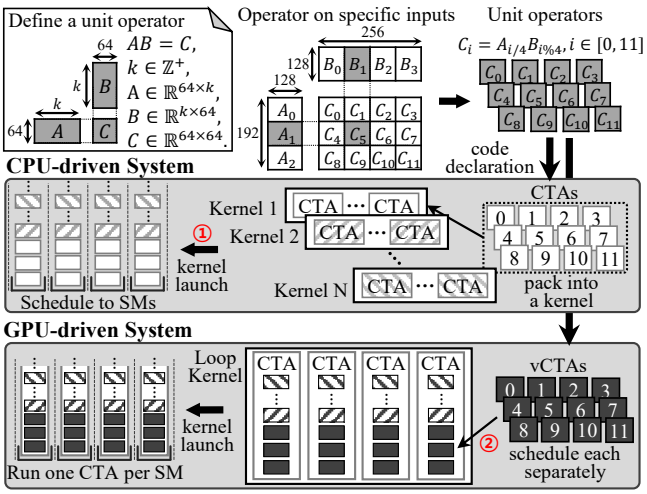


Figure 4: Comparing the procedures for declaring, scheduling, and executing GPU computation tasks between CPU- and GPU-driven systems. For instance, the figure shows a matrix multiplication operator with a 192x256 output, which is split into 12 unit operators that calculate 64x64 outputs each.

to a specific SM inside the loop kernel. While a CPU-driven system relies on the non-programmable hardware scheduler that distributes the CTAs across SMs at kernel launch (① in Figure 4), a GPU-driven system implements a custom logic that distributes vCTAs across CTAs (② in Figure 4). By launching one CTA per SM that assigns each CTA to use the entire resources of an SM, ARK can control the SM-affinity of vCTAs in a programmable manner. This enables fine-grained GPU scheduling, which is useful for the GPU-driven system to implement various computational optimization techniques such as *operator fusion* [17, 24, 35].

Migration of existing code to ARK is straightforward as ARK can reuse existing GPU kernel implementations with minimal modification: replacing the CTA ID (`blockIdx` in CUDA), thread ID (`threadIdx` in CUDA), SM-local memory address (shared memory in CUDA), and synchronization functions (e.g., `__syncthreads()` in CUDA) into corresponding constants or functions provided by the ARK framework. This modification guarantees the correctness of the framework which we have extensively verified.

As shown in Figure 4, offline scheduling writes a code snippet of each vCTA inside the if-branch of the loop kernel that only a particular CTA (or SM) enters. Since each CTA statically executes specific vCTAs that are planned offline, the GPU actually runs a static `while()` loop rather than being controlled dynamically – internal busy-polling loops inside vCTAs handle runtime events. For example, in Figure 5b, each of CTA 0 (`ctaId` is 0) and CTA 1 (`ctaId` is 1) are assigned three vCTAs from the operator `op_0` and two vCTAs from the operator `op_1`. Each CTA uses 256 threads, and vCTAs from `op_0` are executed sequentially by thread 0∼127, while tasks from `op_1` are executed by thread 128∼255 (which im-

```
__device__ void op_0(int vcta_id) {
  Add<...>(&BUF[1024], &BUF[9728], &BUF[1024], vcta_id);
}
__device__ void op_1(int vcta_id) {
  Matmul<...>(
    &BUF[11776], &BUF[9728], &BUF[16384], vcta_id);
}
```

(a) Operators.

```
__global__ void loop_kernel(volatile int *iter) {
  for (;;) {
    // Wait until iteration is requested by the host.
    if (threadId == 0) { while (*iter == 0) {} }
    __syncthreads();
    // Run iterations.
    for (int i = 0; i < *iter; ++i) {
      if (ctaId == 0) {
        if      (threadId < 128) { op_0(0); op_0(2); op_0(4); }
        else if (threadId < 256) { op_1(0); op_1(2); }
      } else if (ctaId == 1) {
        if      (threadId < 128) { op_0(1); op_0(3); op_0(5); }
        else if (threadId < 256) { op_1(1); op_1(3); }
      } ...
    }
    // Inform the host that iterations are done.
    if (threadId == 0) { *iter = 0; }
    __syncthreads();
  }
}
```

(b) Loop kernel.

Figure 5: Example of auto-generated code by the ARK scheduler. Note that the code is simplified for readability.

plies that each vCTA is implemented to use 128 co-working threads). Each vCTA is declared by passing a certain vCTA ID to a GPU function that defines an operator like in Figure 5a. The kernel code library of ARK provides the implementation of common operators (Add or Matmul in the figure) that take the addresses of data chunks and a vCTA ID as runtime arguments.[9] The framework assigns proper offsets to the global GPU buffer (`BUF`) for each data chunk, and the vCTA ID locates a specific part of the chunk that the vCTA deals with.

3.3 Offline Scheduler

Figure 6 shows the scheduling workflow in ARK. Overall, it reads the DAG of a DL model and generates the corresponding loop kernel code. The ARK scheduler is composed of a high-level scheduler and a profiling module. The high-level scheduler implements operator fusion with profiling results fed by the module. In the initial phase, it builds an *OpGraph* that spots all operators and their dependencies in the model, and generates the code to profile all types of vCTAs that are needed. Then, the high-level scheduler generates its first scheduling decision with the profiling results. The decision may consist of multiple different candidates that need to be profiled to choose the fastest one, then it iterates the overall process to compare against multiple other candidates, which may require additional profiling. The scheduler finally returns the loop kernel when only a single candidate remains.

Reducing compilations in the profiler. Since the code generator conducts deterministic scheduling with static vCTA-SM affinity, it can accurately estimate the performance (i.e. latency and core resource usage) of every scheduling decision by only profiling the performance of vCTAs, which reduces

[9] Other arguments such as input data sizes can be fixed during compilation by passing as template arguments, which we omit here.

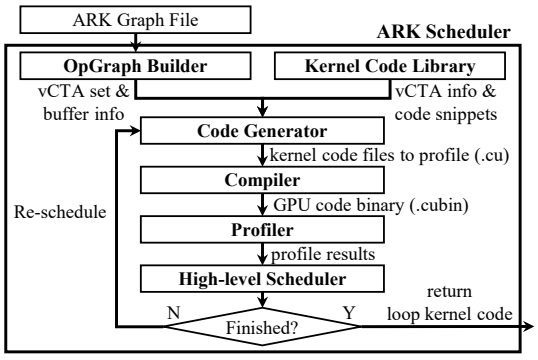


Figure 6: The ARK scheduler workflow.

the compilation for evaluating scheduling strategies. Say there are n parallel operators and each operator has m different implementations of the unit operator (or vCTAs),[10] then up to $O(m^n)$ different kernels should be compiled to find the best fusion decision. Since this number could be unreasonably large, existing works have developed heuristics to focus on only promising candidates [17].

At first glance, this appears to require only $O(nm)$ kernels for vCTA evaluations, but it is more complicated as vCTAs often complete faster when they are run concurrently on the same SM than when executed serially, which we say they have *joint efficiency*. Joint efficiency arises largely due to two causes: (1) because the L1 cache hit ratio improves as they access the common memory space running on the same SM, (2) because the execution of one vCTA hides the memory access of another (and vice versa) that improves simultaneous utilization of ALUs and LSUs. The first case is often found in the vCTAs from the same operator, while the second case is prevalent in most vCTAs, i.e., almost all vCTAs have the joint efficiency with each other.

Considering the joint efficiency, in general, we need to measure the latency when different types of vCTAs co-run on the same SM, which requires one kernel compilation for each. Say up to k vCTAs can run simultaneously in one SM, then the complexity of the number of compilations is $\sum_{i=1}^{k} \binom{n}{i} m^i = O(n^k m^k)$. In practice, this is much smaller than $O(m^n)$ because k is typically a small constant ≤ 4 due to the limitation of SM resources (# of maximum threads, bytes of shared memory, and # of registers).

SM load balancing in the code generator. The code generator automatically maximizes the SM utilization of the loop kernel by distributing parallel vCTAs across SMs to balance their workload. Unfortunately, finding the optimal load balancing is an NP-hard problem due to the joint efficiency. A brute-force searching would take unreasonably long due to the large number of vCTAs to schedule simultaneously.

To tackle this issue, we implement a heuristic load balanc-

[10] It is common to implement multiple different unit operators for the same operator, e.g. cuBLAS [30] implements at least 8 different-sized unit matrix multiplications and choose one depending on the input sizes.

ing on SM by leveraging an existing *graph partitioning* algorithm. Graph partitioning is a popular load balancing problem that splits a graph into a given number of subgraphs by cutting several edges, while achieving two goals: (1) balancing the total node weights of subgraphs and (2) minimizing the total weights of cut edges. We represent the SM load balancing problem into a graph partitioning problem. Specifically, we first group independent vCTAs that need to be distributed across SMs. Each group is represented as a graph where each node represents a vCTA and each edge indicates that the connecting nodes (i.e. vCTAs) have joint efficiency. The node weight is the latency of running the vCTA on an SM, and the edge weight measures joint efficiency, which is calculated as the fraction of latency reduction when we run both vCTAs simultaneously in the same SM compared with when we run both sequentially.

However, it takes too long to run the partitioning because it makes too many edges – since almost all vCTAs have joint efficiency with each other, the graph becomes nearly a mesh connection. To accelerate the algorithm, we adopt *hypergraph* representation [2] instead of an ordinary graph, which represents an equal-weighted mesh connection of multiple nodes as a single edge called *hyperedge*. Fortunately, this representation substantially reduces the time for code generation especially when we use a large batch size (which creates a lot of vCTAs), from tens of hours to only several seconds.

3.4 Limitations

The vCTA-based scheduling takes a whitebox approach that assumes all operators to be open-sourced, thus ARK cannot schedule close-sourced binaries such as cuDNN [28] (similar to Rammer [24]). Also, the offline scheduler of ARK only supports static computational graphs, which is less flexible comparing to e.g. PyTorch's dynamic graph [13]. However, such a limitation is commonly found in many popular frameworks including TensorRT [35] and ONNX Runtime [25].

4 Implementation

This section describes technical details of ARK.

4.1 DMA Engine Implementations

We first present our DMA engine interface, and then introduce our software and hardware DMA engines.

4.1.1 Interface

The key consideration of our interface design is ensuring high communication performance while keeping the interface consistent across software and hardware platforms. One key issue lies in the design of a DMA request message from GPU, which we call a *send request* (SR), as it has significant impact on the performance and the implementation complexity. In terms of hardware, receiving a large SR whose size exceeds the data bus width (64 bits in modern 64-bit processors) will take multiple cycles, which would require SR buffer management, reassembly of segmented SRs, and handling dropped SRs (caused by SR buffer overflow). Implementing them on hardware would significantly complicate the logic and increase the spatial cost. As implementing them on hardware would significantly complicate the logic and increase the spatial cost, we share an 8-byte SR design for both software and hardware engines. While it is challenging to hold the metadata of a general memory copy (two addresses and a copy length) within 8 bytes, we address this by adopting a small number of send/recv buffers, which reduces the address space by replacing general 8-byte addresses with a few bits of buffer indices. This is feasible thanks to the static nature of collective communication where the communicating entities are fixed – it enables offline pre-scheduling of data transfers so that receivers know which data arrives at which buffer without any additional metadata received at runtime. Meanwhile, the DMA requests on different buffers are pipelined for low latency and high throughput.

In terms of software, keeping an SR buffer would be more efficient as it would otherwise require extra control to prevent overwriting a previous SR. That is, unlike a hardware implementation where a fully received SR can immediately trigger the internal DMA pipeline at every cycle, a software thread could overwrite an unread SR unless the sender (GPU) coordinates with the receiver (the DMA stack) prior to sending a new SR. Unfortunately, such coordination would incur an extra delay as the GPU needs to read a remote flag on the DMA stack before sending an SR. We address this issue by maintaining a specialized ring buffer for SR, where the GPU checks only a local replica of the buffer head before sending an SR, and the replica is asynchronously updated by the DMA stack. This removes the coordination delay from the critical path of communication while providing a consistent SR interface for both software and hardware engines.

4.1.2 Software Engine

Our software engine harnesses CPU as the data plane while GPU serves as the control plane. We implement a CPU thread that busy-waits for SRs and invokes `cudaMemcpy` or RDMA writes accordingly, i.e., it leverages the existing hardware DMA engine on the sender GPU. Note that this is different from CPU-controlled communication as we use CPU only for data plane operations while the control plane (event handling) is managed by GPU threads. For high throughput, the busy-waiting loop drains all SRs in the ring buffer and invoke copy once for sending on a continuous memory space. Also, instead of slow `cudaEvent`, we use MMIO for the CPU-GPU communication that delivers SR, SC (Send Completion), and RC (Receive Completion) signals, which takes only 2∼3µs.

Alternatively, the software engine can perform MMIO with CPU threads instead of initiating the hardware DMA engine, which can reduce the `cudaMemcpy` overhead (i.e., sending a copy request from CPU to the DMA engine on GPU). However, this approach fails to achieve the line rate in most host

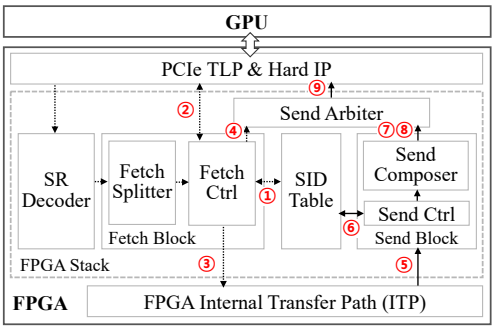


Figure 7: Implementation of the hardware DMA engine.

CPU architectures due to their poor throughput of crossing the PCIe root complex [41, 44]. This issue might be resolved in the future CPU architectures or by leveraging ARM cores on SmartNICs [4], which is left as our future work.

4.1.3 Hardware Engine

We implement a custom hardware with FPGA for DMA operations, which delivers two benefits over our software engine prototype. First, our hardware engine avoids the extra communication delay incurred by the overhead of `cudaMemcpy` as it performs DMA directly. Second, unlike existing hardware DMA engines on GPU, our custom hardware implements pipelining of multiple parallel DMA operations. This helps achieve a high data rate even for sending small data chunks. Table 2 shows resource usage of our implementation on an Intel Arria 10 FPGA.

Note that our FPGA prototype is limited to support the communication between only two GPUs and it does not support NVLink as there is no programmable hardware (or an off-the-shelf device) that can connect to NVLink. Instead, we consider it as a proof-of-concept that demonstrates the ideal benefit rather than a practical device that can be deployed on a large scale. A more practical implementation would be realized by future advances in CPU, GPU, or SmartNICs.

Figure 7 shows the hardware structure of inter-GPU communication stack on the FPGA. Unlike the existing GPU DMA engine, our DMA stack is designed to pipeline multiple DMA requests with different SIDs to be handled simultaneously. This is implemented by splitting a long-length request into multiple short-length sub-requests, which prevents head-of-line blocking and improves the PCIe throughput when GPU sends multiple different data at the same time. We explain how each request is processed by the sender- and the receiver-side stacks, respectively.

Sender side. When the sender stack receives an SR, the Fetch Block reads the decoded SR and retrieves the requested SID, which is translated into the physical source GPU address by looking up the SID Table (①). Using the address, the Fetch Ctrl fetches one sub-request at a time and it may fetch multiple times if the copy length is long. Each sub-request reads the corresponding source data from the GPU and stores it in a FIFO buffer of the Fetch Ctrl (②). When the source data is fully read from the GPU, the stored data and the sub-request are forwarded to the receiver stack through FPGA Internal Transfer Path (ITP). (③). After processing all sub-requests out of an SR, the Fetch Ctrl gives an SC flag to the Send Arbiter, which will be written on the GPU-side SC flag. (④).

Receiver side. The receiver stack receives the sub-request from the sender stack and stores the data into a FIFO buffer of the Send Ctrl (⑤). At the same time, the SID information in the sub-request is translated into the physical destination GPU address (⑥). The Send Ctrl sends the data to the destination address, and when it is done, the Send Composer sends an RC flag to the Send Arbiter, which will be written on the GPU-side RC flag (⑦, ⑧).

Module Name	ALMs		BRAM Blocks	
	#	Capacity	#	Capacitry
FPGA Stack	14253	3.34%	188	6.93%
PCIe	1364	0.32%	13	0.48%

Table 2: Resource usage of a single DMA stack.

Resource usage and limitations. We implement the DMA stack on Intel Arria 10 FPGA [16]. Table 2 shows that each stack is implemented at a low cost, using only 14253 ALMs and 188 M20K BRAMs. Note that our current implementation supports communication between only two GPUs by directly connecting the FPGA ITP interfaces of their corresponding FPGA stacks. Our design considers leveraging DUA [41] to support routing between multiple stacks (either intra- or inter-machine), but we leave it as future work.

4.2 Loop Kernel Implementation

This section explains several details of optimizing the loop kernel performance in ARK.

Per-thread register optimization. GPU kernels often fine-tune the number of concurrent threads per SM by evaluating the trade-off between running more threads (gain more parallelism) vs. running fewer threads with more registers per each (gain more computational throughput per thread). So, the loop kernel also needs to tune it. The ARK scheduler generates multiple versions of the loop kernel with a different number of per-thread registers and picks the best-performing one. Actually, in NVIDIA GPUs, only 32, 64, 128, and 256 are available candidates due to hardware limitation.

Dependency on GPU Architecture. Section 3.2 explains that ARK launches one CTA per SM, but it may launch two or more CTAs per SM depending on the GPU architecture. This is because one CTA may be limited to utilize the entire resources of an SM in some architecture. In such cases, we need to launch two CTAs per SM to use the entire SM resources. The ARK scheduler automatically analyzes the resource requirement of the loop kernel and determines the number of CTAs per SM accordingly.

Program size. We reduce the program size of a loop kernel by coalescing multiple identical unit operators, e.g., if

a model consists of many convolution operators, only several unique implementations of convolution will be actually defined, which are shared across all operators. Thus, the program size depends only weakly on the number of operators in the model. Instead, it is subject to the aggregate size of operator implementations, which is very limited – e.g., cuBLAS provides only ∼10 instances of a matrix-multiplication implementation on a single GPU architecture, while a loop kernel can accommodate over 5000 instances. This should cover an arbitrary DL program as the size of the matrix-multiplication implementation is one of the largest among the popular operators in DL.

5 Evaluation

We evaluate ARK by comparing it with existing DL frameworks largely in three different aspects. First, the fast inter-GPU communication of ARK contributes to higher end-to-end throughput and lower latency of DL applications. Second, the benefits on communication are obtained without losing the computational throughput of GPU. Third, ARK has flexibility to support various parallelism strategies including data-, tensor-, and pipeline-parallelism.

5.1 Experiment Setup

Software Engine. For experiments that use the software DMA engine, unless specified differently, we use two Intel Xeon Gold 6240R CPUs (48 lcores each, 2.40 GHz) and eight NVIDIA V100 GPUs. We have two NUMA nodes in the machine but only a single NUMA node hosts all GPUs, i.e. node 0 connects two PCIe v3 switches to its PCIe root complex and each switch is directly connected to 4 GPUs. For multi-node experiments, we use four Azure NDv4 SKUs [7] with 32x NVIDIA A100 GPUs in aggregate (8 per node), where each GPU has dedicated 200 Gbps NVIDIA Mellanox HDR InfiniBand connection.

Hardware Engine. For experiments that use the hardware DMA engine, we use an Intel Xeon Gold 5118 CPU (24 lcores, 2.30 GHz), two NVIDIA V100 GPUs, and an Intel Arria 10 FPGA. Both GPUs and the FPGA are behind the same PCIe v3 switch. We use the hardware engine only for experiments in Section 5.2 and Section 5.5.

5.2 DMA Engine Performance

Figure 8 compares the performance of communication between two GPUs with our DMA engines (G-Drv-S and G-Drv-H) over a CPU-controlled communication baseline (C-Drv). C-Drv is our own minimal implementation of a typical CPU-driven system, but unlike TensorFlow, C-Drv leverages asynchronous control using `cudaEvent` when the event is used only by GPUs, which further reduces CPU-GPU synchronizations to accelerate inter-GPU communication.

We measure the throughput by sending many parallel messages at the same time and reporting the maximum throughput

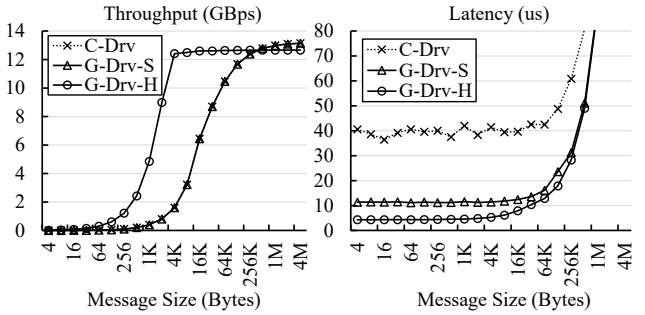


Figure 8: Performance comparison between the CPU-controlled communication (C-Drv) and the GPU-controlled DMA engines (G-Drv-S (software) and G-Drv-H (hardware)) over PCIe v3.

achieved with varying message sizes. For latency measurements, we implement a ping-pong application and report one-way latency – unlike throughput measurements, this includes communication event handling delays. This experiment assumes a favorable scenario for the CPU-controlled baseline where we can adopt the asynchronous control (explained in Section 2.2.2). In this scenario, a one-way trip requires triggering only two GPU events and two stream synchronizations.

In the left graph of Figure 8, our software engine (G-Drv-S) shows the same throughput as that of C-Drv, since both use `cudaMemcpy` for the data-plane. In contrast, our hardware engine (G-Drv-H) shows huge throughput improvement, saturating the bandwidth with only 8 KB messages while G-Drv-S needs 4 MB messages for saturation. This is because the hardware DMA engine pipelines processing multiple DMA requests while `cudaMemcpy` cannot. This improvement would be especially beneficial when GPU sends multiple messages to different destinations at the same time, e.g., all-to-all communication for expert-parallelism, which is popular for scaling out state-of-the-art Transformer-based models [11].

We note that the maximum achieved throughput of G-Drv-H is 3.68% lower than G-Drv-S. This is because an external DMA stack needs to send both read and write requests to sender and receiver GPUs, respectively, while the native DMA engine on the sender GPU needs to send only write requests. However, as the gap is small, it would not affect the end-to-end application performance much.

The right graph of Figure 8 shows that the one-way latency of C-Drv is at least ∼39.3µs on average. In contrast, G-Drv-S and G-Drv-H achieve 3.5x and 9.1x better latency, respectively. This is because our DMA engines handle the communication events directly in GPU threads while C-Drv relies on the `cudaEvent` interface that suffers from large overhead to trigger the events and synchronize streams. This improvement would be especially beneficial when GPUs perform split-and-gather of intermediate results to distribute the workload, as in tensor-parallelism [22, 26]. One thing to note about our DMA engine is that the benefit is obtained with little GPU cycle consumption. We evaluate this in the following section.

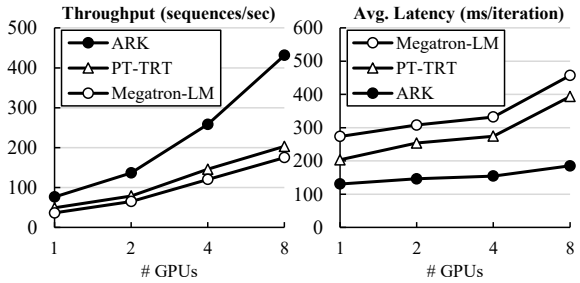


Figure 9: BERT-Large data-parallel training throughput and average latency per iteration with varying numbers of GPUs (sequence length 384, batch size 10, mixed-precision).

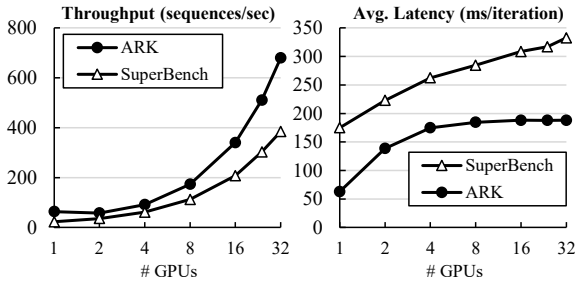


Figure 10: GPT-2 data-parallel training throughput and average latency per iteration with varying numbers of GPUs (sequence length 384, batch size 4, mixed-precision).

5.3 Avoiding Communication Interference

To compare the interference between computation and communication of using NCCL against using our DMA engine, we evaluate data-parallel training throughput of ARK by training representative NLP models.

Baselines. PT-TRT accelerates PyTorch [12] by adopting TensorRT [35], which does not scale out to multiple machines. Megatron-LM [26] is a PyTorch-based framework that supports large-scale training of NLP models but we use only for single-node experiments here. SuperBench [42] provides formal DL benchmarks for system performance evaluation also based on PyTorch, which we use for multi-node experiments. All baselines leverage NCCL [32] for communication.

Single Node. Single-node experiments train BERT-Large [10] model using up to 8x V100 GPUs as shown in Figure 9. The figure shows that ARK outperforms Megatron-LM and PT-TRT respectively by **2.46x** and **2.12x** with 8 GPUs. We find two reasons for the speedup.

First, NCCL adversely affects the computational throughput during back-propagation while ARK does not as it leverages DMA instead of employing GPU threads for data copy. Specifically, 64.5% of the end-to-end gap between ARK and PT-TRT with 8 GPUs is obtained as NCCL operations slow down due to the interference of MMIO with back-propagation computation, showing only 5.0 GBps of all-reduce throughput. We find that NCCL kernels result in 45.0% of slowdown of the overall back-propagation computation, an increase from 107.63 ms to 156.02 ms. On the other hand, our DMA engine suffers near-zero interference by initiating DMA directly instead of using MMIO, achieving 9.10 GBps of all-reduce throughput (**1.82x** faster).

Second, ARK performs more efficient computation on GPU. For example, for about 37.8% of the computation time of PT-TRT, it executes 1.2 thousands of memory-intensive kernels per iteration, such as element-wise arithmetic or intra-GPU data movement. Running these operators as separate kernels would be inefficient because it would incur unnecessary kernel launches and intra-GPU synchronizations. ARK largely reduces such overhead as it schedules all operators in a single loop kernel, similar as operator fusion [17, 24, 35].

Multiple Nodes. Multi-node experiments train the GPT-2 [36] model using up to 32x A100 GPUs as shown in Figure 10. All results use only InfiniBand for communication (no NVLink) and use the ring reduction algorithm. The figure shows that ARK outperforms SuperBench by **1.77x** with 32 GPUs. Furthermore, while per-iteration latency of SuperBench is consistently increasing, the increment in ARK is only marginal. This shows the efficiency of our communication stack over NCCL, which minimizes the interference between communication and computation. We also find a big computational benefit of ARK even without communication (when using a single GPU), which is further explained in the following section.

5.4 Offline Scheduling Evaluation

This section shows that the offline scheduler of ARK can generate comparable or even better GPU kernels comparing with existing DL optimization techniques. Rather than claiming state-of-the-art performance in DL optimization, we intend to show that the communication gain of our GPU-driven system does not come up with any computational performance drop.

We compare the inference performance of popular DL models over different frameworks using a single GPU. The DL models include image classification (ResNet-50 [14] and GoogLeNet [43]), object detection (SSD [23]), and NLP (BERT-Large [10]) models. TensorFlow (TF) is the primary comparison target of ARK because it supports flexible parallelism for DL applications like ARK. We also compare with TensorFlow-XLA (TF-XLA) [1] that implements automatic operator fusion in the TF back-end, but it is not always beneficial to the performance because the fused kernel might perform worse than using vendor-provided kernels (e.g. cuDNN) without fusion. Rammer [24] and TensorRT implement optimized operator fusion that often outperforms TF or TF-XLA, but they support only limited parallelism. For example, TensorRT supports only intra-node data-parallelism by adopting it to accelerate other frameworks like TF and PyTorch, as TensorRT itself does not support distributed execution. Nimble [20] presents careful asynchronous control (or ahead-of-time scheduling) of GPU kernels to reduce runtime

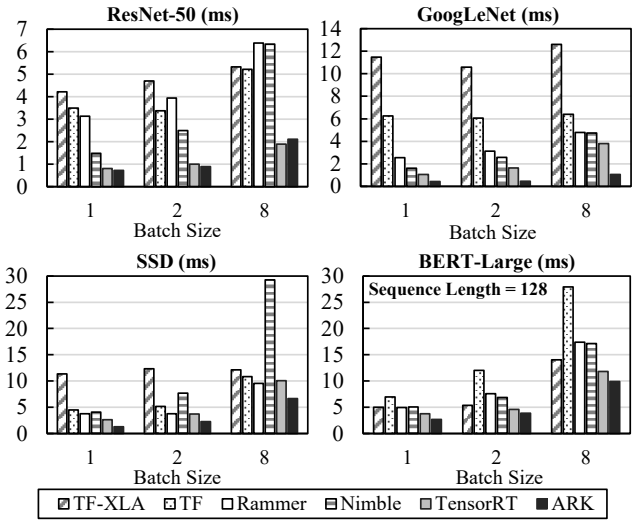


Figure 11: Inference latency comparison of popular DL models over different DL frameworks using a single GPU. All experiments use mixed-precision computation.

overhead of kernel launch and GPU events. As explained in Section 2.2.2, however, asynchronous control is limited to tackle the communication overhead. Nimble also works only on a single GPU at the moment.

Figure 11 shows that ARK achieves faster single-GPU inference against existing frameworks in most cases. For instance, ARK shows **1.11x∼3.56x** lower latency than TensorRT, except the case of ResNet-50 with batch size 8 that is ∼9.90% worse than TensorRT. This is because our matrix multiplication kernel is slower than the cuDNN [28] kernel used in TensorRT in this case (note that we implement convolution via matrix multiplication). ARK currently does not implement vCTAs specialized for large matrix multiplications (one side of the unit operator's output is larger than 256 elements), so it is often slower than existing kernels when the model consists of large matrix multiplications.

We note that the gain of ARK is especially large when the model consists of many parallel operators like GoogLeNet or SSD. This is because our high-level scheduler maximizes overall SM utilization by choosing the best vCTA (or unit operator) for each parallel operators. Specifically, when a lightweight operator runs alone in the GPU, we schedule it to use fine-grained vCTAs so that it utilizes more concurrent SMs. In contrast, when the GPU is overloaded due to other co-running operators, we need to use coarse-grained vCTAs to utilize SMs more efficiently. This is because coarse-grained vCTAs work on more input data at the same time and thus have more opportunities to better utilize the parallelism in an SM. As explained in Section 3.3, the optimization to find the best-performing vCTAs is easy in the ARK framework because it accurately estimates the performance with different vCTAs without running all candidates. We note that other frameworks do not provide a similar optimization like this.

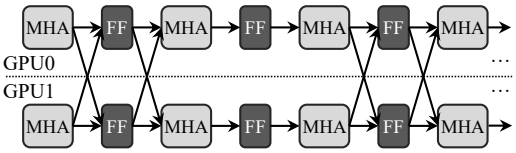


Figure 12: MoE model-parallel execution for Transformer architecture using 2 GPUs, composed of MHA (multi-headed attention) and FF (feed-forward) modules.

Architecture	Message Size (KB)	Time Gap (us)
BERT-Large [10]	256	60.9
GPT-3 XL [8]	512	187.4
T5 3B [37]	256	166.9
M4 [6]	256	60.9

Table 3: The message size and the smallest time gap between transactions for MoE inference. The input sequence length is 128. Time gaps are measured using the ARK framework.

5.5 Tensor-parallel Inference

This section presents the latency improvement with the tensor-parallel approach called mixture-of-experts (MoE) that efficiently scales up the Transformer [45] architecture, which is commonly used in many popular NLP models [6, 8, 11, 37]. This method is suggested to scale NLP models to one trillion of model parameters [11,22], but since we do not have enough GPUs to run the entire model, we evaluate the tensor-parallel inference of the model using two GPUs. In real practice, this is replicated to other GPUs to apply pipeline-parallelism (for training or inference) and data-parallelism (only for training) as well at the same time.

Figure 12 illustrates the MoE execution. The message size and the smallest time gap in-between the exchanges depend on the model hyperparameters, and some examples are shown in Table 3. Even though we present only 2-GPU experiments here, the result would be similar to a larger-scale one because MoE is designed to send each message only up to a small constant number (e.g. two in GShard [22]) of selected GPUs, not to all other GPUs.

We evaluate ARK using the hardware engine with three different comparison baselines – TF, TF-XLA, and C-Drv. Note that TensorRT-accelerated TensorFlow (TF-TRT) does not support model-parallelism, so it is not evaluated here.

Results in Figure 13 shows that ARK outperforms TF and TF-XLA by **1.66x∼3.48x** and **1.25x∼2.31x**, respectively. In terms of only the communication latency, ARK reduces it by 3.68x∼5.65x and 1.77x∼3.31x, respectively. Overall, C-Drv achieves better communication latencies over TF or TF-XLA, but its computation is less efficient because it reuses GPU kernel implementations in ARK but it does not benefit from ARK scheduler optimization. We also find that the GPU-driven communication of ARK delivers a substantial speedup over the CPU-driven communication of C-Drv, as shown in Section 5.2. We note that ARK computation is slower than

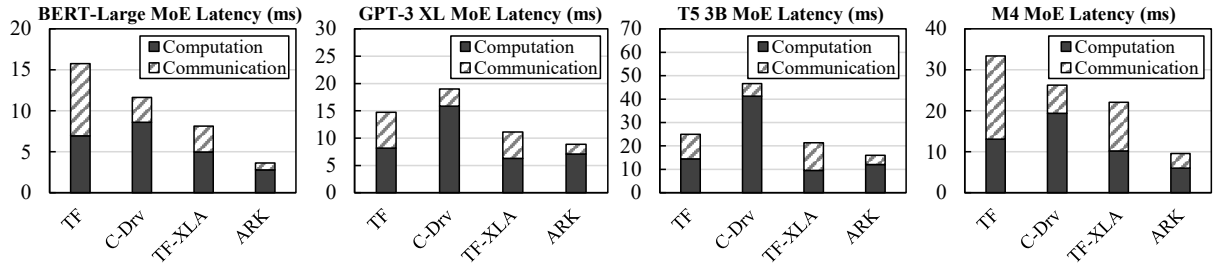


Figure 13: MoE inference latencies with different NLP model architectures (batch size 1, mixed-precision).

TF-XLA in GPT-3 XL and T5 3B. This is because our matrix multiplication kernel performs worse than TF-XLA in these cases, as explained in Section 5.4.

5.6 Pipeline-parallel Training

In this section, we train the GPT-3 [8] 6.7B model, which is the largest variation of GPT-3 that can fit the memory of eight V100 GPUs via pipeline-parallel training. The model consists of 32 sequential layers and each GPU trains 4 layers in the sequential order – GPU 0 reads the input data and runs the forward-pass of layer 0∼3, and the 16 MB output is passed to GPU 1, and so on. When GPU 7 completes the forward-pass, it moves on to the backward-pass of layer 31∼28, and the 16 MB of back-propagating gradient is passed to GPU 6, and so on. We use the mixed-precision computation and set the number of pipeline stages to 5, the batch size of each stage to 1, and the sequence length to 2048. ARK uses the emulated DMA stack in this evaluation.

In this experiment, the training throughputs of TF, TF-XLA, Megatron-LM, and ARK are 0.35, 0.47, 1.69, and 2.38 sequences per second, respectively, i.e. ARK outperforms TF, TF-XLA, and Megatron-LM by 6.80x, 5.06x, and 1.40x, respectively. In this case, most of the improvement of ARK comes from the computational efficiency on GPU, as pipeline-parallel training typically overlaps most of the communication delay with the computation time. This evaluation shows that ARK delivers the gain of operator fusion while supporting flexible parallelism for DL.

6 Future Work & Related Work

We expect that hardware advances in near future would enable more efficient implementations. For example, implementing our software DMA engine on SmartNIC would avoid the throughput issue of the PCIe root complex [44] via direct PCIe connection with GPUs (e.g., NVIDIA H100 CNX [9] combines GPU with SmartNIC), which enables efficient MMIO on SmartNIC. NVIDIA has announced their hardware accelerators for inter-GPU communication on SmartNICs (e.g., all-to-all engine on NVIDIA BlueField-3 [29]), which implies that a similar implementation with our hardware engine might be realized in the future. Additionally, host CPU architectures in the future may fix the root complex issue, which will enable our software DMA engine to replace `cudaMemcpy` with CPU-side MMIO, or even more efficiently, DMA engines on CPU (e.g., Intel I/OAT [15] or AMD PTDMA [21]).

ACE [38] proposes offloading the entire collective communication logic to a hardware accelerator that resides on intra-machine fabric, which cannot be extended to an external network (Ethernet, InfiniBand, etc). Our work differs from ACE as it is generally applicable to any (R)DMA networking and we can reuse most of existing software logic in popular collective communication libraries.

GPUnet [18] presents a network socket API set for GPU threads and leverages CPU intervention to let GPU threads to trigger DMA. This is inefficient as they add a substantial intervention overhead especially for small messages because they do not pipeline processing multiple DMA requests. Its throughput could be suboptimal as it implements a general socket interface on GPU while ARK reduces the overhead by leveraging offline scheduling to remove the metadata to be managed during runtime.

Nimble [20] accelerates DL execution by minimizing runtime scheduling overhead of kernels, but it works only on a single GPU. The proposed methods also cannot help reduce communication event handling overhead as it still relies on the CPU-side control using `cudaEvent` and multi-stream interfaces. ARK tackles this by letting GPU threads fully control all computation and communication tasks.

7 Conclusion

This paper envisions a GPU-driven code execution system that enables autonomous control of GPU throughout the entire lifetime of DL applications. We present the GPU-controlled DMA engine at the heart of the GPU-driven system that enables GPUs to communicate with each other without any external control. To avoid interference between computation and communication, we design our DMA engine and offline GPU scheduling to consume little GPU resources for communication, so that its high communication performance is delivered without sacrificing computational throughput of GPU. While our software engine already shows benefits over commodity hardware, we also present a proof-of-concept of a hardware engine that shows even higher performance, which indicates that our system performance would be further improved with future advances in commodity hardware such as CPU, GPU, or SmartNIC.

Acknowledgements

We appreciate the feedback by our shepherd, Danyang Zhuo, as well as anonymous reviewers of NSDI'23. This work is in part support by the ICT Research and Development Program of MSIT/IITP, Korea, under [2022-0-00531, Development of in-network computing techniques for efficient execution of AI applications] and [2018-0-00693, Development of an ultra low-latency user-level transfer protocol].

References

[1] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla, 2021. [Online; accessed Dec 2022].

[2] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct *k*-way hypergraph partitioning algorithm. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2017.

[3] AMD. Introducing AMD CDNA™ 2 Architecture. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf, 2021. [Online; accessed Dec 2022].

[4] AMD. Alveo SN1000 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html, 2022. [Online; accessed Dec 2022].

[5] AMD. ROCm Communication Collectives Library (RCCL). https://github.com/ROCmSoftwarePlatform/rccl, 2022. [Online; accessed Dec 2022].

[6] Naveen Arivazhagan, Ankur Bapna, Orhan Firat, Dmitry Lepikhin, Melvin Johnson, Maxim Krikun, Mia Xu Chen, Yuan Cao, George F. Foster, Colin Cherry, Wolfgang Macherey, Zhifeng Chen, and Yonghui Wu. Massively multilingual neural machine translation in the wild: Findings and challenges. *CoRR*, abs/1907.05019, 2019.

[7] Microsoft Azure. ND A100 v4-series - Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series, 2022. [Online; accessed Dec 2022].

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[9] Charu Chaubal. Build Mainstream Servers for AI Training and 5G with the NVIDIA H100 CNX. https://developer.nvidia.com/blog/build-mainstream-servers-for-ai-training-and-5g-with-the-nvidia-h100-cnx/, 2022. [Online; accessed Dec 2022].

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2019.

[11] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[12] The Linux Foundation. PyTorch. https://pytorch.org, 2022. [Online; accessed Dec 2022].

[13] The Linux Foundation. How Computational Graphs are Constructed in PyTorch. https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/, 2023. [Online; accessed Jan 2023].

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[15] Intel. Fast memcpy with SPDK and Intel® I/OAT DMA Engine. https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html, 2017. [Online; accessed Dec 2022].

[16] Intel. Intel® FPGAs - Intel® Arria® 10 FPGAs. https://www.intel.com/content/www/us/en/products/details/fpga/arria/10.html, 2022. [Online; accessed Dec 2022].

[17] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[18] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for GPU programs.

In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[19] Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andrés Felipe Cruz-Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. Scalable and efficient moe training for multitask multilingual models. *CoRR*, abs/2109.10465, 2021.

[20] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[21] Michael Larabel. AMD PTDMA Driver Landing For Linux 5.15 After Two Years In The Works – Phoronix. https://www.phoronix.com/scan.php?page=news_item&px=AMD-PTDMA-For-Linux-5.15, 2021. [Online; accessed Dec 2022].

[22] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020.

[23] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[24] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[25] Microsoft. ONNX Runtime. https://onnxruntime.ai/, 2023. [Online; accessed Jan 2023].

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters. *CoRR*, abs/2104.04473, 2021.

[27] NVIDIA. Using NCCL with CUDA Graphs. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/cudagraph.html, 2020. [Online; accessed Dec 2022].

[28] NVIDIA. CUDA Deep Neural Network (cuDNN). https://developer.nvidia.com/cudnn, 2021. [Online; accessed Dec 2022].

[29] NVIDIA. NVIDIA BlueField-3 DPU – Programmable Data Center Infrastructure On-a-Chip. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2021. [Online; accessed Dec 2022].

[30] NVIDIA. cuBLAS. https://developer.nvidia.com/cublas, 2022. [Online; accessed Dec 2022].

[31] NVIDIA. GPUDirect. https://developer.nvidia.com/gpudirect, 2022. [Online; accessed Dec 2022].

[32] NVIDIA. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl, 2022. [Online; accessed Dec 2022].

[33] NVIDIA. NVLink & NVSwitch: Fastest HPC Data Center Platform. https://www.nvidia.com/en-us/data-center/nvlink/, 2022. [Online; accessed Dec 2022].

[34] NVIDIA. PTX ISA – Cache Operators. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#cache-operators, 2022. [Online; accessed Dec 2022].

[35] NVIDIA. TensorRT SDK. https://developer.nvidia.com/tensorrt, 2022. [Online; accessed Dec 2022].

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[37] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[38] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. Enabling compute-communication overlap in distributed deep learning training platforms. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[39] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[41] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct universal access: Making data center resources available to FPGA. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[42] SuperBench. SuperBench Documentation. https://microsoft.github.io/superbenchmark/, 2022. [Online; accessed Dec 2022].

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[44] Nathan R Tallent, Nitin A Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. Evaluating on-node gpu interconnects for deep learning workloads. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2017.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing

Tianfeng Liu*[1,4,3], Yangrui Chen*[2,3], Dan Li[1,4], Chuan Wu[2], Yibo Zhu[3], Jun He[3], Yanghua Peng[3], Hongzheng Chen[3,5], Hongzhi Chen[3], Chuanxiong Guo[3]
[1]*Tsinghua University,* [2]*The University of Hong Kong,* [3]*ByteDance,*
[4]*Zhongguancun Laboratory,* [5]*Cornell University,*

Abstract

Graph neural networks (GNNs) have extended the success of deep neural networks (DNNs) to non-Euclidean graph data, achieving ground-breaking performance on various tasks such as node classification and graph property prediction. Nonetheless, existing systems are inefficient to train large graphs with billions of nodes and edges with GPUs. The main bottlenecks are the process of preparing data for GPUs – subgraph sampling and feature retrieving. This paper proposes BGL, a distributed GNN training system designed to address the bottlenecks with a few key ideas. First, we propose a dynamic cache engine to minimize feature retrieving traffic. By co-designing caching policy and the order of sampling, we find a sweet spot of low overhead and a high cache hit ratio. Second, we improve the graph partition algorithm to reduce cross-partition communication during subgraph sampling. Finally, careful resource isolation reduces contention between different data preprocessing stages. Extensive experiments on various GNN models and large graph datasets show that BGL significantly outperforms existing GNN training systems by 1.9x on average.

1 Introduction

Graphs, such as social networks [23, 36], molecular networks [19], knowledge graphs [21], and academic networks [47], provide a natural way to model a set of objects and their relationships. Recently, there is increasing interest in extending deep learning methods for graph data. Graph Neural Networks (GNNs) [22, 36, 46] have been proposed and shown to outperform traditional graph learning methods [50, 57, 59] in various applications such as node classification [36], link prediction [56] and graph property prediction [51].

Real-world graphs can be massive. For example, the user-to-item graph on Pinterest contains over 2 billion entities and 17 billion edges with 18 TB data size [53]. As a major online service provider, we also observe over 100 TB size of graph data, which consists of 2 billion nodes and 2 trillion edges. Such large sizes make it impossible to load the entire graph into GPU memory (at tens of GB) or CPU memory (at hundreds of GB), hence turning down proposals that adopt full graph training on GPUs [55]. Recent works [23, 28, 53] have resorted to mini-batch sampling-based GNN training, aggregating neighborhood information on sampled subgraphs.

Distributed systems [2, 17, 48] for this training typically include *distributed graph store servers* to store partitioned large-scale graphs and *worker machines* where each worker has one GPU for model training. Each training iteration contains three stages: (1) *sampling subgraphs* stored in distributed graph store servers, (2) *feature retrieving* for the subgraphs from graph store servers to workers, and (3) forward and backward *computation* of the GNN model.

The first two stages, which we refer to as *data I/O and preprocessing*, are often the performance bottlenecks in such sampling-based GNN training. After analyzing popular GNN training frameworks (*e.g.,* DGL [48], PyG [17], and Euler [2]), we made two key observations. (1) High data traffic for retrieving training samples: when the sampled subgraph is stored across multiple graph store servers, there can be frequent cross-partition communication for sampling; retrieving corresponding features from the storage to worker machines also incurs large network transmission workload. (2) Modern GPUs can perform the computation of state-of-the-art GNN models [22, 36, 46] quite fast, leading to high demand for data input. To mitigate these problems, Euler adopts parallel feature retrieval; DGL and PyG prefetch the sampling results. Unfortunately, none of them fully resolves the I/O bottleneck. For example, we observe only around 10% GPU utilization in a typical DGL training job on a large graph (§2 and §5), which means around 90% of GPU cycles are wasted.

In this paper, we propose BGL, a GPU-efficient GNN training system for large graph learning, to accelerate training and achieve high GPU utilization (near 100%). Focusing on eliminating data I/O and preprocessing bottlenecks, we identify three key challenges in the existing frameworks, namely: (1) very heavy network traffic for retrieving features, (2) large

*Tianfeng Liu and Yangrui Chen contributed equally to this work as first authors.

cross-partition communication overhead during sampling, and (3) resource contention between different training stages. We address those challenges, respectively.

The biggest bottleneck of distributed GNN training systems often lies in retrieving large features (§2.3). PaGraph [38], a state-of-the-art cache design for GNN training, uses a static cache (no replacement during training) and explicitly avoids dynamic caching policy (replacing some cached features at runtime) because of high overhead. However, we find that static cache has low hit ratios when the graphs are so large that only a small fraction of nodes can be cached. Hence, we co-design a dynamic cache policy and the sampling order of nodes. We show that a FIFO policy has acceptable overhead and high hit ratios combined with our *proximity-aware ordering*. The key idea is to leverage *temporal locality* – in nearby mini-batches, we always attempt to visit the neighboring training nodes in the graph. This approach largely increases the cache hit ratio of FIFO policy. We will further explain the details of how we ensure the consistency of our multi-GPU cache engine and GNN convergence in §3.2.

After optimizing feature retrieval, the cross-partition communication for subgraph sampling could become the major performance bottleneck. Existing algorithms either do not scale to large graphs or ignore *multi-hop neighbor* connectivity inside each partition. It leads to heavy cross-partition communication because, in GNN training, the sampling algorithm usually requests *multi-hop neighbors* from a given node. Hence, we design a graph partition algorithm tailored for the typical GNN sampling algorithms. Our algorithm (in §3.3.2) strives to maintain multi-hop connectivity in each partition, while maintaining load balance partitions and scaling to giant graphs.

Finally, data preprocessing in GNN training takes multiple stages and is much more complex than that in traditional DNN training. Execution of some stages may compete for CPU and bandwidth resources, throttling the performance. Existing frameworks largely ignore it and let the preprocessing stages freely compete with each other. Unfortunately, some stages do not scale well with more resources. They may acquire more resources than they need, leading to blocking other stages. Hence, we optimize the resource allocation of data preprocessing by profiling-based resource isolation. Our key idea is to formulate the resource allocation problem as an optimization problem, use profiling to find out the resource demands of each stage, and isolate resources for each stage.

We implement BGL, including the above design points, and replace the data I/O and preprocessing part of DGL with it. The design of BGL is generic – e.g., BGL can also be used with Euler's computation backend. However, our evaluation focuses on using BGL with the DGL GPU backend because it is more mature and performant. We conduct extensive experiments using multiple representative GNN models with various graph datasets, including the largest publicly available dataset and an internal billion-node dataset. We demonstrate that BGL outperforms existing frameworks, and the geometric mean of speedups over PaGraph, PyG, DGL, and Euler is 1.91x, 3.02x, 7.04x, and 20.68x, respectively. With the same GPU backend as DGL, BGL can push the V100 GPU utilization to 99% even when graphs are stored remotely and distributedly, higher than existing frameworks. It also scales well with the size of graphs and the number of GPUs.

2 Background and Motivation

2.1 Sampling-based GNN Training

We start by explaining sampling-based GNN training.

Graph. The most popular GNN tasks [1] are to train on graphs with node features, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$, where $\mathcal{V}$ and $\mathcal{E}$ denote the node set and edge set of the graph, and $\mathcal{F}$ denotes the set of feature vectors assigned to each node. For example, in the graph Ogbn-papers [47], each node (*i.e.*, paper) has a 128-dimensional feature vector representing the embeddings of the paper title and abstract. We assume graph structures and node features are *immutable* in this paper.

Graph neural networks (GNNs). Graph neural networks are neural networks learned from graphs. The basic idea is collectively aggregating information following the graph structure and performing various feature transformations. For instance, the Graph Convolution Network (GCN) [36] generalizes the convolution operation to graphs. For each node, GCN aggregates the features of its neighbors using a weighted average function and feeds the result into a neural network. For another example, GraphSAGE [23] is a graph learning model that uses neighbor sampling to learn different aggregation functions on different numbers of hops.

Real-world graphs, such as e-commerce and social networks [13, 53, 55], are often large. The Pinterest graph [53] consists of 2B nodes and 17B edges, and requires at least 18 TB memory during training. Even performing simple operations for all nodes would require significant computation power, not to mention the notoriously computation-intensive neural networks. Similar to other DNN training tasks, it is appealing to use GPUs to accelerate GNN training.

Sampling-based GNN training. There are two camps of training algorithms adopted in existing GNN systems: *full-batch training* and *mini-batch training*. Full-batch training loads the entire graph into GPUs for training [36], like NeuGraph [40] and ROC [31]. Unfortunately, for very large graphs like Pinterest's, such an approach would face the limitation of GPU memory capacity.

Thus, we focus on the other approach, *mini-batch training*, or often called *sampling-based GNN training*. In each iteration, this approach samples a subgraph from the large original graph to construct a mini-batch as the input to neural networks. Mini-batch training is more popular and adopted by literature [11, 23, 54] and popular GNN training frameworks like DGL [48], PyG [18] and Euler [2].

[1] We focus on node classification tasks in this work.

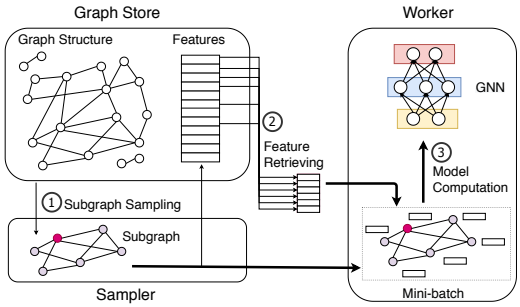


Figure 1: Sampling-based GNN training process.

The process of sampling-based GNN training is shown in Figure 1. The fixed graph data (including the graph structure and node features) are partitioned and stored in a distributed *graph store*. Multiple workers run on *worker machines*, with each worker equipped with one GPU. Each training iteration consists of three stages: ① **Subgraph sampling**: *Samplers* sample a subgraph from the original graph and send it to workers. ② **Feature retrieving:** After *workers* receive the subgraph, the features of its nodes are further retrieved from the graph store server and placed in GPU memory. ③ **Model computation:** Like typical DNN training, workers on GPU forward-propagate the prepared mini-batch through the GNN model, calculate the loss function, and then compute gradients in backward propagation. Then model parameters are updated using optimizers (*e.g.,* SGD [61], Adam [35]).

In the rest of this paper, we refer to the first two stages as *Data I/O and Preprocessing*.

2.2 Data I/O and Preprocessing Bottlenecks

Unfortunately, existing GNN training frameworks suffer from data I/O and preprocessing bottlenecks, especially when running model computation on GPUs. Here, we test two representative frameworks, DGL [48] and Euler [2]. We train GraphSAGE [23] model with one GPU worker. Using the partition algorithms of DGL and Euler, we split the Ogbn-papers graph [47] into four partitions and store them on four servers as a distributed graph store. More configuration details and the other framework results are in §5.

Figure 2 shows the training time of one mini-batch and the time breakdown of each stage. 87% and 82% of the training time were spent in data I/O and preprocessing by Euler and DGL, respectively. Long data preprocessing time leads to not only poor training performance but also low GPU utilization. The maximum GPU utilization of DGL and Euler is 15% and 5%, respectively, as shown in Figure 3.

In GNN training, such a bottleneck is much more severe than in DNN training like computer vision (CV) or natural language processing (NLP) for two main reasons.

First, due to the neighbor explosion problem [12, 54], the size of mini-batch data required by each training iteration is very large. For example, if we sample a three-hop subgraph from Ogbn-products with batch size 1,000 and fan out {15,10,5}, each mini-batch consists of 5MB subgraph structure (roughly 400,000 nodes) and 195 MB node features. Assuming that we use a common training GPU server like AWS p3dn.24xlarge [4] (8x NVIDIA V100 GPUs and 100Gbps NIC) as the worker, and that we could saturate the 100Gbps NIC pulling such data, we can only pull 60 mini-batches of data in every second.

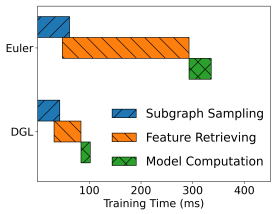


Figure 2: Training time per mini-batch of DGL and Euler.

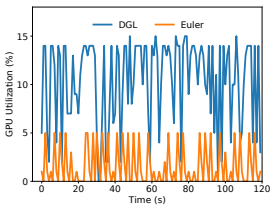


Figure 3: GPU utilization of DGL and Euler.

Second, the model sizes and required FLOPS of GNN are much smaller than classic DNN models like BERT [15] or ResNet [25]. V100 needs only 100MB and 20ms to compute a mini-batch of popular GNN models like GraphSAGE. P3dn.24xlarge can compute 400 mini-batches per second.

There is clearly a huge gap between the data I/O and preprocessing speed, and GPU computation speed. Consequently, though frameworks like DGL and Euler adopt pipelining, the data I/O and preprocessing bottlenecks can only be hidden by a small fraction and dominate the end-to-end training speed.

Some recent work [20, 29, 38] also observed this problem and made promising progress. Unfortunately, it still falls short in performance (§5) and cannot handle giant graphs well. Next, we will elaborate on the main challenges existing GNN training frameworks face.

2.3 Challenges in Removing the Bottlenecks

We identify three main challenges. Two are on large communication traffic for feature retrieving and subgraph sampling (as shown in Figure 1 and 2). The other is about resource contention when running all the stages together.

Challenge 1: Ineffective caching for node feature retrieving. As shown in Figure 2, due to the large volume of data being pulled to workers, node feature retrieval renders the biggest bottleneck. A natural idea to minimize such communication traffic is to leverage the power-law degree distribution [16] of real-life graphs. For example, PaGraph [38] adopted a static (no replacement at runtime) cache that stores the predicted hottest node features locally. Upon cache hit, the traffic of feature retrieving can be saved. Unfortunately, on giants graphs like Pinterest graph [53], such a static cache may only be able to store a small fraction of nodes due to memory constraints. We find, when only 10% of nodes can be cached, the static cache only yields <40% cache hit ratios.

Why not use dynamic (replacing some caches at runtime) cache policies? It is challenging because it would incur large searching and updating overhead, pointed out in [38]. Overheads become even larger when the cache is large (tens of GB) and stored on GPU. Our best-effort implementation

Table 1: Qualitative comparison of graph partition algorithms.

Partition Algorithms	Scalability to Giant Graphs	Balanced Training Nodes	Multi-hop Connectivity
Random [2, 30]	✓	✓	✗
METIS [32] & ParMETIS [33]	✗	✓	✓
GMiner [10]	✓	✗	✗
PaGraph [38]	✗	✓	✓

echos [42, 44] – we also find that popular policies like LRU and LFU lead to a near 80-millisecond overhead for updating.

Nevertheless, we will show in §3.2 that it is still possible to achieve a good trade-off between cache hit ratios and dynamic cache overhead by exploiting the characteristics of GNN training and carefully designing the cache engine.

Challenge 2: Need for a graph partition algorithm that is scalable and friendly to subgraph sampling. Beyond node feature retrieving, communication overhead of subgraph sampling renders another major bottleneck.

The partition algorithms affect the sampling overheads in two ways. First, they determine cross-partition communication overhead. GNN sampling algorithms construct a subgraph by sampling from a training node's *multi-hop* neighbors. If the neighbors are hosted on the same graph store server, the *sampler* colocated with graph store servers can finish sampling locally. Otherwise, it must request data from other servers, incurring a high communication overhead. Like random partitioning[2] [2, 30], naive algorithms are agnostic to the graph structure. Most state-of-the-art (SOTA) partition algorithms on graph processing and graph mining, like GMiner [10] and CuSP [26], only consider one-hop connectivity instead of multi-hop connectivity, which is suboptimal.

Second, partition algorithms determine the load balance across graph store servers and sampler processes. In a training epoch, one must iterate all *training nodes* and sample subgraphs based on them. For good load balance, one should balance the training nodes across partitions. However, SOTA graph partition algorithms only consider balancing all the nodes, of which only 10% [27, 47] are training nodes. Because they focus on maintaining neighborhood connectivity, they may produce less balanced partitions than the pure random algorithm, especially imbalanced for the training nodes.

Since we aim for GNN training on giant graphs, the partition algorithm must be scalable to giant graphs as well. Like the METIS [32, 33] used by DGL, some partition algorithms rely on maximal matching to coarsen the graph, which is not friendly to giant graphs due to high memory complexity [24]. Some other algorithms, such as PaGraph [38], have high time complexity and are not friendly to giant graphs.

Ideally, we need a partition algorithm that works on giant graphs and simultaneously minimizes the cross-partition communication and load imbalance during sampling. As shown in Table 1, none of the existing partition algorithms satisfies our needs, which motivates our algorithm (§3.3).

[2]Also including Lux [30], which is a random partition algorithm that frequently re-partitions the graph for load balancing.

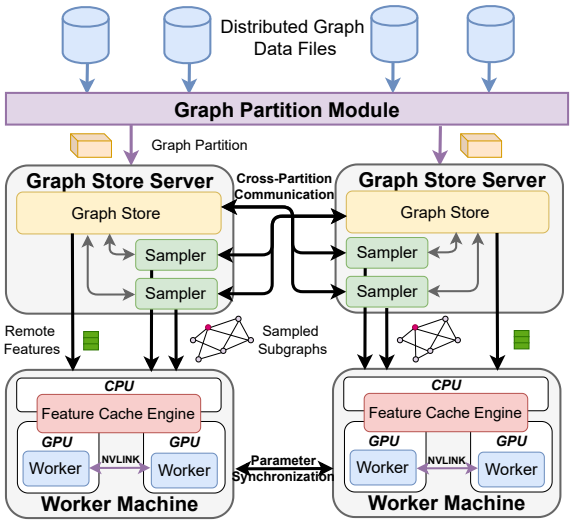


Figure 4: The architecture of BGL.

Challenge 3: Different data preprocessing stages contend for resources. When running all stages together, we further identify a unique problem of GNN training – the preprocessing is much more complex than traditional DNN training. The subgraph sampling, subgraph structure serialization and de-serialization, node feature retrieving, and cache engine all consume CPU and memory/PCIe/network bandwidth resources. We observe that if all the processes freely compete for resources, the resource contention may lead to poor performance. Some operations may try to acquire more resources than what they need and hence block other operations, while they do not scale well with more resources.

Existing GNN training frameworks largely ignore this problem. DGL, PyG, and Euler either blindly let all processes freely compete or leave the scheduling to underlying frameworks like TensorFlow and PyTorch. The low-level frameworks are agnostic to the specifics in GNN training, and thus are also naive and suboptimal. Our answer to this challenge is a carefully designed resource isolation scheme (§3.4).

3 Design

We design BGL to address the challenges presented in §2.3.

3.1 Architecture and Workflow

The overall architecture of BGL is shown in Figure 4. A training job has the following stages.

Pre-training preparation: graph partition. The *graph partition module* loads the graph data stored in the distributed storage system (*e.g.,* HDFS), and shards the whole graph into several partitions. Graph partitioning is a *one-time cost*, and the results can be saved in storage and used by other GNN training tasks later. Then, each partition is loaded into a graph store server's memory, ready for subgraph sampling.

To address Challenge 2 in §2.3, BGL's graph partition module first uses multi-source BFS to merge nodes into several blocks for reducing the graph size. Optimal graph partitioning is NP-hard [7]. Hence, we propose a partition heuristic considering both multi-hop connectivity of blocks and training workload balancing to maximize the partition locality, thus minimizing the cross-partition sampling time.

Subgraph sampling at each training step. Samplers run on the CPUs of graph store servers. They select several training nodes and sample their multi-hop neighbors by iteratively sampling next-hop neighbors several times. If all the next-hop neighbors are stored in the current graph store server, samplers can get the list locally; otherwise, they need to send network requests to other graph store servers.

Training GNN using the sampled subgraphs. Each worker in BGL runs on 1 GPU. It receives sampled subgraphs from samplers and retrieves features of subgraph nodes from graph store servers, with a local *feature cache engine* to improve the retrieving efficiency.

To address Challenge 1 in §2.3, BGL's feature cache engine adopts an algorithm-system co-design. We leverage the temporal locality — in nearby mini-batches, we always attempt to train nodes with close distance in the graph. Combined with a FIFO policy, BGL achieves high cache hit ratios and low cache overheads. Increasing the temporal locality of training nodes may influence the convergence of GNN models. We show BGL can preserve the SOTA training accuracy by carefully introducing randomness in ordering training nodes. On the system side, we exploit high-bandwidth GPU-to-GPU communication with NVLinks, and design a multi-GPU cache supporting dynamic caching strategies.

Finally, BGL uses a fine-grained pipeline, allowing parallel and asynchronous execution of each stage. To address Challenge 3 in §2.3, BGL adopts resource isolation when assigning resources to each pipeline stage. Specifically, BGL formulates an optimization problem and assigns isolated resources accordingly to minimize the execution time of each pipeline stage under resource constraints (§3.4).

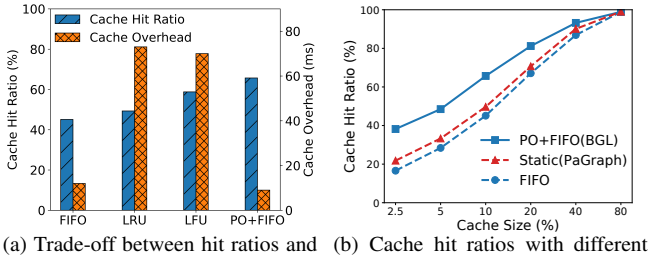
3.2 Feature Cache Engine

Feature retrieving contributes to the majority of communication overheads. We propose a feature cache engine, which uses system-algorithm co-design to minimize this overhead.

3.2.1 Dynamic Cache Policy

The first question is, which dynamic caching policy should we choose? PaGraph [38] indicates that dynamic policies have too high overheads. Based on our best-effort implementation[3], we compare popular caching policies, including LRU, LFU and FIFO in Figure 5a. Since cache queries arrive in batches, we define the cache hit ratio as the percentage of hit nodes in total number of nodes in a batch. The cache overhead is

[3]We implement LFU and LRU with $O(1)$ time complexity and use a contiguous 1D array as a HashMap to speed up key searching.



(a) Trade-off between hit ratios and overhead (10% cache size).

(b) Cache hit ratios with different cache sizes.

Figure 5: We test the cache hit ratios and overhead on Ogbn-papers with different cache sizes. PO is short for proximity-aware ordering, which is proposed in §3.2.2.

the *amortized* time, including cache lookup on all nodes *and* cache update upon cache misses. Hence, a higher cache hit ratio, representing less frequent cache updates for dynamic caching, can help reduce the amortized overhead.

LRU [42] and LFU [44] indeed have intolerable cache overhead. FIFO's overhead (<20ms per batch) meets the throughput requirement for GNN training – as mentioned in §2, an iteration of typical GNN model computation on GPU is around 20ms. In an asynchronous pipeline with cache as a part of data prefetching, FIFO cache will not become the bottleneck.

However, FIFO's cache hit ratio is unimpressive – it is even lower than static policy's (Figure 5b). The reason is that FIFO does not leverage the distribution of node features. Regardless of how hot the node feature is, it is evicted as frequently as other colder node features.

3.2.2 Proximity-Aware Ordering

To address the above problem, we propose *proximity-aware ordering* – in nearby mini-batches, we always attempt to visit the neighboring training nodes in the graph. Figure 5b shows that FIFO combined with proximity-aware ordering can achieve the highest cache hit ratio among all candidate cache policies while maintaining low cache overhead.

We observe that each node may appear more than once among different training batches (e.g., node ⑨ in Figure 6a appears three times in sampled subgraphs). This gives us an opportunity for data reuse by caching node features in nearby mini-batches (a.k.a., *temporal locality*). With random training nodes sampling, the chances of a node in nearby training batches are low. In order to increase the probability, we propose to select training nodes in a BFS order. BFS preserves the graph connectivity in terms of number of hops. Hence, nearby training nodes in graphs are more likely to be selected in consecutive batches. As a result, this ordering increases the probability that each node appears in consecutive batches and improves the cache hit ratio.

For example, in Figure 6a, starting from a BFS root node ⑰, we can generate a BFS sequence of training nodes. Random ordering (Figure 6b) results in no cache hits in the first three batches. On the contrary, with proximity-aware ordering (Figure 6c), the second batch and the third batch contain nodes that exist in the previous batches (*i.e.*, {⑰, ⑨, ③}

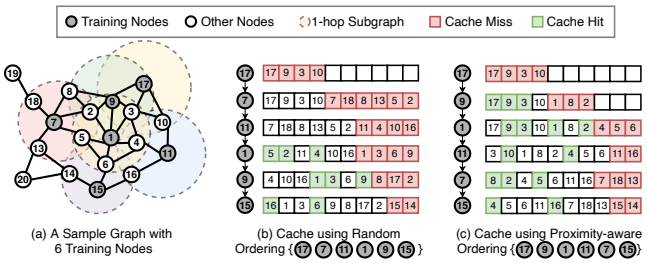


Figure 6: Compared to random ordering, using proximity-aware ordering improves hit ratios of FIFO cache.

in the second batch and {⑨, ③, ①, ②} in the third batch). Consequently, FIFO cache hits are improved from 8 to 14.

However, there is a trade-off between improving the temporal locality and ensuring model convergence. Traversal-based ordering improves the temporal locality but violates the i.i.d. requirement of SGD, leading to different label distributions of batches and slowing model convergence. On the other hand, random ordering, such as random shuffling, achieves state-of-the-art model accuracy by selecting random training nodes, with the cost of poor temporal locality.

Our proximity-aware ordering needs to balance the above trade-off. The key idea is that SGD is robust enough, and slightly relaxing the i.i.d. requirement does not influence the convergence rate. Theorem 3.15 in [41] shows that if there is *little difference* between the output distribution of one ordering algorithm A and the uniform distribution, A will not cause accuracy degradation. Hence, in BGL, samplers still select training nodes based on BFS traversal, while we carefully introduce randomness to reduce the difference.

We introduce the following randomness. First, we use several different BFS sequences, instead of only one, and each of them is generated by selecting random BFS roots. To form a training batch, we select training nodes from different sequences in a round-robin manner. Second, we circularly shift each BFS sequence by a random position. Since giant graphs have lots of small connected components [37], they are more likely to be traversed at last and appended at the end of each BFS sequence in our implementation. This deterministic behavior harms the model accuracy. Shifting by a random position minimizes its impact to the model, and circular shifting preserves the order of consecutive nodes in BFS sequences.

How many BFS sequences should we select? We find, as long as the model convergence is guaranteed, we should use the minimum number of sequences to maximize the temporal locality. Meng et al. [41] define the difference ε, named *shuffling error*, as the total variation distance between the two distributions, and proves that, if $\varepsilon \leqslant \sqrt{bM}/n$, the convergence is not influenced, where b is the batch size, M is the number of workers and n is the size of training data.

Based on the above theorem, we determine the number of sequences as follows. We use the label distribution to calculate the shuffling error. The label distribution of proximity-aware ordering is estimated as the probability of each label appearing in each mini-batch. Before training, BGL firstly generates

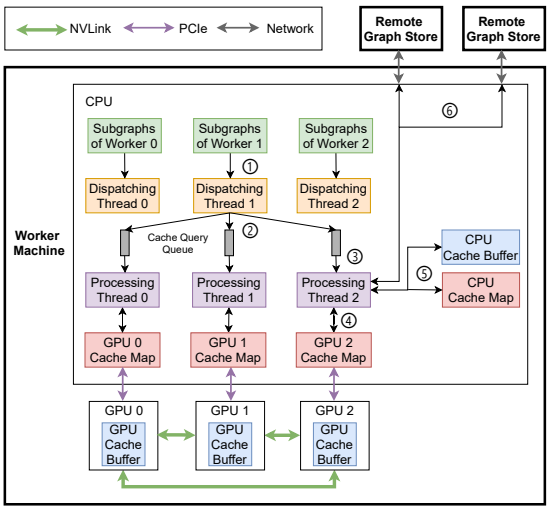


Figure 7: Structure and workflow of feature cache engine.

hundreds of BFS sequences. After that, it gradually increases the number of BFS sequences from one until the shuffling error is smaller than the requirement of convergence ($\sqrt{bM}/n$). During training, BGL constructs each training batch by introducing randomness and reusing generated sequences. This procedure incurs negligible overheads (<1% training time).

3.2.3 Maximizing Cache Size

Based on the observation that GNN models are typically small (§2.3) and large GPU and CPU memory are unused, in BGL, we jointly use the memory of multiple GPUs (if the training job uses multiple GPUs) and CPU memory to build a two-level cache, which can enlarge the cache size and increase the cache hit ratio. The detailed structure and cache workflow of our feature cache engine is shown in Figure 7.

Multi-GPU Cache. We create one *cache map* and one *cache buffer* for each GPU. Cache map is a HashMap with node IDs as keys and the pointers to buffer slots in cache buffer as values. Cache buffer contains buffer slots, storing node features. Each GPU cache map manages its own cache buffer.

To avoid wasting precious GPU memory, we ensure no duplicated entries among all GPU cache buffers by assigning different and disjoint node IDs to each GPU cache map (`mod` by the number of workers). A GPU can fetch node features from another GPU via P2P GPU memory copy using NVLinks. As mentioned in §2.2, transferring 60 mini-batches can saturate the 100Gbps NIC and PCIe 3.0 x16 bandwidth. Hence, using NVLinks not only provides high bandwidth and low latency for inter-GPU communication, but it also alleviates heavy communication in the network and PCIe links.

Since CPU memory is much larger than GPU memory, BGL also adds a CPU cache on top of the multi-GPU cache to further increase the cache size and reduce the communication traffic to graph store servers. The CPU cache uses the same cache policy as the GPU cache, so we omit the details.

Cache Workflow that Guarantees Consistency. As shown in Figure 7, the workflow of the cache engine goes as follows.

After receiving sampled subgraphs (①), dispatching threads split the subgraph nodes by `mod` operation into multiple *cache queries*[4] and send them to *cache query queues* (②). Each processing thread is assigned to one GPU cache buffer and processes all cache queries on this buffer (③). It first looks up the subgraph nodes in the GPU cache map and then gathers cached features of those nodes from GPU cache buffers (④). In case of GPU cache misses, it looks up the CPU cache map for uncached nodes, gathers cached feature tensors from CPU cache buffer, and sends them to the GPU (⑤). The remainders are requested from graph store servers and sent to GPUs once received (⑥). Finally, the cache map and the cache buffer are updated according to our FIFO caching policy.

Though node features are immutable (§2), cache buffers are still mutable. The cache buffer and the cache map may be *inconsistent* when some buffer slots are read and written by different GPU workers simultaneously (which occurs when different nodes are assigned to the same buffer slot). To ensure the consistency between the cache buffer and the cache map, a naive solution is to use locks for each buffer slot. But, this locking means synchronization in CUDA APIs for GPUs, leading to large overhead. Our solution is to queue all the operations towards a given GPU cache, including queries and updates. Only one processing thread polls the queue and then reads or writes the corresponding GPU cache buffer. This reduces the overhead by 8x compared with using locks while avoiding racing.

3.3 Graph Partition Module

3.3.1 Partition Workflow

Graph partitioning largely impacts the cross-partition communication when sampling subgraphs. As described in §2.3, a good partition algorithm should have the following properties: (1) scalability to billion-node graphs, ensuring (2) multi-hop connectivity, and (3) training load balance.

Our algorithm exploits two types of processes: *block generators* and *a block assigner*. Block generators generate blocks, each of which is a connected subgraph and treated as one node in the coarsened graph. The block assigner collects blocks of the coarsened graph from block generators and assigns each block to one partition. We outline the three major steps of our partition algorithm in Figure 8.

(1) Multi-level Coarsening: Each block generator loads disjoint graph data from HDFS and generates blocks on the loaded graph.

Different from merging procedures used in other partition algorithms (*e.g.*, maximal matching in METIS), we use multi-source BFS to generate blocks, which can preserve multi-hop connectivity in the original graph. The block generator randomly chooses a few nodes as the BFS source nodes. Each source node is assigned a unique block ID and broadcasts the block ID to its neighbors. Once the block size (*i.e.*, the number of nodes with the same block ID) exceeds a threshold (*e.g.*, 100K), or there are no unvisited neighbors in BFS, a block is generated. When all nodes are visited, the block generating procedure stops. At the same time, block generators maintain a mapping from the node ID to block ID, and synchronize it among them for uncoarsening.

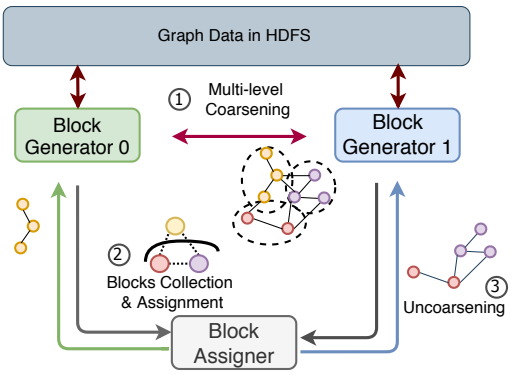


Figure 8: The partition workflow. Node colors denote different blocks in the coarsened graph (step ②), or the nodes belonging to different blocks (steps ① and ③).

However, we find billion-node graphs have numerous connected components [37]. After one round of coarsening, the coarsened graph still contains a large quantity of nodes, which results in large partition complexity. Hence, we further deploy a multi-level coarsening strategy. First, for small blocks connecting to large blocks[5], we merge them to their large block neighbors. Second, other small blocks without large block neighbors are randomly merged. By considering neighborhood relationship, this approach not only speeds up the partition process but also preserve the multi-hop connectivity.

(2) Block Collection & Assignment: The block assigner collects the blocks of the multi-level coarsened graph from block generators. It applies a greedy assignment heuristic for each block, targeting both multi-hop locality and training node balancing. The block assigner then broadcasts the block partitions to the generators. We leave the details of the assignment heuristics in §3.3.2.

(3) Uncoarsening: Upon receiving the block assignment results from the block assigner, the block generators start mapping back the blocks to the nodes in the original graph, *i.e.*, uncoarsening. The partition results are then saved to the HDFS file (step ③ of Figure 8).

As a result, our partition algorithm has low time complexity and is friendly to giant graphs. Let $\mathcal{E}_1$ be the set of edges in the coarsened block graph after BFS. $\mathcal{E}_2$ denotes the set of edges in the graph for assignment after multi-level block merging, and j denotes the number of hops to maintain connectivity. We reduce the time complexity of the assignment to $O(|\mathcal{E}_2|^j)$, much lower than SOTA $O(|\mathcal{E}|^j)$ [38], where $|\mathcal{E}_2| \ll |\mathcal{E}|$. The total partitioning complexity is $O(|\mathcal{E}| + |\mathcal{E}_1| + |\mathcal{E}_2|^j)$.

[4] A cache query contains all nodes which are assigned to one GPU cache buffer by `mod` operation in a sampled subgraph.

[5] Empirically, we set blocks with top 10% sizes as large blocks.

3.3.2 Assignment Heuristic

Since optimal graph partitioning is NP-hard [7], we propose a new heuristic for assigning blocks to partitions by considering the special requirements of GNN training.

Our heuristic is to derive the block assignments by solving the following maximization problem:

$$\max_{i\in[k]}\left\{\left(\sum_j |P(i)\cap\Gamma^j(B)|\right)\cdot\left(1-\frac{|T(i)|}{C_T}\right)\cdot\left(1-\frac{|P(i)|}{C}\right)\right\}$$

where k is the number of partitions; each partition is referred by its index $P(i)$. Based on this heuristic, each block B is assigned to the partition with the maximum value.

The first term in the heuristic is the *multi-hop block neighbor* term, $\sum_j |P(i)\cap\Gamma^j(B)|$, which counts the intersection between the set of j-hop neighbor blocks of B, $\Gamma^j(B)$, and the current partition $P(i)$. Using this term, we tend to assign the current block to a partition with the maximum number of neighbors and preserve the multi-hop connectivity. Second, we introduce the *training node penalty* term, $(1-|T(i)|/C_T)$, where $T(i)$ denotes the set of training nodes that have been assigned to the ith partition, and $C_T = |\mathcal{T}|/k$ denotes the training node capacity constraint on each partition. By maximizing this term, each partition is enforced with the same number of training nodes. Third, we introduce the *node penalty* term, $(1-|P(i)|/C)$, where $C = |\mathcal{V}|/k$ is the capacity constraint on each partition. This term is commonly used in existing partition algorithms to balance the number of nodes among the partitions. Finally, we multiply the three terms to maximize them simultaneously.

3.4 Resource Isolation For Contending Stages

To improve resource utilization and training speed, we divide GNN training into 8 asynchronous pipeline stages (see Figure 9) with careful consideration of data dependency and resource allocation. This is more complex than traditional DNN training. Some of the stages contend for CPU, Network, and PCIe bandwidth resources: (i) Processing sampling requests and constructing subgraphs compete for CPUs on graph store servers. (ii) Subgraph processing (e.g., converting graph format) and executing cache workflow compete for CPUs in the worker machine. (iii) Moving subgraphs and copying features to GPUs compete for PCIe bandwidth.

However, we find that if all the processes freely compete for resources, the resource contention may lead to poor performance. A key reason is that some operators may acquire more resources than what they actually need and block other stages, with which they do not scale well.

For example, we observe that for the executing cache workflow stage (Stage 4 in Figure 9), when the number of CPU cores exceeds a threshold (e.g., 40), the performance converges or even degrades with more CPU cores (e.g., more than 64). This is because of the memory bandwidth limit, synchronization and scheduling overhead in the multi-threading library like OpenMP [8].

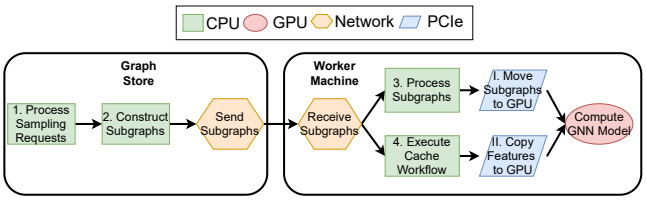


Figure 9: GNN training pipeline in BGL.

To solve the above problem, we propose a *profiling-based resource allocation* to assign isolated resources to different pipeline stages. We first profile the execution time of each stage and then adjust resource allocation to balance the execution time of each stage. We formulate the following optimization problem to compute the best resource allocation in a given GNN training task:

$$\min\ \max\left\{\frac{T_1}{c_1},\frac{T_2}{c_2},T_{net},\frac{T_3}{c_3},\frac{D_I}{b_I},f(c_4),\frac{D_{II}}{b_{II}},T_{gpu}\right\}$$
$$\text{s.t.}\quad c_1+c_2\leqslant C_{gs},\quad c_3+c_4\leqslant C_{wm},\quad b_I+b_{II}\leqslant B_{pcie}$$

The objective is to minimize the maximal completion time of all pipeline stages. The constraints are resource capacity constraints for CPU on graph store servers, CPU on worker machines, and PCIe bandwidth. The main decision variables are c_i ($i\in\{1,2,3,4\}$), the number of CPUs required for the ith stage; and b_i ($i\in\{I,II\}$), PCIe bandwidth of the ith stage. All the other quantities are profiled by our system, including the time of the ith stage T_i, the data size of processed subgraphs D_I, and the average data size of missed features D_{II} when the cache is stable. C_{gs} and C_{wm} denote the number of CPU cores on graph store servers and worker machines, respectively, and B_{pcie} is the PCIe bandwidth of the worker machines. We assume linear acceleration of CPU execution, except on processing caching operation (Stage 4 in Figure 9). We introduce a fitting function $f(c_4)=a/c_4+d$ to output the completion time of caching stage with a certain number of CPU cores c_4, where a and d are approximated by pre-running.

We use brute-force search to find the optimal resource allocation. To reduce the search space, we add integer assumptions on bandwidth variables b_I and b_{II}. The time complexity is $O(C_{gs}^2+C_{wm}^2+B_{pcie}^2)$ in the worst case. On average, our method spends less than 20ms on searching for the best resource allocation strategy for GNN training pipeline.

4 Implementation

We implement BGL with over 4,400 lines of C++ code and 3,300 lines of Python code. We reused the graph store module and GPU backend of the open-sourced Deep Graph Library (DGL v0.5 [1,48]), and utilized the graph processing module of GMiner [10] for partitioning. Our design can be applied to other GNN frameworks with little change. We are collaborating with the DGL team to upstream our implementation.

Requirement. BGL exploits NVLinks/NVSwitches for high-bandwidth low-latency cross-GPU communication for multi-GPU cache. Our measurement shows that without NVLinks, the feature cache engine retrieves cached features from other

Table 2: Datasets used in evaluation.

	Ogbn-products	Ogbn-papers	User-Item
Nodes	2.44M	111M	1.2B
Edges	123M	1.61B	13.7B
Feature Dimension	100	128	96
Classes	47	172	2
Training Set	196K	1.20M	200M
Validation Set	393K	125K	10M
Test Set	2.21M	214K	10M

GPUs via PCIe with much lower bandwidth, which could decrease throughput of BGL by 50%.

Feature Cache Engine. Cache workflow in feature cache engine contains several GPU operations, such as copying tensor from CPU memory to GPU memory and launching kernels to copy tensor from/to other GPUs. To make cache processing asynchronous, we enqueue all cache GPU operations into a *dedicated* CUDA stream, and pre-allocate dedicated CPU memory as buffers and pin these memory. Our cache engine uses CUDA Unified Virtual Addressing and enables fast GPU P2P communication on each cache processing thread. The cache processing thread enqueues a lightweight CUDA callback function into the CUDA stream, which counts the number of finished cache queries and notifies workers.

To further expedite FIFO performance, BGL uses multiple OpenMP threads to execute FIFO concurrently. We maintain an atomic `tail` shared by all threads to record the next column index of the GPU cache buffer for insertion or eviction. When inserting a new node, each thread finds the next position by atomically increasing `tail`, and the real position is `(tail+1)%buffer_size`. If this position has an old node, it evicts the old node from the GPU cache map. Since we assume node features are immutable during training, old node features are implicitly evicted by inserting new node features.

Inter-Process Communication. We use separate processes for sampling, feature retrieving, and GNN computation stages. To minimize the IPC overhead, we use shared memory to avoid unnecessary memory copy among different processes. Specifically, we use Linux Shared Memory and CUDA IPC to avoid unnecessary CPU and GPU memory copy, respectively.

5 Evaluation

5.1 Methodology

Testbed. We evaluate BGL on a heterogeneous cluster with 4 GPU servers and 32 CPU servers. The GPU server has 8 Tesla V100-SMX2-32GB GPUs (connected by NVLink v2), 96 vCPU cores, and 356GB memory. Each CPU server has 96 vCPU cores and 480GB memory. All servers are interconnected with 100Gbps Mellanox CX-5 NICs. The graph datasets are stored in HDFS.

Datasets. As shown in Table 2, we train GNNs on three datasets with different sizes, including two public graph datasets: Ogbn-products [27] and Ogbn-papers [47], as well as a proprietary web-scale graph dataset: User-Item.

GNN Models. We evaluate BGL with three representative GNN models: GCN (Graph Convolution Network) [36], GAT (Graph Attention Network) [46] and GraphSAGE [23]. We use the same model hyper-parameters as OGB leaderboards [3], e.g., 3 layers and 128 hidden neurons per layer.

Mini-batch Sampling Algorithms. In our experiments, we use Neighbor Sampling [23], which is shown to achieve comparable model performance with full-batch graph training.[6] Except for the experiment in §5.7, we set the mini-batch size to 1000, *i.e.,* each mini-batch contains 1000 sampled subgraphs and each subgraph contains one training node and its three-hop neighbors with fanout {15,10,5}.

Baselines. We use four open-sourced and widely-used GNN training frameworks as baselines for comparison[7].

- Euler [2]: Euler (v1.0) is a distributed graph learning system built atop TensorFlow [5]. We use TensorFlow's GPU backend for acceleration.
- DGL [1]: DGL is a deep learning library for graphs, compatible with multiple deep learning frameworks. We use the DGL v0.5 release (DistDGL [58]).
- PyG [17]: PyG (v1.6.0) extends PyTorch for deep learning on graphs. It contains a mini-batch loader for multi-GPU support in a single machine.
- PaGraph [38]: PaGraph is a sampling-based GNN framework with a static cache strategy on GPU, which supports multi-GPU in a single server.

Specifically, PyG co-locates graph store servers and workers and allows graph sampling on the same machine only, making it unable to process large graph datasets (i.e., Ogbn-papers and User-Item) due to memory limit. Hence, we only compare BGL with PyG on Ogbn-products dataset. When training on User-Item dataset with DGL and PaGraph, we separate the graph store servers from the workers since our GPU servers do not have enough memory to load the graph partitions. To evaluate the performance boundary, we use 4, 8 and 32 CPU-based graph store servers for all frameworks on Ogbn-products, Ogbn-papers and User-Item respectively.

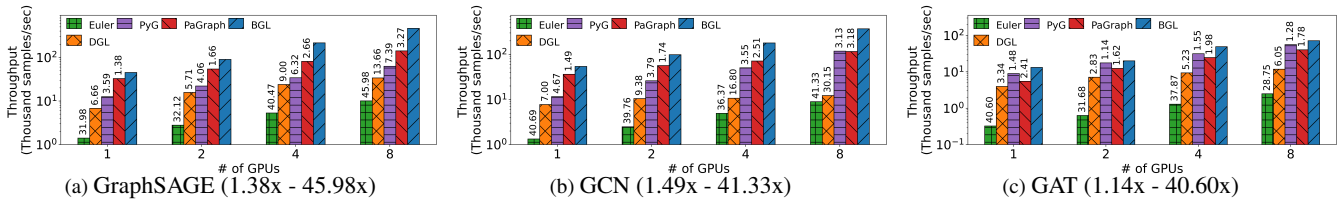
Graph Partitioning. DGL uses METIS partitioning for small graphs (*i.e.*, Ogbn-products), and Random partitioning for large graphs that cannot be fitted into a single machine (*i.e.*, Ogbn-papers and User-Item). Euler uses random partitioning for all graphs, and BGL uses the proposed algorithm in §3.3, where we set $j = 2$, *i.e.*, searching two-hop neighbors.

5.2 Overall Performance

Figure 10, 11 and 12 show the training speed of baselines and BGL in *log* scale when training the three GNN models

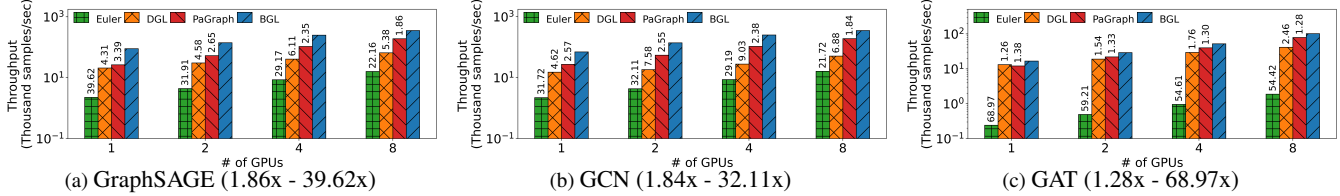
[6] BGL can also be applied to other vertex-centric GNN sampling algorithms, e.g., layer-wise sampling [11] and random walk sampling [53]. We omit the evaluation of other sampling algorithms since it is beyond our scope.

[7] We omit P^3 [20] because it is not open-sourced.



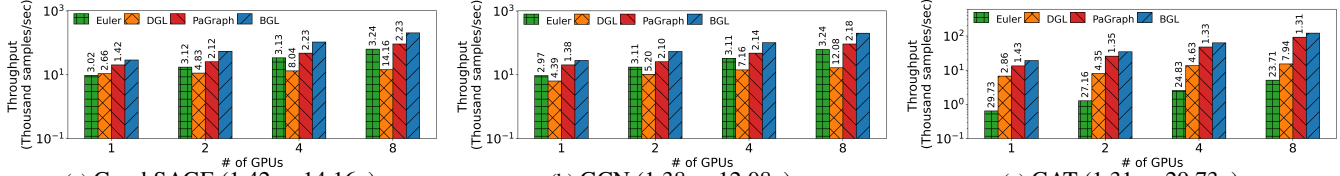
(a) GraphSAGE (1.38x - 45.98x) (b) GCN (1.49x - 41.33x) (c) GAT (1.14x - 40.60x)

Figure 10: Throughput of 3 GNN models on Ogbn-products in *log* scale. Numbers above bars are speedups of BGL over other systems.



(a) GraphSAGE (1.86x - 39.62x) (b) GCN (1.84x - 32.11x) (c) GAT (1.28x - 68.97x)

Figure 11: Training throughput of 3 GNN models on Ogbn-papers in *log* scale. Numbers above bars are speedups of BGL over other systems.



(a) GraphSAGE (1.42x - 14.16x) (b) GCN (1.38x - 12.08x) (c) GAT (1.31x - 29.73x)

Figure 12: Training throughput of 3 GNN models on User-Item in *log* scale. Numbers above bars are speedups of BGL over other systems.

on three graph datasets, with the number of workers ranging from 1 to 8, where each worker has one GPU. We use *samples/sec* as the metric to measure the training speed. A sample is a sampled subgraph of one training node.

Different Frameworks. BGL achieves 1.14x - 69x speedups over four baselines in all settings. BGL has 69x (the most) speedup over Euler. This is because Euler's random sharding in graph partition has very low data locality, resulting in frequent cross-partition communication in sampling. DGL does not cache features on GPU, introducing significant feature retrieving time. Thus, BGL outperforms DGL by up to 30x. PaGraph performs the best among baselines. It places graph structure data on each GPU with static caching on node features, leading to much faster data preprocessing. Even in this case, BGL still has up to 3.27x speedup, thanks to dynamic feature caching and resource isolation for contending pipeline stages. BGL outperforms all other systems, and the geometric mean of speedups over PaGraph, PyG, DGL and Euler is 1.91x, 3.02x, 7.04x and 20.68x, respectively.

Different GNN models. The training performance varies significantly across different GNN models. We see that BGL achieves significantly higher performance improvement with GraphSAGE and GCN models, by up to 30x as compared to DGL and PyG. With the computation-intensive GAT model, however, the training speed of PyG and DGL is closer to that of BGL. Hence the gain for BGL ranges from 14% to 8x. It is because the GAT model is computation-bound due to incorporating the attention mechanism into the propagation step, while its communication is less intensive than the other two GNN models; the higher ratio of computation over other stages results in a smaller improvement space for BGL. We see that Euler performs the worst in GAT, since it does not optimize the GPU kernels for irregular graph structures.

Scalability. BGL also outperforms other frameworks in terms of scalability. Without caching features on GPU, the throughput of baseline frameworks is bounded by PCIe bandwidth. For example, DGL has only 3x speedups when increasing the number of GPUs from 1 to 8. BGL reduces the transmitted data through PCIe bandwidth with efficient GPU cache, resulting in linear scalability in throughput. Multi-GPU systems often suffer poor scalability due to synchronization overhead or resource contention. However, our design and implementation of multi-GPU memory sharing scales well with the increased number of GPUs. With extra bandwidth brought by NVLink, accessing cache entries on other GPUs introduces negligible overhead. On the contrary, the increased cache capacity improved the cache hit ratio (Figure 5b) and reduced overall feature retrieving time (Figure 13).

We observe the relatively lower improvement with the User-Item dataset. On the billion-node graph dataset, the subgraph sampling and feature retrieving becomes more time consuming, due to the inconsistent sampling performance of DGL graph store server and sparse graph structure. Hence, BGL cannot produce the similar level of overlapping with the unchanging model computation time.

GPU Utilization. We compare the GPU utilization achieved by BGL and DGL with the same GPU backend. We run GraphSAGE and GAT models on Ogbn-products dataset with 8 GPU. BGL achieves 99% GPU utilization with the computation-intensive GAT model, while DGL's utilization is only 38%. For GraphSAGE model with shallow neural layers, BGL improves the GPU utilization from 10% to 65%.

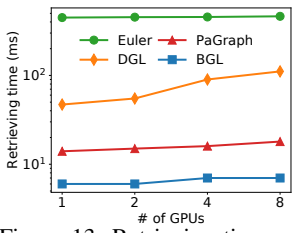


Figure 13: Retrieving time per mini-batch on Ogbn-papers.

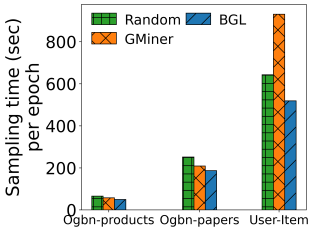


Figure 14: Graph sampling time per epoch during training.

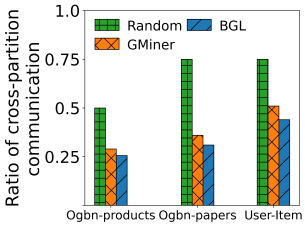


Figure 15: BGL reduces ratio of cross-partition communication.

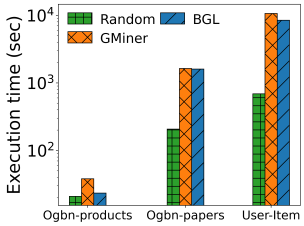


Figure 16: One-time partitioning execution time before training.

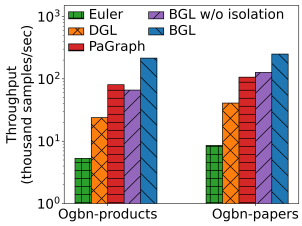


Figure 17: Resource isolation improves training throughput.

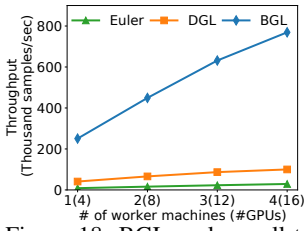
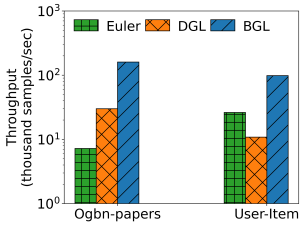
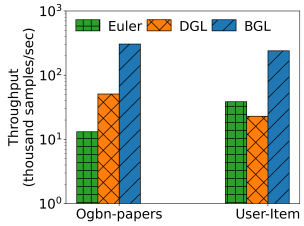


Figure 18: BGL scales well to multiple worker machines.



(a) BS 1000, 3 hops, FO {10,10,10}.



(b) BS 500, 2 hops, FO {10,25}.

Figure 19: Training throughput of GraphSAGE using different hyper parameters on 4 GPUs. BS and FO stand for 'batch size' and 'fanout'.

5.3 Impact of Feature Cache Engine

In §3.2, we have shown the cache hit ratio with different cache policies and cache sizes. The trend of them is similar on other datasets. Here, we present the amortized feature retrieving time with the feature cache engine.

We compare the feature retrieving time of one mini-batch using different GPUs on Ogbn-papers. We implement PaGraph static caching policy in BGL, which caches the features of high-degree nodes. Euler and DGL do not have cache, so the feature retrieving time is the elapsed time of transmitting features from graph store servers to GPU memory. As shown in Figure 13, due to high cache hit ratios and low cache overhead, the feature retrieving time of BGL is the shortest among all systems. Compared to other systems on 1 GPU worker, BGL reduces the feature retrieving time by 98%, 88% and 57% for Euler, DGL and PaGraph, respectively.

5.4 Impact of Graph Partition

We compare the graph partition algorithm in BGL with Random and GMiner partitioning, since only these two partition algorithms can scale to Ogbn-papers and User-Item. We evaluate the sampling time per epoch and the one-time partition time (counted from loading the graph data to saving the partition results to files) under different partition algorithms. Ogbn-products, Ogbn-papers and User-Item are divided into 2, 4 and 4 partitions, respectively.

Figure 14 shows the graph sampling time (per epoch) under different partition algorithms. BGL achieves the best performance across different graph datasets, reducing the sampling time by at least 20% over Random partition algorithm. Compared to GMiner, BGL manages to drop the sampling time by 14% and 10% for Ogbn-products and Ogbn-papers, respectively, thanks to its training node balancing and multi-hop connectivity of partitioning.

The reduction in sampling time mainly comes from the reduced cross-partition (inter-server) communication during distributed neighbor sampling. As shown in Figure 15, by including multi-hop locality when partitioning, BGL reduces the ratio of cross-partition communication by 25%, 44%, and 33% for Ogbn-products, Ogbn-papers and User-Item, respectively. The cross-communication traffic is only determined by the number of partitions, but not the number of graph store servers or worker machines.

Partitioning a large-scale graph is time consuming. Hence, BGL introduces multi-level coarsening to mitigate the extra complexity brought by computing multi-hop locality. Figure 16 shows BGL's partition algorithm runs as fast as the well-optimized original GMiner, and is even better than GMiner on graph User-Item with 20% reduction of time.

5.5 Impact of Resource Isolation

To evaluate the effectiveness of our resource isolation mechanism, we compare BGL with Euler, DGL, PaGraph, and BGL without resource isolation when training GraphSAGE with 4 GPUs on datasets Ogbn-products and Ogbn-papers. 'BGL w/o isolation' is a naive resource allocation method that shares all pipeline stages resources. It increases resource utilization but incurs larger contention and parallel overhead.

As shown in Figure 17 (in log scale), BGL achieves the highest throughput. Both BGL and 'BGL w/o isolation' outperform Euler and DGL. Due to the overhead of resource contention, the performance of 'BGL w/o isolation' on Ogbn-products is even lower than that of PaGraph. BGL uses resource isolation method, which mitigates the resource contention among different pipeline stages and incurs a lower parallel overhead of OpenMP. As a result, BGL speedups the throughput by up to 2.7x, compared to the naive resource allocation strategy without isolation and PaGraph.

5.6 Scalability to Multiple Worker Machines

To show the scalability of multiple worker machines, we vary the number of worker machines from 1 to 4, and each has 4 GPUs. We train GraphSAGE model on graph Ogbn-papers with Euler, DGL and BGL. The number of graph store servers remains the same as in §5.2.

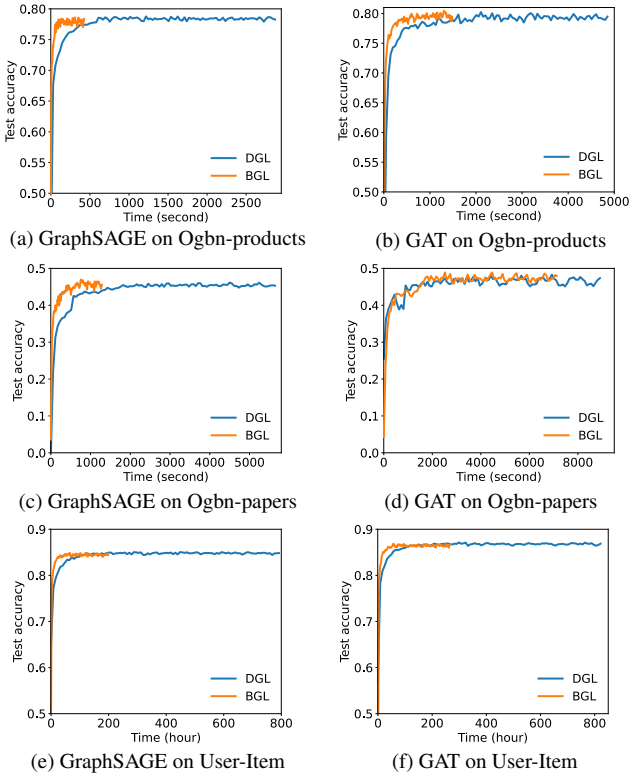
As shown in Figure 18, BGL improves throughput from 250K to 769K (76% of linear scalability) when the number of worker machine increases from 1 to 4. Due to no feature cache on GPU and bottleneck in PCIe and network bandwidth, throughput of Euler and DGL cannot scale well when increasing the number of worker machines. Since our GPU servers only use NVLink v2, the cache engine cannot share GPU memory across machines, and BGL's throughput increases slightly slower than linear scaling.

5.7 Impact of Hyper Parameters

To verify the robustness of BGL, we evaluate training speedup under different hyperparameters (batch size, number of layers and fanouts). As shown in Figure 19, we use another two widely-used training settings in OGB leaderboards [3]. We train GraphSAGE on graph Ogbn-papers and User-Item with 4 GPUs. BGL outperforms DGL and Euler as well. The geometric mean of speedup of BGL for Euler and DGL is 10.44x and 7.50x, respectively. The computation of 2-layer GraphSAGE is faster than that with 3 layers. Hence, throughput of three systems in Figure 19b is higher than in Figure 19a.

5.8 Model Accuracy

To verify the correctness of BGL, we evaluate the test accuracy on GAT and GraphSAGE with Ogbn-products, Ogbn-papers and User-Item. Each task is trained with 100 epochs for convergence. DGL uses RO while BGL uses PO. As shown in Figure 20, BGL converges to almost the same accuracy as the original DGL but the convergence of BGL is much faster.



(a) GraphSAGE on Ogbn-products (b) GAT on Ogbn-products

(c) GraphSAGE on Ogbn-papers (d) GAT on Ogbn-papers

(e) GraphSAGE on User-Item (f) GAT on User-Item

Figure 20: BGL achieves the same accuracy as DGL, using 1 GPU.

6 Related Work

Graph Partition Algorithms. Graph partitioning is widely adopted when processing large graphs. NeuGraph [40] leverages the Kernighan-Lin [34] algorithm to partition graphs into chunks with different sparsity levels. Cluster-GCN [12] constructs the training batches based on the METIS [32] algorithm, together with a stochastic multi-clustering framework to improve model convergence. When dealing with large graphs in distributed GNN training, partition algorithms, such as Random [2, 30, 39], Round-Robin, and Linear Deterministic Greedy [6], are often used [2, 48, 55, 60]. They incur low partitioning overhead while not ensuring partition locality.

GNN Training Frameworks. In recent years, new specialized frameworks have been proposed upon existing deep learning frameworks to provide convenient and efficient graph operation primitives for GNN training [2, 17, 40, 48, 60]. Other than DGL [48], Euler [2] and PyG [17], NeuGraph [40] translates graph-aware computation on dataflow and recasts graph optimizations to support parallel computation for GNN training. However, it can only train small graphs on multi-GPUs in a single machine. AliGraph [60] is a GNN system that consists of distributed graph storage, optimized sampling operators and runtime to support both existing GNNs and in-house developed ones for different scenarios. AGL [55] is a scalable and integrated GNN system implemented on MapReduce [14] that guarantees good system properties. However, neither Aligraph nor AGL exploits GPU acceleration.

GNN Training Acceleration. Various systems have been devoted to improving GNN training performance.

Some works [9, 31, 40, 45, 49] target full-batch training. GNNAdvisor [49] explores the GNN input properties and proposes a 2D workload management and specialized memory customization for system optimizations. DGCL [9] proposes a communication planning algorithm to optimize GNN communication among multiple GPUs with METIS partition. Both projects assume graphs are stored in a single machine.

Some works [20, 38, 60] target mini-batch training. PaGraph [38] adopts static GPU caching for high-degree nodes. GNNLab [52] proposes a pre-sampling-based static caching policy. They assume that a graph can be loaded in a single machine, making them infeasible for billion-node graphs.

P^3 [20] reduces retrieving feature traffic by combining model parallelism and data parallelism. However, hybrid parallelism in P^3 incurs extra synchronization overhead. Its performance suffers when hidden dimensions exceed 128 (a com-

mon practice in modern GNNs). Further, P^3 overlooked the subgraph sampling stage, where random hashing partitioning leads to extensive cross-partition communication.

Some works try to improve graph sampling performance on GPUs, such as NextDoor [29] and C-SAW [43]. However, their performance is limited by small GPU memory. Hence, they are not suitable for giant graphs.

7 Conclusion

We present BGL, a GPU-efficient GNN training system for large graph learning that focuses on removing the data I/O and preprocessing bottleneck to achieve high GPU utilization and accelerate training. To minimize feature retrieving traffic, we propose a dynamic feature cache engine with proximity-aware ordering, and find a sweet spot of low overhead and high cache hit ratio. BGL employs a novel graph partition algorithm tailored for sampling algorithms to minimize cross-partition communication during sampling. We further optimize the resource allocation of data preprocessing using profiling-based resource isolation. Our extensive experiments demonstrate that BGL significantly outperforms existing GNN training systems by 1.91x on average. We will open-source it in the future and hope to continue evolving it with the community.

Acknowledgement

We are thankful to the anonymous NSDI reviewers and our shepherd, Ying Zhang, for their constructive feedback. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800800, Hong Kong Innovation and Technology Commission's Innovation and Technology Fund (Partnership Research Programme with ByteDance Limited, Award No. PRP/082/20FX), the National Natural Science Foundation of China under Grant U21B2022, Tsinghua University-China Mobile Communications Group Co.,Ltd. Joint Institute, and grants from Hong Kong RGC under the contracts HKU 17204619, 17208920 and 17207621.

References

[1] Deep Graph Library (DGL). https://github.com/dmlc/dgl, 2020.

[2] Euler. https://github.com/alibaba/euler, 2020.

[3] OGB Leaderboards. https://ogb.stanford.edu/docs/leader_nodeprop/, 2020.

[4] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2021.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[6] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming Graph Partitioning: An Experimental Study. *VLDB Endow.*, 11(11):1590–1603, 2018.

[7] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006.

[8] J Mark Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proc of 1st European Workshop on OpenMP*, volume 8, page 49. Citeseer, 1999.

[9] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 130–144, 2021.

[10] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Proc. of the 13th ACM European Conference on Computer Systems (EuroSys)*. ACM, 2018.

[11] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *Proc. of the 6th International Conference on Learning Representations (ICLR)*, 2018.

[12] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proc. of the 25th ACM International Conference on Knowledge Discovery & Data Mining (KDD)*, 2019.

[13] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. *In Proc. of VLDB Endow.*, 2015.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On Power-Law Relationships of the Internet Topology. *ACM SIGCOMM computer communication review*, 29(4):251–262, 1999.

[17] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. *CoRR*, abs/1903.02428, 2019.

[18] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[19] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein Interface Prediction using Graph Convolutional Networks. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[20] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 551–568, 2021.

[21] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. Knowledge Transfer for Out-of-Knowledge-Base Entities : A Graph Neural Network Approach. In *Proc. of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[22] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

[23] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[24] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis S. Liu, Georgios Theodoropoulos, and Wentong Cai. Distributed edge partitioning for trillion-edge graphs. *Proc. VLDB Endow.*, 12(13):2379–2392, 2019.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[26] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 439–450. IEEE, 2019.

[27] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR*, abs/2005.00687, 2020.

[28] Wen-bing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive Sampling Towards Fast Graph Representation Learning. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[29] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 311–326. ACM, 2021.

[30] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proc. of the VLDB Endowment*, 11(3):297–310, 2017.

[31] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proc. of Machine Learning and Systems (MLSys)*, 2020.

[32] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[33] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distributed Comput.*, 48(1):71–95, 1998.

[34] Brian W Kernighan and Shen Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proc. of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

[36] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proc. of the 5th International Conference on Learning Representations ICLR*, 2017.

[37] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[38] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. PaGraph: Scaling GNN Training on Large

Graphs via Computation-Aware Caching. In *Proc. of ACM Symposium on Cloud Computing (SOCC)*, 2020.

[39] Tianfeng Liu and Dan Li. Endgraph: An efficient distributed graph preprocessing system. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10 - 13, 2022*. IEEE, 2022.

[40] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proc. of USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[41] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 337:46–57, 2019.

[42] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[43] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. C-SAW: a framework for graph sampling and random walk on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 56. IEEE/ACM, 2020.

[44] Ketan Shah, Anirban Mitra, and Dhruv Matani. An o (1) algorithm for implementing the lfu cache eviction scheme. *no*, 1:1–8, 2010.

[45] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514, 2021.

[46] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *Proc. of the 6th International Conference on Learning Representations (ICLR)*, 2018.

[47] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. Microsoft Academic Graph: When Experts Are Not Enough. *Quantitative Science Studies*, 1(1):396–413, 2020.

[48] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR*, abs/1909.01315, 2019.

[49] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 515–531, 2021.

[50] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A Comprehensive Survey on Graph Neural Networks. *CoRR*, abs/1901.00596, 2019.

[51] Sijie Yan, Yuanjun Xiong, and Dahua Lin. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. In *Proc. of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[52] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based GNN training over gpus. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 417–434. ACM, 2022.

[53] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proc. of the 24th ACM International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[54] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *Proc. of the 8th International Conference on Learning Representations (ICLR)*, 2020.

[55] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. AGL: A Scalable System for Industrial-Purpose Graph Machine Learning. *VLDB Endow.*, 13(12):3125–3137, 2020.

[56] Muhan Zhang and Yixin Chen. Link Prediction Based on Graph Neural Networks. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[57] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep Learning on Graphs: A Survey. *CoRR*, abs/1812.04202, 2018.

[58] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv preprint arXiv:2010.05337*, 2020.

[59] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph Neural Networks: A Review of Methods and Applications. *arXiv preprint arXiv:1812.08434*, 2018.

[60] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. AliGraph: A Comprehensive Graph Neural Network Platform. *VLDB Endow.*, 12(12):2094–2105, 2019.

[61] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized Stochastic Gradient Descent. pages 2595–2603. Curran Associates, Inc., 2010.

Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training

Jie You* Jae-Won Chung* Mosharaf Chowdhury
University of Michigan

Abstract

Training deep neural networks (DNNs) is becoming increasingly more resource- and energy-intensive every year. Unfortunately, existing works primarily focus on optimizing DNN training for faster completion, often without considering the impact on energy efficiency.

In this paper, we observe that common practices to improve training performance can often lead to inefficient energy usage. More importantly, we demonstrate that there is a tradeoff between energy consumption and performance optimization. To this end, we propose Zeus, an optimization framework to navigate this tradeoff by automatically finding optimal job- and GPU-level configurations for recurring DNN training jobs. Zeus uses an online exploration-exploitation approach in conjunction with just-in-time energy profiling, averting the need for expensive offline measurements, while adapting to data drifts over time. Our evaluation shows that Zeus can improve the energy efficiency of DNN training by 15.3%–75.8% for diverse workloads.

1 Introduction

Deep neural networks (DNNs) have received ubiquitous adoption in recent years across many data-driven application domains such as computer vision [20, 38, 65], natural language processing [21, 57], personalized recommendation [32, 39], and speech recognition [33]. To effectively support such growth, DNN models are predominantly trained in clusters of highly parallel and increasingly more powerful GPUs [15, 70].

However, growing demand for computation ultimately translates to greater energy demand. For instance, training the GPT-3 model [13] consumes 1,287 megawatt-hour (MWh) [75], which is equivalent to 120 years of electricity consumption for an average U.S. household [1]. This trend continues to grow: Meta reports an increasing electricity demand for AI, despite a 28.5% operational power footprint reduction [96]. Yet, existing literature on DNN training mostly ignores energy efficiency [83].

We observe that *common performance optimization practices for DNN training can lead to inefficient energy usage*. For example, many recent works prescribe large *batch sizes* for higher training throughput [29, 84]. However, we show that maximizing raw throughput may come at the cost of lower energy efficiency. Similarly, modern GPUs allow the configuration of a *power limit* that caps its maximum power draw, but existing solutions often ignore it. Our analysis of four generations of NVIDIA GPUs shows that none of them are entirely power proportional, and drawing maximum power gives diminishing return. Indeed, carefully choosing the right batch size and GPU power limit can reduce energy consumption by 23.8%–74.7% for diverse workloads (§2.2).

Unfortunately, reducing energy consumption is not entirely free – we discover that there is a tradeoff between energy consumption and training time for a given target accuracy (§2.3). Our characterization of the energy-time Pareto frontier highlights two notable phenomena. First, for a given training job, all Pareto-optimal configurations provide varying amounts of energy reductions in comparison to blindly using the maximum batch size and GPU power limit. Second, the amount of reduction in energy consumption often has a non-linear relationship with the increase of training time. This raises a simple question: *how do we automatically identify and navigate the tradeoff between energy consumption and training time for DNN training?*

In this paper, we present Zeus to address this question. Zeus is a plug-in optimization framework that automatically configures the batch size and GPU power limit to minimize the overall energy consumption and training time for DNN training jobs (§3). Unlike some recent works that only consider GPU-specific configurations [11, 87], Zeus simultaneously considers job- and GPU-related configurations. Moreover, it does not require per-job offline profiling or prediction model training [90, 101], both of which can be prohibitive in large clusters with heterogeneous hardware and time-varying workloads [94]. Instead, Zeus takes an online exploration-exploitation approach tailored to the characteristics of DNN training workflows. That is, as new data flow into the pipeline, models need to be periodically re-trained [37], manifesting itself as *recurring jobs* in production clusters [37, 94]. Leveraging this fact, Zeus automatically explores various configurations, measures corresponding gains or losses, and continuously adjusts its actions based on its measurements (§4).

Designing such a solution is challenging due to two sources of uncertainty in DNN training. First, due to the randomness introduced from DNN parameter initialization and data loading, the energy consumed until a DNN reaches its target accuracy varies even when training is run with the exact same configuration [19, 82]. Thus, evaluating a configura-

*Equal contribution.

tion only once does not provide sufficient information about its *expected* energy consumption. Second, since both DNN models and GPUs have diverse architectures and unique energy characteristics [93], offline profiling results do not easily generalize to other DNNs and GPUs. Aggravating these challenges is the large size of the possible configuration space, with each configuration taking hours or even days to evaluate.

Zeus can efficiently determine the optimal set of knobs in the configuration space by *decoupling* the optimization of batch size and power limit without losing optimality. Specifically, it captures the stochastic nature of DNN training by formulating the batch size optimization problem as a Multi-Armed Bandit (MAB) and runs online optimization under random observations using the Thompson Sampling policy [88]. Additionally, Zeus's just-in-time (JIT) energy profiler finds the optimal power limit while training is running, making Zeus a completely online optimization framework.

We have implemented Zeus and integrated it with PyTorch [74] (§5). Evaluation on a diverse workload consisting of speech recognition, image classification, NLP, and recommendation tasks shows that Zeus reduces energy consumption by 15.3%–75.8% and training time by 60.6% w.r.t. simply selecting the maximum batch size and maximum GPU power limit. Zeus converges to optimal configuration among available ones quickly and can adapt to data drift effectively. Zeus's benefits expand to multi-GPU settings as well (§6).

In summary, we make the following contributions:

- To the best of our knowledge, we are the first to characterize the energy consumption vs. performance tradeoff for DNN training in terms of job- and GPU-specific configuration parameters.
- We present an online optimization framework that can learn from and adapt to workload dynamics over time.
- We implement and evaluate the optimizer in Zeus that integrates with existing DNN training workflows with little code change and negligible overhead, while enabling large benefits.

Zeus is open-source and available on GitHub.[2]

2 Motivation

In this section, we present an overview of energy consumption characteristics of DNN training on GPUs, opportunities for reducing energy consumption, and conclude with characterizing the tradeoff between reducing energy consumption and improving training performance.

2.1 DNN Training

Modern DNNs are trained by going over a large dataset multiple times, where each pass over the dataset is termed an *epoch* [28]. One epoch of training consists of thousands of *iterations* of gradient descent over equally sized mini-batches, with the *batch size* affecting model accuracy,[3] training throughput, and energy consumption. The performance of DNN training is often measured in terms of time-to-accuracy (TTA) for a given target accuracy [19], and increasing training throughput (or precisely goodput [77]) leads to lower TTA.

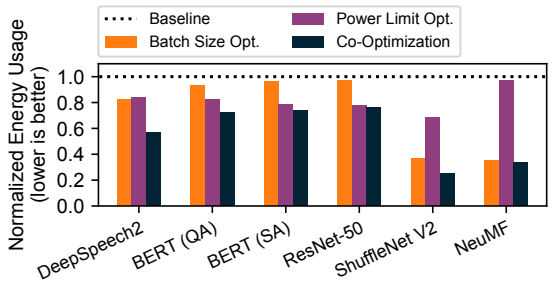


Figure 1: ***Energy usage normalized against baseline for DNN training, measured on NVIDIA V100 GPU. Baseline uses maximum power limit and the default batch size presented in the original model publication when available or the maximum batch size which can consistently reach the target metric.***

Modern DNNs are predominantly trained on increasingly more powerful GPUs, consuming more energy in the process [4, 75, 96]. Recent benchmarks show that GPUs are responsible for around 70% of the total energy consumption during DNN training [22, 41].

In production GPU clusters, as new data flow into the machine learning pipeline, DNNs need to be periodically retrained at intervals as short as every hour [37]. This need manifests itself as *recurring jobs* in the GPU cluster [37, 94].

2.2 Opportunities for Improving Energy Efficiency

We highlight two job and hardware configurations that can cause sizable energy inefficiency in DNN training: (1) batch size and (2) power limit of the GPU.

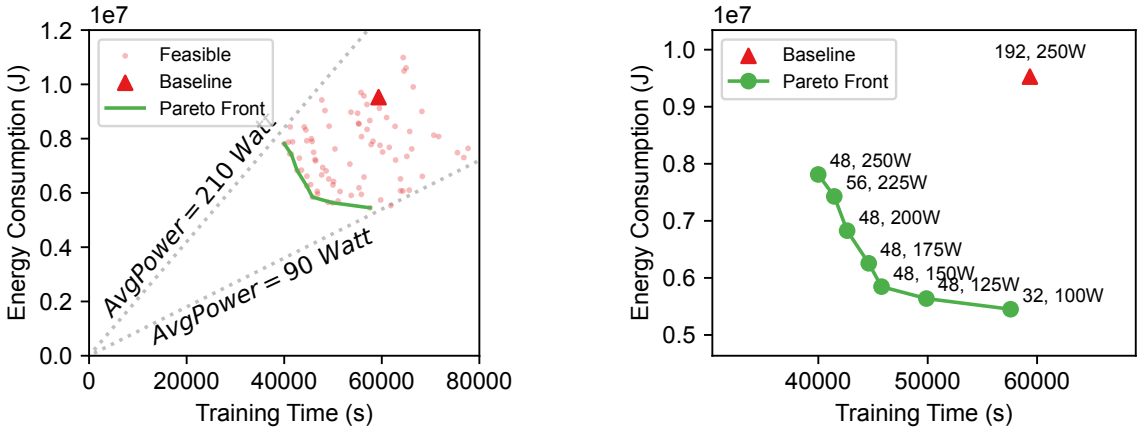
Impact of batch size on energy efficiency. The size of each mini-batch during DNN training (batch size) determines how many samples are processed in one iteration. The higher it is, the faster we can go over the entire input dataset.

We observe across diverse DNN training workloads that common choices of batch size can lead to more energy consumption for the same target accuracy. Specifically, we performed a sweep over a large range of valid batch sizes (from 8 to the maximum batch size that fits in GPU memory) for six deep learning workloads including computer vision (CV), natural language processing (NLP), recommendation, and speech recognition on an NVIDIA V100 GPU (Figure 1).[4] Section 6.1 provides details on workloads and methodology. We find that the energy-optimal batch size (Batch Size Opt. in Figure 1) can lead to 3.4%–65.0% lower energy consumption than the default choice for the same target accuracy.

[2] https://github.com/SymbioticLab/Zeus

[3] In this paper, we specifically consider the *validation accuracy* of the model, which captures how well the model generalizes to unseen data.

[4] We measure GPU power consumption using NVML [2].



(a) *Energy-Time Tradeoff* (b) *Pareto Front Zoom-in*

Figure 2: ***DeepSpeech2 trained with LibriSpeech on NVIDIA V100: (a) ETA vs. TTA. The red dots indicate all feasible configurations. The two gray dotted lines indicate two boundaries characterized by average power consumption. The green line indicates the Pareto frontier over all configurations. (b) Zoom-in view on the Pareto frontier in (a), with batch size and power limit annotated on each data point.***

Impact of GPU power limit on energy efficiency. Setting a GPU's power limit will have the device internally trigger dynamic voltage and frequency scaling (DVFS) such that its power draw does not exceed the power limit [69]. If not set manually, the power limit is at the maximum by default. We performed a sweep over a wide range of GPU power limits[5] for the aforementioned setup. We found that the optimal energy consumption (Power Limit Opt. in Figure 1) may happen at a lower power limit than the maximum and can reduce energy consumption by 3.0%–31.5%.

Joint optimization. As Figure 1 shows, we can achieve even more energy savings (23.8%–74.7% reduction) if we jointly optimize both configurations. Note that we observed similar opportunities for reducing energy consumption for other generations of GPUs as well (Figure 15 in Appendix A).

2.3 Energy-Performance Tradeoffs

Opportunities for reducing DNN training energy consumption comes with a cost. When optimized for energy efficiency, DNN training performance (time-to-accuracy, or TTA) may be impacted. In the following, we characterize this tradeoff.

We define the energy consumption of DNN training until it reaches its target accuracy as its *energy-to-accuracy* (ETA):

$$\texttt{ETA}(b,p) = \texttt{TTA}(b,p) \times \texttt{AvgPower}(b,p), \qquad (1)$$

where p denotes the GPU power limit, b the batch size, and $\texttt{AvgPower}(b,p)$ the average power consumption during training with configuration (b,p). Similar to TTA, ETA captures the end-to-end goal of DNN training.

Note that $\texttt{AvgPower}(b,p)$ is not the same as the GPU power limit. When changes in configuration (b,p) lead to an increase in TTA, ETA does not always follow because $\texttt{AvgPower}(b,p)$ can decrease more. This motivates us to investigate the *tradeoff* between ETA and TTA.

Tradeoff between ETA and TTA. We characterize and elaborate on this tradeoff using DeepSpeech2 trained on LibriSpeech as an example (Figure 2). It shows a scatter plot of (TTA, ETA) for the batch size and power limit sweep experiments in Section 2.2. We observe similar results for other workloads as well (Figure 16 in Appendix B).

Let us start with Figure 2a, where each data point denotes the (TTA, ETA) of training the model for a certain configuration.While sweeping the configurations, we focus on the boundary of all feasible (TTA, ETA) pairs. We find them to be bounded by two straight lines characterizing the average GPU power consumption. When the GPU is under heavy load, the (TTA, ETA) data points appear closer to 210W. On the other hand, when the GPU is under lighter load, its average power consumption tends closer to 90W, which is close to the GPU's idle power consumption of 70W. More importantly, we find a curve along which all (TTA, ETA) pairs achieves Pareto optimality [16], for which we cannot improve ETA without sacrificing TTA, and vice versa.

Now let us take a closer look at the Pareto frontier in Figure 2b, with the configurations used during training annotated along each data point. We highlight two takeaways:

1. These results show that baseline configurations can lead to suboptimal energy efficiency (§2). Moreover, it shows that blindly going for high batch size and power limit configurations can lead to suboptimal TTA as well.
2. There exists a tradeoff between ETA and TTA, with different optimums for each. The configuration optimizing the ETA (b =32, p =100W) is different from that optimizing TTA (b =48, p =250W).

[5] From the minimum to the maximum power limit allowed by NVIDIA System Management Interface [3]; from 100W to 250W for NVIDIA V100.

3 Zeus Overview

Zeus is an optimization framework that navigates the ETA-TTA tradeoff by automatically configuring the batch size and GPU power limit of recurring DNN training jobs. It enables developers to optimize energy and/or performance metrics using a single knob.

3.1 Optimization Metric

Defining a good cost metric for users to express their preference in this tradeoff is critical in designing Zeus. We propose a simple cost metric:

$$C(b,p;\eta) = \eta \cdot \texttt{ETA}(b,p) + (1-\eta) \cdot \texttt{MAXPOWER} \cdot \texttt{TTA}(b,p) \tag{2}$$

Here η is the parameter specified by the user to express the relative importance of energy efficiency and training performance (throughput). When $\eta = 0$, we are only optimizing for time consumption, whereas when $\eta = 1$, we are only optimizing for energy consumption. `MAXPOWER` is the maximum power limit supported by the GPU, a constant introduced to unify the units of measure in the cost metric.

3.2 Challenges in Picking the Optimal Configuration

Combining Equations 1 and 2, we have:

$$C = (\eta \cdot \texttt{AvgPower}(b,p) + (1-\eta) \cdot \texttt{MAXPOWER}) \cdot \texttt{TTA}(b,p). \tag{3}$$

Picking the optimal configuration(s) to minimize the energy-time cost C for DNN training is challenging because the search space $[b \times p]$ is large and obtaining the cost of each configuration is difficult. This is because it is hard to determine the value of both $\texttt{AvgPower}(b,p)$ and $\texttt{TTA}(b,p)$ efficiently, as explained below.

- **Complex power consumption model:** The total energy consumption of a GPU is affected in a non-linear fashion by both the characteristics of the workload such as the number of instructions and memory accesses, as well as the GPU hardware configurations such as the frequency and voltage of the cores and memory on board [6, 46]. Existing efforts estimate GPU energy consumption based on instruction- or kernel-level information [43, 64], which are architecture-specific and workload-dependent.
- **Stochastic nature of DNN training:** Modeling and predicting the duration for training a specific model to target accuracy (TTA) is known to be difficult [31]. Moreover, the randomness introduced during model initialization and data loading leads to variations of TTA, even when the same job is run on the same GPU with the same configuration – TTA variations can be as large as 14% [19].

Fortunately, DNN training jobs often recur in production clusters [37, 94]. This provides opportunities for empirical estimation through repeated measurements across recurrences of the same training job.

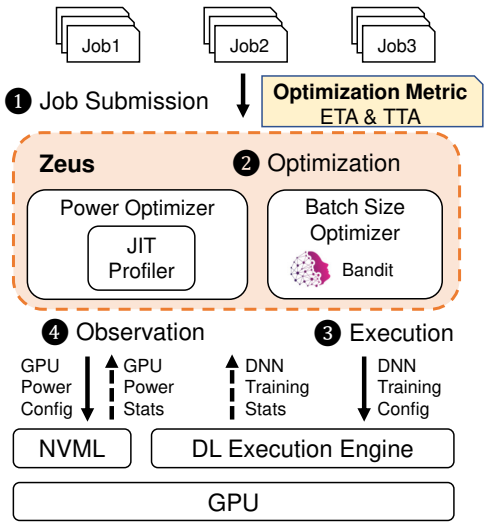


Figure 3: *Zeus Workflow.*

3.3 Architectural Overview

At a high-level, Zeus takes an online exploration-exploitation approach to minimize the aggregate cost of recurrent DNN training jobs. Zeus addresses the aforementioned challenges with two key components:

1. A just-in-time (JIT) online profiler, which efficiently profiles the energy characteristics of the training job online.
2. Multi-Armed Bandit (MAB) with Thompson sampling, which allows us to embrace the stochastic nature of DL training and optimize under uncertainty while also adapting to changing workloads such as data drift.

The combination of the JIT profiler and MAB makes Zeus a fully online solution, allowing it to immediately begin optimizing for incoming jobs.

Workflow of Zeus. Figure 3 shows an overview of the high-level workflow of Zeus. In a production environment, users submit ❶ recurrent DNN training jobs (a tuple of data, model, optimizer, and the target validation metric) to Zeus, along with a set of feasible batch sizes $\mathcal{B}$ and power limits $\mathcal{P}$ to explore. Zeus then predicts ❷ the optimal batch size and power limit configuration based on past execution history, and launches ❸ the training job with that configuration. During and after the training process, ❹ statistics about DNN training (e.g., validation metric) and GPU power consumption are collected and fed back to the Zeus optimizer. The Zeus optimizer learns from the feedback and adjusts its internal states. The training job will be terminated upon either reaching target metric or exceeding a stopping threshold determined by Zeus. The whole process is an automated feedback loop that minimizes the key objective of energy-time cost.

Building Zeus requires both algorithm design and systems support. Next we describe the core optimization algorithm details (§4) and Zeus implementation highlights (§5).

4 Zeus Algorithm Design

In this section, we delve into the details of how Zeus selects the best batch size and GPU power limit to optimize the overall cost of recurrent DNN training tasks. We first present the optimization problem formulation and how we decouple the optimizations of batch size and power limit (§4.1). Next, we show how to optimize power limit (§4.2) and batch size (§4.3) under the decoupled framework. We conclude by discussing how we address common challenging scenarios (§4.4).

4.1 Problem Formulation

The objective of Zeus is to minimize the cost of a recurring job by automatically exploring the feasible set of batch sizes $\mathcal{B}$ and power limits $\mathcal{P}$. In essence, we neither want to incur too much cost searching for the optimal configuration, nor do we want to miss it. Minimizing the *cumulative* cost of the job over recurrences captures the implicit tradeoff between exploration and exploitation. Put formally in terms of the cost function defined by Equation 2, our objective becomes

$$\begin{aligned} \min_{b,p} \quad & \sum_{t=1}^{T} C(b_t, p_t; \eta) \\ \text{s.t.} \quad & b_t \in \mathcal{B}, p_t \in \mathcal{P}, \forall t \in [1, T], \end{aligned} \tag{4}$$

where b_t and p_t respectively denote the batch size and power limit chosen at the tth recurrence of the job, and b and p are vectors of length T.

This is a challenging problem without modification, mainly because the size of the search space can be in the order of hundreds, and each value of $C(b,p;\eta)$ inside the search space can only be obtained by running DNN training until it reaches the target metric. However, further expanding the cost function (Equation 3) allows us to *decouple* the exploration of batch size and power limit, making the problem more tractable:

$$\begin{aligned} & C(b,p;\eta) \\ &= (\eta \cdot \texttt{AvgPower}(b,p) + (1-\eta) \cdot \texttt{MAXPOWER}) \cdot \texttt{TTA}(b,p) \\ &= \texttt{Epochs}(b) \cdot \frac{\eta \cdot \texttt{AvgPower}(b,p) + (1-\eta) \cdot \texttt{MAXPOWER}}{\texttt{Throughput}(b,p)}. \end{aligned} \tag{5}$$

where $\texttt{Epochs}(\texttt{b})$ denotes the number of epochs needed to reach the target, and $\texttt{Throughput}(\texttt{b},\texttt{p})$ epochs per second.

We find two key insights that allow the decoupling of batch size b and power limit p:

1. Given b, $\texttt{AvgPower}(b,p)$ and $\texttt{Throughput}(b,p)$ can be profiled quickly during training for all possible choices of p. This is due to the iterative nature of DNN training, yielding stable power and throughput estimations even with a small number of iterations.
2. $\texttt{Epochs}(b)$ is not affected by the choice of p as changing the power limit does not change what is computed.

This implies that the optimal power limit, given any batch size, can be determined independently based on online profiling. Moreover, since any choice of batch size is automatically accompanied by the optimal power limit, our search space is reduced to the set of batch sizes $\mathcal{B}$.

Formally put, we have decoupled the problem in Equation 4 into an equivalent two-level optimization problem

$$\min_{b \in \mathcal{B}^T} \sum_{t=1}^{T} \texttt{Epochs}(b_t) \cdot \texttt{EpochCost}(b_t;\eta) \tag{6}$$

where

$$\begin{aligned} & \texttt{EpochCost}(b_t;\eta) \\ &= \min_{p_t \in \mathcal{P}} \frac{\eta \cdot \texttt{AvgPower}(b_t,p_t) + (1-\eta) \cdot \texttt{MAXPOWER}}{\texttt{Throughput}(b_t,p_t)}. \end{aligned} \tag{7}$$

When a job arrives, Zeus will first decide which batch size to use based on Equation 6 (§4.3). Then, based on the batch size, Zeus will pick the optimal power limit based on Equation 7 (§4.2).

4.2 Optimizing the Power Limit

We start with how Zeus determines the optimal power limit based on Equation 7, given a choice of the batch size. As highlighted earlier, we leverage the iterative nature of DNN training and the recurrent nature of jobs in production DNN training workflows.

When a job with batch size decision b is submitted, our just-in-time (JIT) profiler is triggered and checks if this batch size had been profiled before. For an unseen batch size b, it profiles $\texttt{AvgPower}(b,p)$ and $\texttt{Throughput}(b,p)$ for all possible power limits p during the first epoch of the job by partitioning the epoch into slices at iteration boundaries and dynamically changing the GPU power limit for each slice. The profile information is fed back to Zeus, and the optimal power limit of the batch size is determined by solving Equation 7. The rest of the epochs are executed with the optimal power limit. Our *online* JIT profiling approach consumes strictly less time and energy compared to offline profiling before running the job, because the profiling process itself contributes to training without affecting its accuracy. We show that JIT profiling incurs negligible overhead in Section 6.5.

4.3 Optimizing the Batch Size

Now we focus on how Zeus determines the batch size b_t for each job recurrence t that optimizes Equation 6. As seen in Section 4.2, $\texttt{EpochCost}(\texttt{b}_\texttt{t};\eta)$ is a cheap and deterministic function that identifies the optimal power limit for any batch size b_t and returns the optimal cost of one epoch. Thus, we may limit our exploration to choosing the optimal batch size because whichever batch size we choose, the optimal power limit will accompany it.

Due to the unpredictable and stochastic nature of DNN training, picking out the optimal batch size without adequate exploration is difficult. Hence, a good solution must (1) incorporate such nature of DNN training into its exploration process, and (2) intelligently tradeoff the cost of exploring for

Input: Batch sizes $\mathcal{B}$
Belief posterior parameters $\hat{\mu}_b$ and $\hat{\sigma}_b^2$
Output: Batch size to run b^*

Function `Predict`$(\mathcal{B}, \hat{\mu}_b, \hat{\sigma}_b^2)$**:**
1 **foreach** batch size $b \in \mathcal{B}$ **do**
  /* Sample from the belief distribution */
2 Sample $\hat{\theta}_b \sim \mathcal{N}(\hat{\mu}_b, \hat{\sigma}_b^2)$
3 **end**
  /* Select the arm with smallest mean cost sample */
4 $b^* \leftarrow \text{argmin}_b\, \hat{\theta}_b$

Algorithm 1: Gaussian Thompson Sampling: Choosing the next batch size to run (`Predict`)

potentially better batch sizes and the gain of exploiting batch sizes that are already known to be good.

Grid search is suboptimal. We argue that exhaustively going through all batch sizes and selecting the one with the smallest cost is still suboptimal due to the stochastic nature of DNN training. That is, because the cost of a DNN training job can differ even when executed with the exact same configurations, it must be modeled as a *cost distribution* with unknown mean and variance. Although performing several trials for each batch size may yield a better estimation of the mean cost, such a strategy leads to *high exploration cost* because it does not quickly rule out obviously suboptimal batch sizes.

Multi-Armed Bandit formulation. Zeus aims to explore the cost of different batch sizes and converge to the optimal batch size, while not incurring too much exploration cost.

Zeus formulates the problem as a Multi-Armed Bandit (MAB) with T trials and B arms, where each trial corresponds to a recurrence of the job and each arm to a batch size in $\mathcal{B}$. MAB is a good fit to our problem scenario in that it captures the stochasticity of DNN training by modeling the cost of each batch size as a random variable. Specifically, we choose the Gaussian distribution [81] due to its representational flexibility. The objective of the MAB formulation is to minimize the *cumulative cost regret* defined as

$$\sum_{t=1}^{T} \texttt{Regret}(b_t; \eta) \tag{8}$$

where the regret of choosing b_t is defined as

$$\texttt{Regret}(b_t; \eta) = \texttt{Epochs}(b_t) \cdot \texttt{EpochCost}(b_t; \eta) - \min_{b,p} \texttt{Cost}(b, p; \eta). \tag{9}$$

Minimizing cumulative cost regret aligns with our objective in Equation 6.

Thompson Sampling. We adopt the Thompson Sampling [81] policy for the MAB formulation to tradeoff exploration and exploitation, not only because it is known to perform well in practice [17, 81] and had successful adoption recently [58, 67], but also because its modeling assumptions fit our problem scenario well.

Input: Batch size b and observed cost C
Previous cost observations C_b for b
Belief prior parameters $\hat{\mu}_0$ and $\hat{\sigma}_0^2$
Output: Belief posterior parameters $\hat{\mu}_b$ and $\hat{\sigma}_b^2$

Function `Observe`$(b, C, \mathcal{C}_b, \hat{\mu}_0, \hat{\sigma}_0^2)$**:**
  /* Add the most recent cost observation to history */
1 $\mathcal{C}_b \leftarrow \mathcal{C}_b \cup \{C\}$
  /* Compute the variance of the cost */
2 $\tilde{\sigma}^2 \leftarrow Var(\mathcal{C}_b)$
  /* Compute the belief distribution's posterior variance */
3 $\hat{\sigma}_b^2 \leftarrow \left(\frac{1}{\hat{\sigma}_0^2} + \frac{|C_b|}{\tilde{\sigma}^2}\right)^{-1}$
  /* Compute the belief distribution's posterior mean */
4 $\hat{\mu}_b \leftarrow \hat{\sigma}_b^2 \left(\frac{\hat{\mu}_0}{\hat{\sigma}_0^2} + \frac{Sum(C_b)}{\tilde{\sigma}^2}\right)$

Algorithm 2: Gaussian Thompson Sampling: Updating the belief distribution (`Observe`)

At a high level, Thompson Sampling is an online procedure that refines its *belief* about the *mean cost* of each arm (batch size) based on experience. At each recurrence, the belief is used to pick the arm with the lowest estimated mean cost (Algorithm 1), and the belief is updated based on the actual cost observed (Algorithm 2).

Specifically, the cost distribution is modeled as a Gaussian distribution with unknown mean θ_b. Then, the belief about θ_b is modeled with its conjugate prior distribution, which is also a Gaussian distribution [24]. That is, $\theta_b \sim \mathcal{N}(\hat{\mu}_b, \hat{\sigma}_b^2)$. Here it is important to note that $1/\hat{\sigma}_b^2$ can be thought as of how confident the policy is in its belief about that arm, with the confidence increasing as it accumulates more observations of the cost of choosing that arm. Then, Thompson Sampling automatically balances exploration and exploitation by choosing the arm with the smallest mean cost sample $\hat{\theta}_b \sim \mathcal{N}(\hat{\mu}_b, \hat{\sigma}_b^2)$ (Algorithm 1). With low confidence (high variance), $\hat{\theta}_b$ will be dispersed across a wider range of costs, having higher chances of getting chosen even if some of its initial observations showed high cost. In contrast, when the arms observed a lot of cost samples and the confidence is high (low variance), $\hat{\theta}_b$ is likely to be centered around the mean observed cost, allowing the exploitation of arms that are known to be good. After the actual cost of an arm is observed, the belief parameters of that arm are updated using the Bayes Rule [81] (Algorithm 2).

The belief prior parameters $\hat{\mu}_0$ and $\hat{\sigma}_0^2$ reflect prior belief about the mean cost of using the batch size for training and the confidence of such belief. Hence, the choice of prior parameters serve as a way to initialize the arms such that they reflect prior knowledge about the cost of each arm. If such

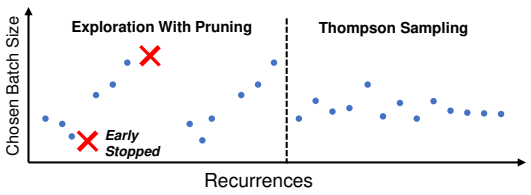


Figure 4: ***An example of batch sizes chosen by Zeus for a recurring job. Each point is a recurrence. During pruning, Zeus explores each batch size 2 times in order to observe the cost variance (Line 2 in Algorithm 2).***

information is not available, which is our default assumption, it is also possible to initialize the arms with a flat prior that assumes no prior knowledge – in our case, this is a Gaussian distribution with zero mean and infinite variance.

In contrast to grid search, our formulation using MAB and Thompson Sampling meets the two requirements mentioned earlier. That is, MAB inherently incorporates the stochastic nature of DNN training in that it models cost as a random variable. Moreover, Thompson Sampling can quickly rule out batch sizes that are obviously suboptimal because the probability of a smaller mean cost being sampled from an arm that observed noticeably large cost is low.

4.4 Extensions for Challenging Scenarios

Handling unknown cost variance. Unlike conventional Gaussian Thompson Sampling applications, we may not assume that the variances of the cost of each arm are known. That is, the cost variance (i.e., how much the cost will fluctuate even when training is run with the same batch size) is not known before any observation. Moreover, the cost variance depends not only on the batch size, but also on the DNN's robustness to the randomness in parameter initialization and data loading, making it difficult to quantify at the time the MAB is constructed. Hence, our approach is to *learn* the cost variance as we observe cost samples (Line 2 in Algorithm 2).

Handling stragglers during exploration. There may be cases where an exploratory job does not reach the target metric within a reasonable amount of cost, especially during the earlier exploration stage. To handle this, we employ *early stopping* and *pruning*. The intuition is that if a batch size does not reach the target metric even after incurring an exceedingly large cost, it is highly unlikely to be the optimal one.

For early stopping, we define a cost threshold $\beta \cdot \min_t C_t$, meaning that when the cost of the current job is to exceed β times the minimum cost observed so far, we stop the job and retry with another batch size. Here β is a parameter to account for the stochastic nature of DL training. By default, we choose $\beta = 2$, with which we should be able to tolerate variations of TTA between different runs of the same configuration, which is usually less than the 14% [19].

For pruning, as illustrated in Figure 4, we begin with the default batch size provided by the user and first try smaller batch sizes until we meet the minimum batch size or a batch size that fails to reach the target metric before the early stopping threshold. The same process is repeated for batch sizes larger than the default batch size. Then, only the batch sizes that reached the target metric are kept in the batch size set we explore. After performing an initial round of pruning, the default batch size is updated to be the one with the smallest cost observed, and we perform pruning once more starting from the new default batch size.

The intuition behind our batch size pruning approach is the convexity we observe in the BS-ETA curve around the optimal batch size (See Figure 5). Moreover, pruning allows Zeus to quickly rule out batch sizes that are noticeably suboptimal (typically too large, leading to more training epochs and loss of accuracy [27, 49], or too small, yielding gradients that are too noisy [80]), thus cutting down the cost of exploration.

The overall process is depicted in Algorithm 3.

Handling concurrent job submissions. Classic multi-armed bandit scenarios assume that the MAB immediately observes the cost of pulling an arm. However, in a DNN

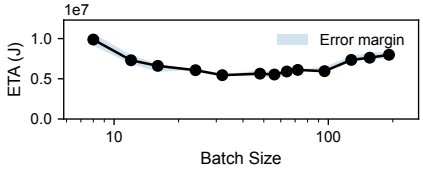


Figure 5: ***ETA of each batch size for DeepSpeech2 trained on LibriSpeech. Plots for rest of the workloads are in the Appendix C.***

Input: Set of batch sizes $\mathcal{B}$
Default batch size b_0
Belief prior parameters $\hat{\mu}_0$ and $\hat{\sigma}_0^2$

/* Exploration With Pruning */
Recurrence $t \leftarrow 0$
repeat *2* **times**
  Explore b_0
  Explore $b < b_0$ until convergence failure
  Explore $b > b_0$ until convergence failure
  $\mathcal{B} \leftarrow \{b : b \text{ converged}\}$
  $b_0 \leftarrow b$ with smallest cost observed
  $t \leftarrow t + |\mathcal{B}|$
end
/* Thompson Sampling */
while $t \leq T$ **do**
  $b^* \leftarrow Predict(\mathcal{B}, \hat{\mu}_b, \hat{\sigma}_b^2 \; \forall b \in \mathcal{B})$
  Run job with batch size b^* and add cost to $\mathcal{C}_b$
  /* Update our belief of the mean cost */
  $\hat{\mu}_b, \hat{\sigma}_b^2 \leftarrow Observe(b, \mathcal{C}_b, \hat{\mu}_0, \hat{\sigma}_0^2)$
  $t \leftarrow t + 1$
end

Algorithm 3: Gaussian Thompson Sampling Batch Size Optimizer.

training cluster, recurring jobs may overlap in their execution when a later job starts before the completion of an earlier job. In this case, the MAB does not get to observe the cost of the earlier job at the time it has to decide the batch size for the later job. For deterministic policies like [8, 56], this leads to duplication exploration of the same batch size back-to-back, reducing the efficiency of exploration.

However, Thompson Sampling naturally mitigates this problem without modification because deciding the next batch size to explore (`Predict`) is a random function. That is, because Thompson Sampling *samples* the estimated mean cost from each arm's belief distribution and returns the arm with the lowest sampled value, concurrent jobs can run different batch sizes even if there was no information gained between the invocations of `Predict`. This is especially the case during the early stage of Thompson Sampling when the arms' belief distributions have large variances (low confidence), losing little exploration efficiency.

During the short initial pruning phase, we run concurrent job submissions with the best-known batch size at that time. As the best batch size constantly updates throughout the exploration stage, this strategy fairly distributes the additional exploration opportunities from concurrent job submissions to batch sizes that are known to converge. We evaluate Zeus's efficacy on handling concurrent job submissions in Section 6.3.

Handling data drift. In production training clusters, the data on which the model is trained shifts, which is one of the reasons why re-training is triggered [61, 63]. The implication of drift in the perspective of the MAB is that the cost distribution of each arm is non-stationary.

Thompson Sampling allows a simple modification that allows us to handle non-stationary cost distributions. Since older cost observations become less and less relevant, we only operate on a window of N most recent cost observations [10], and the belief distributions will not take old observations into account. Unlike exponential decay, windowing also allows the cost variance of the most recent observations to be estimated directly. When old history entries are evicted, computing the new parameters of the arm is also cheap thanks to the conjugate prior property. This way, Zeus transparently adapts to data drifts in an *online* manner, as we show in Section 6.4.

5 Zeus Implementation

Zeus is implemented as a Python library that can be imported into DNN training scripts. The `ZeusDataLoader` class integrates with PyTorch [74]. The class profiles power and throughput online by slicing epochs in iteration boundaries and invoking the NVML [2] library for power limit configuration and profiling. We have observed that five seconds of profiling for each power limit is enough to yield stable results. With the information, the optimal power limit can be automatically determined and applied. Moreover, `ZeusDataLoader` monitors the cost incurred by training and early stops the job if needed. Listing 1 shows an example training loop integrated with Zeus.

```python
from zeus import ZeusDataLoader

train_loader = ZeusDataLoader(
    train_set, batch_size, max_epochs, target_metric)
eval_loader = ZeusDataLoader(eval_set, batch_size)

for epoch in train_loader.epochs():  # may early stop
    for batch in train_loader:
        # Learn from batch
    for batch in eval_loader:
        # Evaluate on batch
    train_loader.report_metric(validation_metric)
```

Listing 1: ***Zeus Integration Example***

Observer Mode. `ZeusDataLoader` supports *Observer Mode*, where it profiles the power consumption and throughput of each power limit and determines the optimal one, but keeps the power limit at the maximum. By doing so, without affecting time or energy consumption, `ZeusDataLoader` reports how much time and energy the job *would have* consumed if the power limit were the optimal one, allowing the user to get an idea of the impact of using Zeus. We believe that such a feature can encourage Zeus's adoption by informing users of its potential savings.

6 Evaluation

We evaluate Zeus's effectiveness in terms of navigating the energy-time tradeoff. Our key findings are as follows:

1. Zeus reduces energy consumption by 15.3%–75.8%. It achieves this by trading off small performance for jobs that are already throughput-optimal; otherwise, it reduces training time by up to 60.1% too (§6.2).
2. Zeus quickly converges to optimal configurations (§6.2).
3. Zeus can handle workloads with data drift (§6.4) and overall incurs low overhead (§6.5).
4. Zeus scales to multi-GPU settings (§6.6) and provides consistent savings across four generations of GPUs (§6.7).

6.1 Experimental Setup

Testbed Setup. We evaluate Zeus with four generations of NVIDIA GPUs as specified in Table 2.

Workloads. Table 1 summarizes our workloads. The default batch size (b_0) is chosen from the original model publication when available; otherwise, it is set to be the maximum batch size which consistently achieves the target accuracy.

In terms of learning rate, models trained with the Adadelta [99] optimizer do not require an initial learning rate. For optimizers that do require an initial learning rate, we made our best effort in choosing a batch size and learning rate pair that achieves reasonable accuracies by experimenting with values from the original publication of the model and those discovered by popular DL frameworks [95].

After collecting the initial batch size and learning rate pairs,

Task	Dataset	Model	Optimizer	b_0	Target Metric
Speech Recognition	LibriSpeech [73]	DeepSpeech2 [33]	AdamW [62]	192	WER = 40.0%
Question Answering	SQuAD [79]	BERT (QA) [21]	AdamW [62]	32	F1 = 84.0
Sentiment Analysis	Sentiment140 [26]	BERT (SA) [21]	AdamW [62]	128	Acc. = 84%
Image Classification	ImageNet [20]	ResNet-50 [38]	Adadelta [99]	256	Acc. = 65%
Image Classification	CIFAR-100 [53]	ShuffleNet-v2 [65]	Adadelta [99]	1024	Acc. = 60%
Recommendation	MovieLens-1M [34]	NeuMF [39]	Adam [51]	1024	NDCG = 0.41

Table 1: ***Models and datasets used in our evaluation. The provided target metrics is the target for each training job. Here b_0 denotes the default batch size presented in the original work when feasible, otherwise we choose the maximum batch size which can consistently reach the target. The BERT(QA) and BERT(SA) means fine-tuning BERT on the tasks of question answering and sentiment analysis, respectively.***

Node	GPU Specification		Host Specification	
HPE Apollo 6500 Gen10 Plus A40 × 4	Model VRAM mArch.	A40 PCIe 48GB Ampere	CPU RAM Disk	AMD EPYC 7513 512GB DDR4-3200 960GB NVMe SSD
CloudLab [23] r7525 V100 × 2	Model VRAM mArch.	V100 PCIe 32GB Volta	CPU RAM Disk	AMD EPYC 7542 512GB DDR4-3200 2TB 7200rpm HDD
Chameleon Cloud [48] RTX6000	Model VRAM mArch.	RTX6000 24GB Turing	CPU RAM Disk	Xeon Gold 6126 192GB 256GB SSD
Chameleon Cloud [48] P100 × 2	Model VRAM mArch.	P100 16GB Pascal	CPU RAM Disk	Xeon E5-2670 v3 128GB 1TB HDD

Table 2: ***Hardware used in the evaluation.***

when we scale the batch size, we applied Square Root Scaling [42] for adaptive optimizers such as Adam [51] following recent theoretical results [30].

Baselines. We compare against the following baselines:

1. *Default ($b = b_0$, $p =$ `MAXPOWER`).* This is often the default configuration used by practitioners, where the GPU power limit is set to, or rather *not changed from*, the maximum. This is the most conservative baseline with no exploration.
2. *Grid Search with Pruning.* This one tries out one configuration of (b, p) for each recurrence of the job and selects the best one. We optimize naïve grid search by having it prune out batch sizes that failed to reach the target metric.

Metric. Our primary metrics are ETA (energy consumption) and TTA (training time). Ideally, we want to reduce both; but due to their tradeoff, sometimes it may not be possible to simultaneously do both.

Defaults. All experiments are done on NVIDIA V100 GPUs, unless otherwise mentioned. By default, we highlight $\eta = 0.5$ to strike a balance between ETA and TTA. Later, we sweep η from 0 to 1 (§6.7). The early-stopping threshold β is set to 2, and we also sweep β from 1.5 to 5 (§6.7).

Methodology. Due to resource constraints and environmental concerns, we cannot afford to repeatedly train all of our workloads with various configurations end-to-end hundreds of times sequentially. However, similar to how Zeus *decouples* the exploration of batch size and power limit, we may apply the same decoupling in our experimentation. That is, we instead take a trace-driven approach, where we collect two kinds of trace data:

1. Training trace. We train all possible combinations of models and batch sizes until convergence and record the number of epochs the model took to reach its target accuracy. We repeat this with four different random seeds for every combination to capture the stochasticity in DNN training.
2. Power trace. We use our JIT profiler to collect the throughput and average power consumption of all possible combinations of model, batch size, and power limit.

We then replay these traces when we need to train a model and reconstruct its TTA and ETA values in order to evaluate the decisions made by Zeus and baselines. Moreover, since we have access to all the possible choices and their outcomes, we also know the optimal choice. Therefore, with the traces, we can evaluate the regret achieved by Zeus and baselines.

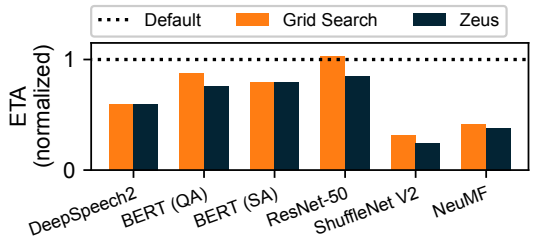
Note that Zeus does not directly learn from these traces (which would be offline-profiling), but instead only learns from the *replay* of these traces in an online fashion.

While the aforementioned trace-driven method is used widely throughout our evaluation, we run Zeus end-to-end for the evaluation of handling data drift (§6.4) because it is more expensive to construct the trace for the drifting dataset.

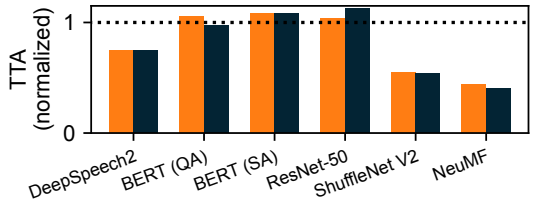
6.2 Zeus Performance

In this section, we evaluate the performance of Zeus in terms of energy consumption and training time as well as the convergence characteristics of our Multi-Armed Bandit algorithm. Each experiment is run across multiple recurrences of DNN training jobs. We select the recurrence number to be $2 \cdot |\mathcal{B}| \cdot |\mathcal{P}|$, so that the Grid Search baseline finishes exploration and also has plenty of chances to exploit its choice.

Improvements in ETA. Figure 6a shows the energy consumption (ETA) of the last five recurrences of Zeus and Grid Search w.r.t. the Default baseline, aiming to compare the final point each approach converged to. Zeus reduces energy consumption (ETA) by up to 15.3%–75.8% w.r.t. the baseline. This is also comparable to the reduction we found by exhaustively searching through all the configurations in Section 2 as well as by using Grid Search.

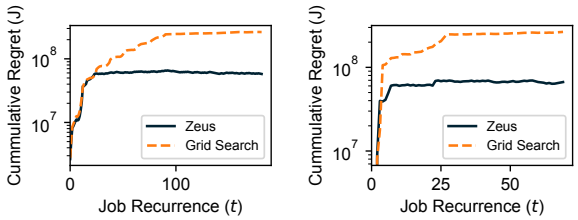


(a) *Energy Consumption*



(b) *Training Time*

Figure 6: ***Zeus reduces energy consumption for all workloads. (a) energy consumption, (b) training time of each workload, normalized by the Default baseline. Results are computed with the last five recurrences, capturing the knobs each method converged to.***



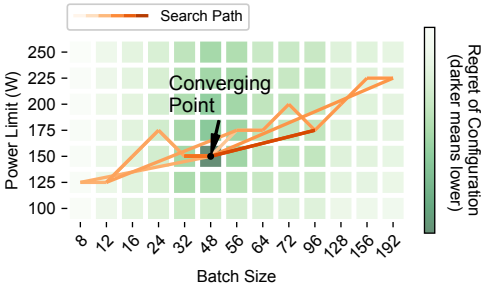
(a) *DeepSpeech2* (b) *ResNet-50*

Figure 7: ***Cumulative regret of Zeus vs. Grid Search for (a) DeepSpeech2 and (b) ResNet-50.***

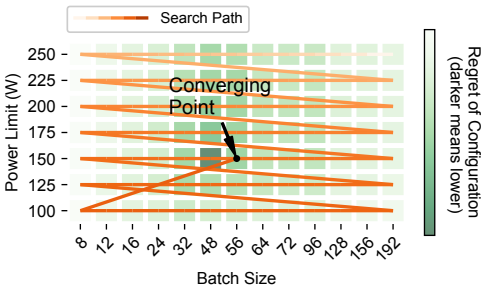
Tradeoff with TTA. Figure 6b shows the time consumption (TTA) of the last five recurrences of Zeus and Grid Search w.r.t. the Default baseline. Even though Zeus reduces training time by up to 60.1%, for some workloads TTA is increased by 12.8% (Figure 6b). This is due to the tradeoff between ETA and TTA, which is the central focus of this paper. This is especially true for workloads with a b_0 tuned for minimizing training time, where there is little room for TTA improvement.

Cumulative regret. While Zeus and Grid Search perform close to each other, Zeus uses significantly smaller amount of resources to converge. As a bandit-based solution, the effectiveness of our algorithm can be quantified via regret, the difference between the decision selected and the optimal choice (Equation 9 in Section 4.3).

Figure 7 shows the cumulative regret of Zeus and Grid Search for DeepSpeech2 and ResNet-50. The optimal configuration is identified separately by an exhaustive parameter sweep. We observe that in both workloads, Zeus is able



(a) *Zeus*



(b) *Grid Search*

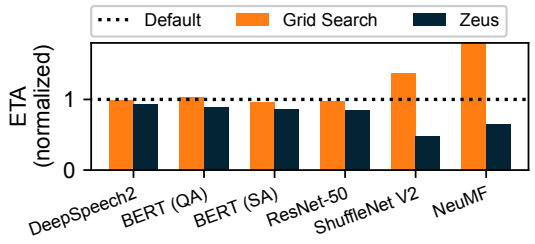
Figure 8: ***Search paths of (a) Zeus and (b) Grid Search for Deepspeech2. The heatmap in the background shows the regret of each (Batch Size, Power Limit) configuration. Darker background denotes lower regret and therefore better configuration. The colored line with shifting color shows the search path, with darker color being later recurrences.***

to achieve better regret from the first job recurrence. Zeus reaches the plateau in the cumulative regret earlier than Grid Search, which means it converges to the optimal solution earlier. We observe similar results for other workload as well (Appendix D). In the worst case, Grid Search results in 72× more cumulative regret than Zeus until convergence.

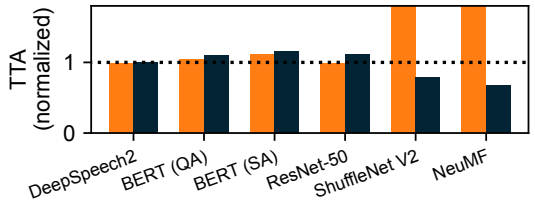
Convergence to a Pareto-optimal configuration. Despite having no information about the application beforehand, Zeus learns the energy characteristics of it online in a few iterations. Figure 8 shows the search path of Zeus and Grid Search during training DeepSpeech2. Due to the decoupling in the optimization of power limit and batch size, Zeus explores the configuration space more efficiently and converges to the optimal configuration much faster. We observe similar results for other workloads (see Appendix E). Moreover, in Figure 8b we observe that Grid Search may not even converge to optimal configuration. This is due to the stochastic nature of DNN training, with even the same batch size yielding different energy and time consumptions. Hence, Grid Search may choose a suboptimal configuration when a suboptimal configuration luckily yields good energy and time consumptions.

6.3 Trace-Driven Simulation Using the Alibaba Trace

Here we evaluate how Zeus can save energy and time consumption for DNN training in large clusters. We run trace-



(a) *Energy Consumption*



(b) *Training Time*

Figure 9: ***Zeus reduces energy consumption for all jobs in the Alibaba cluster trace [94], compared to Grid Search and Default. (a) Energy consumption with Zeus comparing against baselines, (b) Training time of each type of workload. Both are normalized by the Default baseline.***

driven simulation using the Alibaba GPU cluster trace [94] which contains over 1.2 million jobs spanning a period of two months. The Alibaba GPU cluster trace is suitable for our evaluation for two reasons. First, the trace identifies groups of recurring jobs, and each job is annotated with its group ID. Second, jobs within the same group show overlap in their execution, allowing us to evaluate Zeus's capability of handling concurrent job submissions with Thompson Sampling.

In order to assign job groups to the workload (Table 1) that best resembles its runtime, we remove jobs that did not successfully terminate and run K-Means clustering [36] on the mean job runtime of each group to form six clusters. Then, we match the six clusters with our six workloads in the order of their mean runtime. When running simulation, in order to capture the intra-cluster runtime variation of each job, we scale the job runtime with the ratio of the job's original runtime to its cluster's mean runtime. We compare Zeus with Default and Grid Search and plot the results in Figure 9.

Figure 9a shows the cumulative energy consumption of training using all three approaches. Zeus outperforms both baselines for workloads of all types and sizes. Note that there are scenarios where the Grid Search performs worse than Default, due to it wasting too much energy and time during the exploration stage. Thanks to Zeus's *early stopping* and quick online power optimization, its energy and time cost during the exploration stage is significantly reduced. Across all the models, Zeus reduces training energy usage by 7%–52%. Figure 9b shows the training time using Zeus to be increased by at most 16%, and in many cases even decreased by up to 33%. Finally, similar to earlier experiments, Zeus had significantly lower cumulative regret than Grid Search.

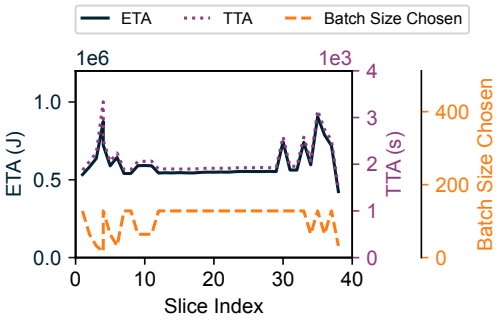


Figure 10: ***Energy and time consumption of training BERT with Zeus on Capriccio and the batch size chosen for each slice.***

6.4 Handling Data Drift

While there are previous works that attempt to identify and address data drift in general ML settings [63], existing datasets are classification tasks based on small feature vectors [12, 35], completely synthetic [25, 44], or private [66].

Therefore, we create and open-source a new sentiment analysis dataset called *Capriccio* that is suitable for evaluating DNN models. Capriccio consists of 1.6 million tweets over three months from the Sentiment140 [26] dataset, labeled with sentiment scores and the timestamp of the tweet. We emulate data drift by capturing a sliding window of 500,000 tweets (roughly the amount of tweets in one month) at a time and moving the window forward by each day, generating 38 slices. We skip empty dates to avoid having identical slices.

We train BERT [21] on Capriccio with Zeus configured with a window size of 10, roughly corresponding to a time frame of two weeks on Twitter. We plot the selected batch size for each recurrence (slice) and its corresponding ETA and TTA of training in Figure 10. It can be seen that spikes in ETA and TTA (signaling that the current batch size may no longer be optimal) trigger the exploration of a batch size that is different from the one previously converged to.

6.5 Overhead of JIT Profiling

Measurements with the Deepspeech2 model using the default batch size b_0 show that JIT profiling results in a 0.01% increase in energy consumption and a 0.03% increase in time consumption. Such a tiny overhead is possible because the time needed to profile all power limits is very small (less than one minute) while one epoch of training spans hours (which is typical for DL workloads). Measurements on ShuffleNet-v2, which has much shorter epoch duration, show that JIT profiling results in a 0.6% increase in terms of time consumption and a 2.8% reduction in energy consumption.

6.6 Scaling to Multi-GPU

While the primary focus of this paper is on single-GPU settings, in this section, we show that Zeus can be extended to single-node multi-GPU training settings by profiling the power consumption of all GPUs that participate in training.

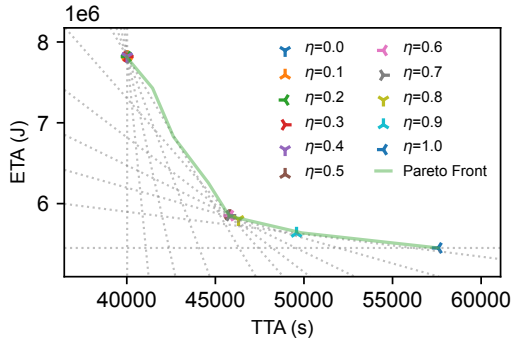


Figure 11: ***Pareto Front of DeepSpeech2 and how*** η ***navigates it.***

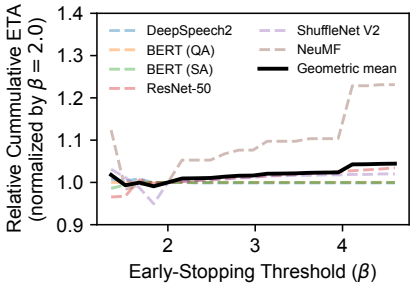


Figure 12: ***Relative cumulative energy consumption of Zeus across all jobs, w.r.t. the early-stopping threshold*** β.

Extensions to distributed multi-GPU setups that involve network communication is a potential future work.

Extending to multi-GPU allows us to compare our energy and time consumption with Pollux [77], a state-of-the-art distributed cluster scheduler that dynamically tunes the batch size *during* DNN training in order to maximize *goodput*. Training DeepSpeech2 on LibriSpeech on four NVIDIA A40 GPUs, Zeus consumes 12% more time but 21% less energy, comparing favorably. We especially note that while Pollux does not take energy into account, Zeus allows the user to select a different energy-time tradeoff point (e.g., speed up training but consume more energy) by selecting an appropriate η.

6.7 Sensitivity Analysis and Ablation Studies

Impact of η**.** To characterize the impact of η as defined in Equation 2, we perform a sweep of $0 \leq \eta \leq 1$ when training DeepSpeech2 and plot the resulting optimal (TTA, ETA) in Figure 11. We also plot the corresponding Pareto Front for reference. We observe that the resulting (TTA, ETA) data points fall closely to the Pareto Front. Moreover, we plot the lines along which the C in Equation 2 is a constant, shown as the dotted lines. As expected, these lines form an envelope around the Pareto Front. Additional sensitivity analysis for η can be found in Appendix F.

Impact of early-stopping threshold β**.** To study impact of the early-stopping threshold β, we sweep β from 1.5 to 5 and measure the cumulative ETA across all jobs. We calculate the difference in ETA relative to our default choice of $\beta = 2.0$, and plot the result of all jobs as well as a geometric mean across all jobs in Figure 12. The result shows that the default

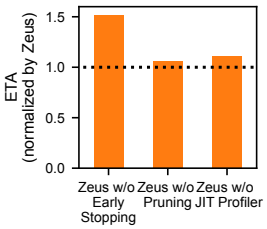


Figure 13: ***Performance breakdown of Zeus.***

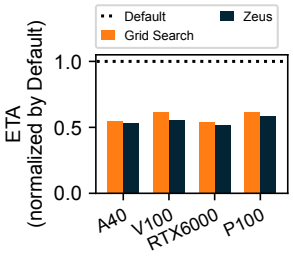


Figure 14: ***Normalized ETA w.r.t. GPU models.***

$\beta = 2.0$ chosen by Zeus achieves the lowest geometric mean across all jobs. The intuition behind this is that when β is too low, Zeus prematurely stops exploratory runs, reducing the effectiveness of exploration. In contrast, when β is too high, it dilutes the benefit of early stopping which leads to inflated exploration cost.

Impact of individual components. In order to show the gains from each component, we show the degradation of removing one component from Zeus: no early stopping (setting β to infinity), no pruning (keeping a batch size that didn't reach the target accuracy), and no JIT profiling (profiling each power limit in different recurrences). Figure 13 shows the slowdown relative to Zeus after disabling these components. We observe that the Zeus benefits mostly from early stopping.

Impact of GPU models. Figure 14 shows the geometric mean of ETA normalized against Default across all jobs. Zeus achieves consistent ETA reductions across four generations of NVIDIA GPUs. See Appendix G for all results.

7 Discussion

Choice of configuration knobs. In this paper, we pick the batch size and GPU power limit as the configuration knobs for Zeus to optimize. We choose these two to strike a balance in the tradeoff between the granularity of control and the size of the search space. For instance, one can set the frequency and voltage for individual components on the GPU for more fine-grained control and potentially higher energy efficiency, but this would result in prolonged exploration in the bigger search space. In contrast, we choose the GPU power limit, which effectively controls both frequency and voltage via DVFS and reduces the search space.

On the DL job configuration side, we pick the batch size as the knob for a similar reason. Changing the batch size has a broader impact on energy consumption of end-to-end DNN training, because it affects both the training time and the average power consumption during training. In comparison, other candidate configuration knobs such as learning rate fall short because they only affect the training time.

Hyperparameter optimization. Hyperparameter optimization is an important workload, where many DL training jobs (trials) are submitted with specific hyperparameters chosen from a user-defined search space [9, 59, 60, 98]. If the users

submit these trials with a specific batch size, they can specify the feasible batch size set $\mathcal{B}$ to only contain that single batch size. In this case, Zeus can still reduce energy consumption by searching for the optimal GPU power limit.

Supporting distributed training. Zeus currently only supports single-node training, but it can easily be extended to support distributed scenarios. Since the same type of GPU will have the same time and power consumption characteristics, we can apply the same power limit configuration across all GPUs to avoid stragglers. The definition of cost can be extended to sum over the time and energy consumption of all GPUs participating in training, and all other components in our solution can remain identical.

Supporting heterogeneous GPUs. Our solution assumes that the training job runs on the same type of GPU across all of its recurrences. However, in practice, this may not always be possible due to varying resource contention or availability.

It is straightforward to add support for heterogeneous GPUs under our formulation. That is, cost values observed from one GPU can be *translated* to values that represent the characteristics of another GPU. As shown in Equation 6, energy-time cost can be written as the product of $\texttt{Epochs}(b)$ and $\texttt{EpochCost}(b;\eta)$. Here, the former term is independent with the choice of the GPU. Moreover, the latter term can be quickly profiled on any GPU because it consists of only $\texttt{AvgPower}(b,p)$ and $\texttt{Throughput}(b,p)$. Thus, we can obtain cost values that represent the new GPU by quickly profiling $\texttt{EpochCost}(b;\eta)$ for each batch size on the new GPU and multiplying it with $\texttt{Epochs}(b)$ observed from the previous GPU. These translated cost observations can then be used to learn a new MAB that specializes on the new GPU.

8 Related Work

DNN training. A large body of recent studies focus on creating fast kernels for tensor operations [18, 45, 92, 100], efficiently placing data and/or computation [55, 72, 78, 97], and optimizing communication [76, 91]. However, most of them optimize for TTA and are oblivious of their energy impact. These works can be applied together with Zeus, potentially accelerating training while making it energy efficient.

Another recent effort in reducing TTA (without considering energy) in multi-GPU DNN training settings is Pollux [77]. Pollux dynamically changes the batch size *during* training based on the Gradient Noise Scale (GNS) [68]. However, GNS does not theoretically capture the generalization of the model [68] and can only be efficiently approximated when there are more than one GPUs participating in training. Zeus, on the other hand, optimizes and trades off TTA and ETA by tuning the batch size *across* job recurrences and does not alter the model's convergence characteristics.

Energy measurement for Deep Learning. A recent line of research has analyzed the energy consumption [75] as well as the environmental impact [54, 85] for training large DNN models inside a cluster. On the device-level, benchmarking efforts have been made to understand the energy efficiency and performance of training DNN on GPUs and other accelerators [93]. Several Python frameworks have been built for measurement [14, 40] and prediction [5] of energy consumption for DNN training. Zeus takes a similar software-based approach to measure power consumption via NVML [2], in order to perform JIT profiling of DNN training jobs.

Energy optimization for Deep Learning. Existing work has investigated energy-accuracy tradeoff in the context of DNN inference with new neural network architecture [89] and algorithm-hardware co-design [86], and training strategies such as warm-start [7] and gradient-matching-based data subset selection [50]. Other works optimize energy for DNN training on multiple GPUs with scheduling [47] and task mapping [52]. Zeus complements these solutions as it can be plugged in transparently into these frameworks.

Several works have studied the impact of GPU dynamic frequency and voltage scaling (DVFS) and power configuration on the energy consumption and performance of DNN training [11, 52, 87, 90, 101], wherein they focus on the tradeoff between the transient metric of system throughput and power consumption. While these work rely on offline modeling and profiling, Zeus focuses on a more realistic end-to-end metric of energy-to-accuracy and is fully online.

BatchSizer [71] introduces batch size as a control knob to optimize for energy efficiency of DNN inference. Zeus focuses on DNN training, and takes a holistic approach, optimizing both GPU and job configurations together.

9 Conclusion

In this work, we sought to understand and optimize the energy consumption of DNN training on GPUs. We identified the tradeoff between energy consumption and training time, and demonstrated that common practices can lead to inefficient energy usage. Zeus is an online optimization framework for recurring DNN training jobs that *finds* the Pareto frontier and allows users to *navigate* the frontier by automatically tuning the batch size and GPU power limit of their jobs. Zeus outperforms the state-of-the-art in terms of energy usage for diverse workloads and real cluster traces by continuously adapting to dynamic workload changes such as data drift. We earnestly hope that Zeus will inspire the community to consider energy as a first-class resource in DNN optimization.

Acknowledgements

Special thanks to CloudLab and Chameleon Cloud for making Zeus experiments possible. We would also like to thank the reviewers, our shepherd Jayashree Mohan, and SymbioticLab members for their insightful feedback. We also thank our colleague Rui Liu for his helpful suggestions. This work is in part supported by NSF grants CNS-1909067 and CNS-2104243 and a grant from VMWare. Jae-Won Chung is additionally supported by the Kwanjeong Educational Foundation.

References

[1] How much electricity does an American home use? https://www.eia.gov/tools/faqs/faq.php?id=97&t=3.

[2] NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[3] NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.

[4] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. Treehouse: A case for carbon-aware datacenter software. In *HotCarbon*, 2022.

[5] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems, 2020.

[6] Yehia Arafa, Ammar ElWazir, Abdelrahman ElKanishy, Youssef Aly, Ayatelrahman Elsayed, Abdel-Hameed Badawy, Gopinath Chennupati, Stephan Eidenbenz, and Nandakishore Santhi. Verified instruction-level energy consumption measurement for NVIDIA GPUs. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, 2020.

[7] Jordan Ash and Ryan P Adams. On warm-starting neural network training. *NeurIPS*, 2020.

[8] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[9] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *NeurIPS*, 2011.

[10] Omar Besbes, Yonatan Gur, and Assaf Zeevi. Stochastic multi-armed-bandit problem with non-stationary rewards. *NeurIPS*, 2014.

[11] Srikant Bharadwaj, Shomit Das, Yasuko Eckert, Mark Oskin, and Tushar Krishna. Dub: Dynamic underclocking and bypassing in NoCs for heterogeneous GPU workloads. In *2021 15th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2021.

[12] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Philipp Kranen, Hardy Kremer, Timm Jansen, and Thomas Seidl. Moa: Massive online analysis, a framework for stream classification and clustering. In *Proceedings of the first workshop on applications of pattern analysis*, 2010.

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

[14] Qingqing Cao, Aruna Balasubramanian, and Niranjan Balasubramanian. Towards accurate and reliable energy measurement of NLP models. In *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*, 2020.

[15] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access*, 8:225134–225180, 2020.

[16] Yair Censor. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization*, 4(1):41–59, 1977.

[17] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. *NeurIPS*, 2011.

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[19] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *ACM SIGOPS Operating Systems Review*, 2019.

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[22] Jesse Dodge, Taylor Prewitt, Remi Tachet des Combes, Erika Odmark, Roy Schwartz, Emma Strubell, Alexandra Sasha Luccioni, Noah A. Smith, Nicole DeCario, and Will Buchanan. Measuring the carbon intensity of AI in cloud instances. In *ACM Conference on Fairness, Accountability, and Transparency*, 2022.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *ATC*, 2019.

[24] Daniel Fink. A compendium of conjugate priors. 1997.

[25] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In *Brazilian symposium on artificial intelligence*, 2004.

[26] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *Stanford CS224N project report*, 2009.

[27] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[29] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[30] Diego Granziol, Stefan Zohren, and Stephen Roberts. Learning rates as a function of batch size: A random matrix theory approach to neural network training. *Journal of Machine Learning Research*, 23(173):1–65, 2022.

[31] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, 2019.

[32] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's DNN-based personalized recommendation. In *HPCA*, 2020.

[33] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[34] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *ACM transactions on interactive intelligent systems (TIIS)*, 5(4):1–19, 2015.

[35] Michael Harries and New South Wales. Splice-2 comparative evaluation: Electricity pricing. 1999.

[36] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[37] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *HPCA*, 2018.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[39] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, 2017.

[40] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning. *Journal of Machine Learning Research*, 21(248):1–43, 2020.

[41] Miro Hodak, Masha Gorkovenko, and Ajay Dholakia. Towards power efficiency in deep learning on data center hardware. In *IEEE International Conference on Big Data*, 2019.

[42] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *NeurIPS*, 2017.

[43] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *ISCA*, 2010.

[44] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM international conference on Knowledge discovery and data mining (SIGKDD)*, 2001.

[45] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[46] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. AccelWattch: A power modeling framework for modern GPUs. In *MICRO*, 2021.

[47] Dong-Ki Kang, Ki-Beom Lee, and Young-Chon Kim. Cost efficient GPU cluster management for training and inference of deep learning. *Energies*, 15(2):474, 2022.

[48] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. Lessons learned from the chameleon testbed. In *ATC*, 2020.

[49] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

[50] Krishnateja Killamsetty, S Durga, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. Grad-match: Gradient matching based data subset selection for efficient deep model training. In *ICML*, 2021.

[51] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[52] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa, and Hiroshi Nakamura. Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping. In *2013 IEEE 31st International Conference on computer design (ICCD)*. IEEE, 2013.

[53] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[54] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[55] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, 2021.

[56] Tze Leung Lai, Herbert Robbins, et al. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[57] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *ICLR*, 2020.

[58] Sebastien Levy, Randolph Yao, Youjiang Wu, Yingnong Dang, Peng Huang, Zheng Mu, Pu Zhao, Tarun Ramani, Naga Govindaraju, Xukun Li, et al. Predictive and adaptive failure mitigation to avert production cloud VM interruptions. In *OSDI*, 2020.

[59] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-Tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. *Proceedings of Machine Learning and Systems*, 2:230–246, 2020.

[60] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[61] Weixin Liang and James Zou. Metashift: A dataset of datasets for evaluating contextual distribution shifts and training conflicts. *arXiv preprint arXiv:2202.06523*, 2022.

[62] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

[63] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018.

[64] Cheng Luo and Reiji Suda. A performance and energy consumption analytical model for GPU. In *2011 IEEE ninth international conference on dependable, autonomic and secure computing*, 2011.

[65] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient CNN architecture design. In *ECCV*, 2018.

[66] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. In *MLSys*, 2022.

[67] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *SIGMOD*, 2021.

[68] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[69] Xinxin Mei, Qiang Wang, and Xiaowen Chu. A survey and measurement study of GPU DVFS on energy conservation. *Digital Communications and Networks*, 3(2):89–100, 2017.

[70] Sparsh Mittal and Sumanth Umesh. A survey on hardware accelerators and optimization techniques for RNNs. *Journal of Systems Architecture*, 112:101839, 2021.

[71] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. Batchsizer: Power-performance tradeoff for DNN inference. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021.

[72] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *SOSP*, 2019.

[73] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an ASR corpus based on public domain audio books. In *IEEE international conference on acoustics, speech and signal processing (ICASSP)*, 2015.

[74] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS*, 2019.

[75] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[76] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *SOSP*, 2019.

[77] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI*, 2021.

[78] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[79] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *EMNLP*, 2016.

[80] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[81] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning*, 11(1):1–96, 2018.

[82] Simone Scardapane and Dianhui Wang. Randomness in neural networks: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(2):e1200, 2017.

[83] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Commun. ACM*, 63(12):54–63, 2020.

[84] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. In *ICLR*, 2018.

[85] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[86] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M Rush, David Brooks, et al. EdgeBERT: Sentence-level energy optimizations for latency-aware multi-task NLP inference. In *MICRO*, 2021.

[87] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. The impact of GPU DVFS on the energy and performance of deep learning: An empirical study. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, 2019.

[88] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 1933.

[89] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. ALERT: Accurate learning for energy and timeliness. In *ATC*, 2020.

[90] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. Dynamic GPU energy optimization for machine learning training workloads. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[91] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ML. In *Proceedings of Machine Learning and Systems*, 2020.

[92] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.

[93] Yuxin Wang, Qiang Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Kaiyong Zhao, and Xiaowen Chu. Benchmarking the performance and energy efficiency of AI accelerators for AI training. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020.

[94] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *NSDI*, 2022.

[95] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *EMNLP*, 2020.

[96] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga, Jinshi Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. Sustainable AI: Environmental implications, challenges and opportunities. In *Proceedings of Machine Learning and Systems*, 2022.

[97] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. GSPMD: general and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

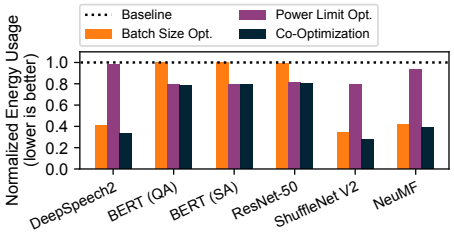
[98] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. Fluid: Resource-aware hyperparameter tuning engine. *MLSys*, 2021.

[99] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

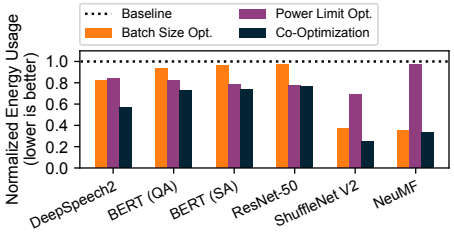
[100] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, 2020.

[101] Pengfei Zou, Ang Li, Kevin Barker, and Rong Ge. Indicator-directed dynamic power management for iterative workloads on GPU-accelerated systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020.

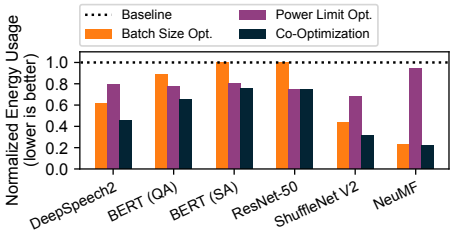
A Energy Savings Potential on GPUs



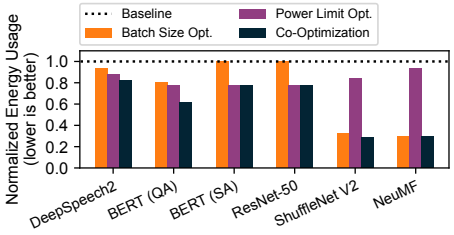
(a) *NVIDIA A40.*



(b) *NVIDIA V100.*



(c) *NVIDIA RTX6000.*



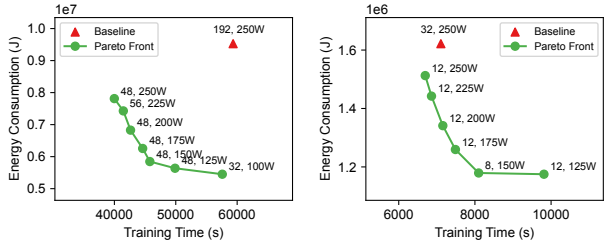
(d) *NVIDIA P100.*

Figure 15: ***Energy usage normalized against Baseline for DNN training, measured on (a) NVIDIA A40 GPU, (b) NVIDIA V100 GPU, (c) NVIDIA RTX6000 GPU and (d) NVIDIA P100 GPU.***

Figure 15 shows the potential for energy savings on four different generations of NVIDIA GPUs: Ampere (A40), Volta (V100), Turing (RTX6000), and Pascal (P100). All four generations show that there are sufficient potential for energy savings, motivating Zeus.

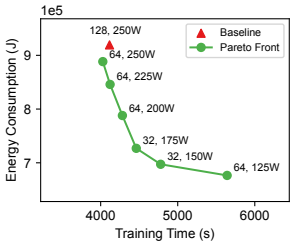
B TTA vs. ETA for All Workloads

Figure 16 plots the Pareto Front for all six workloads and the baseline (default batch size and maximum power limit) is shown as a red triangle. Note that the axes do not start from zero in order to zoom into the Pareto Front. Data points were gathered on an NVIDIA V100 GPU.

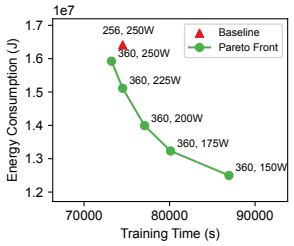


(a) *DeepSpeech2*

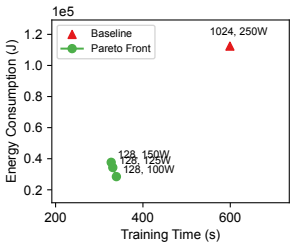
(b) *BERT (QA)*



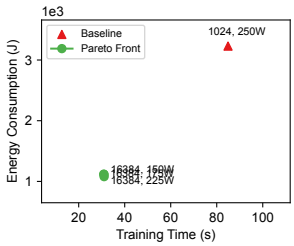
(c) *BERT (SA)*



(d) *ResNet-50*



(e) *ShuffleNet V2*



(f) *NeuMF*

Figure 16: ***ETA vs. TTA across all workloads, with Pareto Front and default configuration highlighted. Measured on an NVIDIA V100 GPU.***

C ETA w.r.t. Configurations for All Workloads

Figures 17 and 18 respectively show the ETA value when batch size and power limit are swept. Especially note the convexity of all BS-ETA curves, which justifies the design of our pruning exploration algorithm.

D Cumulative Regret of All Workloads

Figure 19 shows the cumulative regret of Zeus and Grid Search over job recurrences for all six workloads. In general, Zeus converges to a better knob than Grid Search while being faster.

E Search Paths for All Workloads

Figures 20 and 21 respectively show the search path of Zeus and Grid Search in the 2D configuration space. Thanks to the decoupling of batch size and power limit, Zeus is able to more efficiently navigate the search space and converge to a knob while consuming less energy and time during exploration.

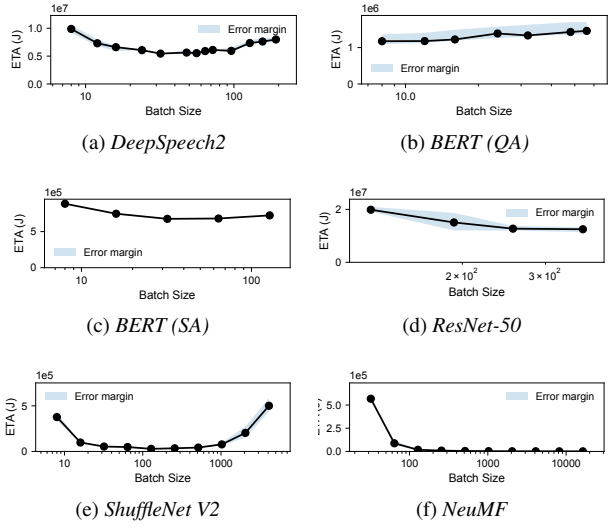


Figure 17: ***ETA w.r.t batch size of different DNN training workload. The blue shade shows the error margin across repeated runs.***

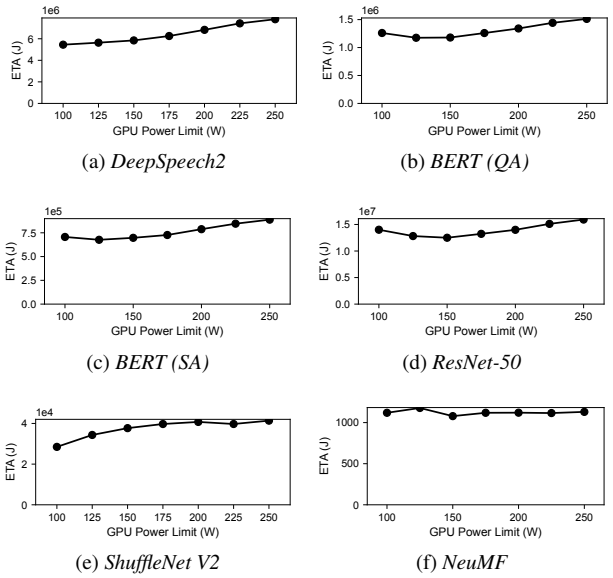


Figure 18: ***ETA w.r.t GPU power limit of different DNN training workload. Measured on an NVIDIA V100 GPU.***

F Additional Sensitivity Analysis

Figure 22 compares both the energy consumption and training time for Zeus against Default. We also calculate and plot the geometric mean across all jobs. The result shows that with higher η, Zeus prioritizes reducing energy consumption over time, leading to higher improvement factor of energy, and vice versa.

G Performance of Zeus on All GPUs

Figure 23 presents the energy and time consumption of all workloads on four different generations NVIDIA GPUs: Ampere (A40), Volta (V100), Turing (RTX6000), and Pascal (P100). The overall trends hold for all GPUs.

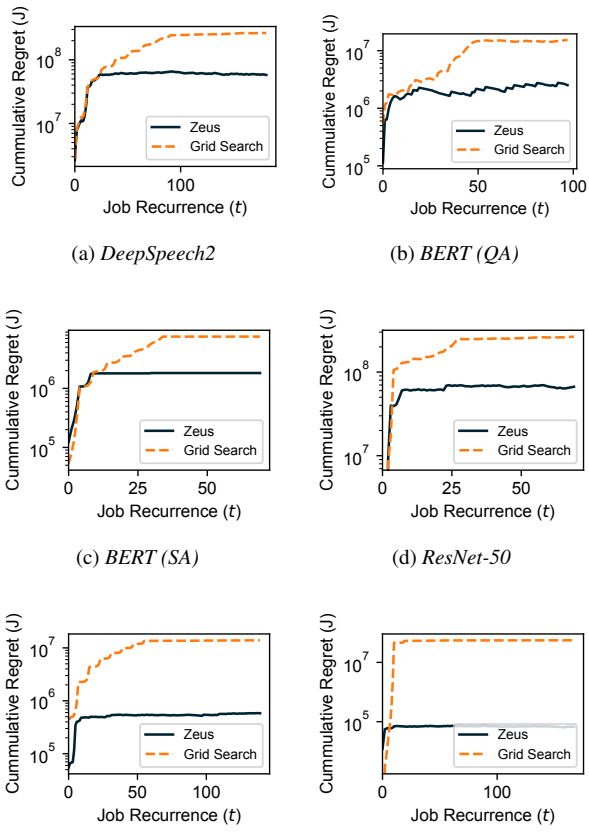
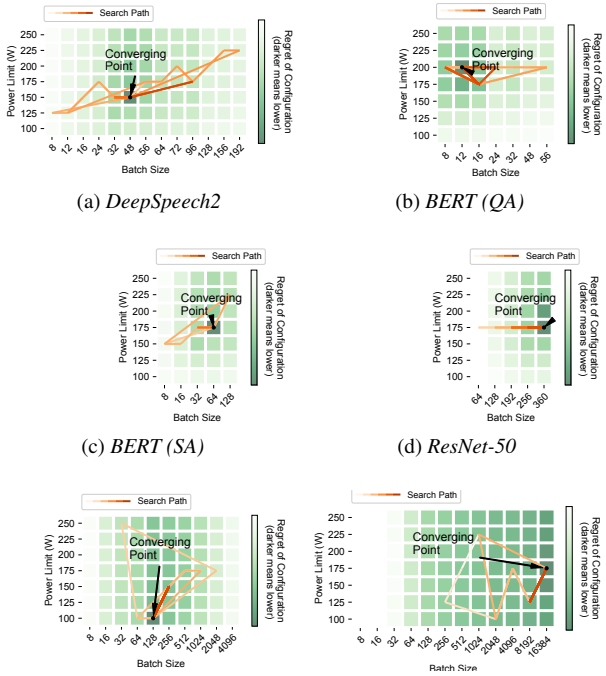


Figure 19: ***Cumulative regret of Zeus vs. Grid Search across all workloads.***

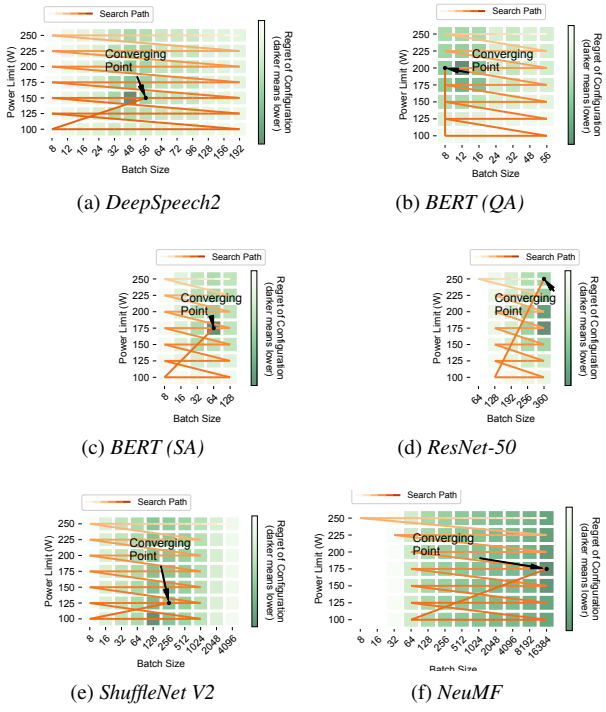


(a) *DeepSpeech2* (b) *BERT (QA)*

(c) *BERT (SA)* (d) *ResNet-50*

(e) *ShuffleNet V2* (f) *NeuMF*

Figure 20: ***Search path of Zeus across all workloads.***

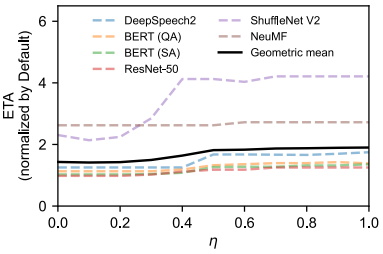


(a) *DeepSpeech2* (b) *BERT (QA)*

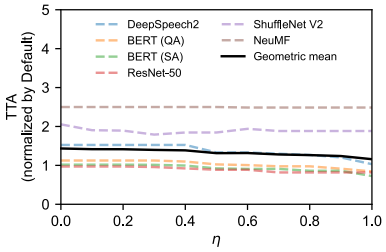
(c) *BERT (SA)* (d) *ResNet-50*

(e) *ShuffleNet V2* (f) *NeuMF*

Figure 21: ***Search path of Grid Search across all workloads.***

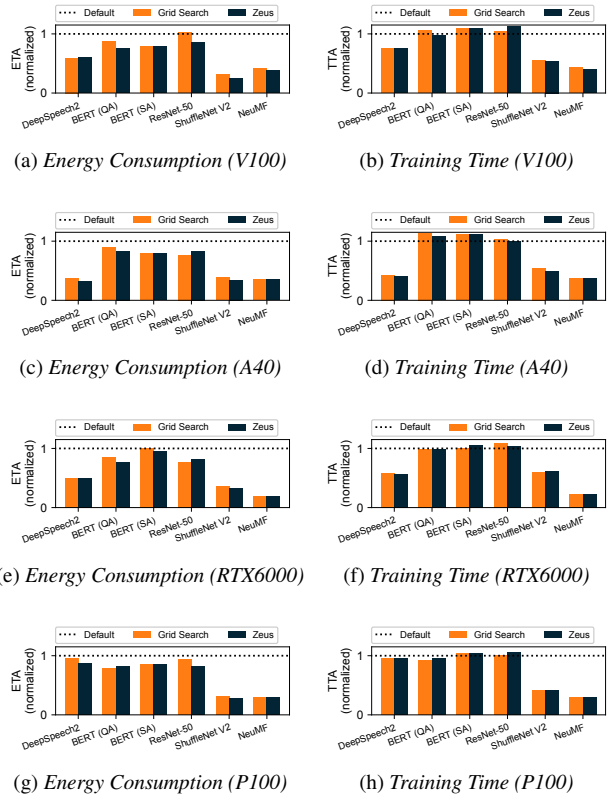


(a) *ETA*



(b) *TTA*

Figure 22: ***Impact of priority knob η on ETA and TTA.***



(a) *Energy Consumption (V100)* (b) *Training Time (V100)*

(c) *Energy Consumption (A40)* (d) *Training Time (A40)*

(e) *Energy Consumption (RTX6000)* (f) *Training Time (RTX6000)*

(g) *Energy Consumption (P100)* (h) *Training Time (P100)*

Figure 23: ***Energy and time consumption of DNN training, normalized against Default for DNN training. Results measured on (a) NVIDIA A40 GPU, (b) NVIDIA V100 GPU, (c) NVIDIA RTX6000 GPU and (d) NVIDIA P100 GPU.***

Remote Procedure Call as a Managed System Service

Jingrong Chen[1,*] Yongji Wu[1,*] Shihan Lin[1] Yechen Xu[3] Xinhao Kong[1]
Thomas Anderson[2] Matthew Lentz[1] Xiaowei Yang[1] Danyang Zhuo[1]

[1]*Duke University* [2]*University of Washington* [3]*Shanghai Jiao Tong University*

Abstract

Remote Procedure Call (RPC) is a widely used abstraction for cloud computing. The programmer specifies type information for each remote procedure, and a compiler generates stub code linked into each application to marshal and unmarshal arguments into message buffers. Increasingly, however, application and service operations teams need a high degree of visibility and control over the flow of RPCs between services, leading many installations to use sidecars or service mesh proxies for manageability and policy flexibility. These sidecars typically involve inspection and modification of RPC data that the stub compiler had just carefully assembled, adding needless overhead. Further, upgrading diverse application RPC stubs to use advanced hardware capabilities such as RDMA or DPDK is a long and involved process, and often incompatible with sidecar policy control.

In this paper, we propose, implement, and evaluate a novel approach, where RPC marshalling and policy enforcement are done as a system service rather than as a library linked into each application. Applications specify type information to the RPC system as before, while the RPC service executes policy engines and arbitrates resource use, and then marshals data customized to the underlying network hardware capabilities. Our system, mRPC, also supports live upgrades so that both policy and marshalling code can be updated transparently to application code. Compared with using a sidecar, mRPC speeds up a standard microservice benchmark, DeathStarBench, by up to 2.5× while having a higher level of policy flexibility and availability.

1 Introduction

Remote Procedure Call (RPC) is a fundamental building block of distributed systems in modern datacenters. RPC allows developers to build networked applications using a simple and familiar programming model [10], supported by several popular libraries such as gRPC [26], Thrift [84], and eRPC [39]. The RPC model has been widely adopted in distributed data stores [19, 41, 83], network file systems [24, 80], consensus protocols [68], data-analytic frameworks [2, 12, 16, 25, 55, 82, 94, 98], cluster schedulers and orchestrators [30, 50], and machine learning systems [1, 65, 72]. Google found that roughly 10% of its datacenter CPU cycles are spent just executing gRPC library code [42]. Because of its importance, improving RPC performance has long been a major topic of research [7, 8, 10, 14, 39, 52, 63, 81, 87, 95, 96].

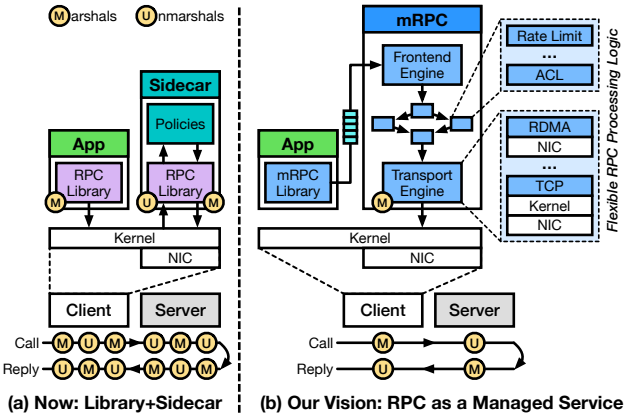


Figure 1: Architectural comparison between current (RPC library + sidecar) and our proposed (RPC as a managed service) approaches.

Recently, application and network operations teams have found a need for rapid and flexible visibility and control over the flow of RPCs in datacenters. This includes monitoring and control of the performance of specific types of RPCs [62], prioritization and rate limiting to meet application-specific performance and availability goals, dynamic insertion of advanced diagnostics to track user requests across a network of microservices [22], and application-specific load balancing to improve cache effectiveness [6], to name a few.

The typical architecture is to enforce policies in a sidecar—a separate process that mediates the network traffic of the application RPC library (Figure 1a). This is often referred to as a service mesh. A number of commercial products have been developed to meet the need for sidecar RPC proxies, such as

*Jingrong Chen and Yongji Wu contributed equally.

Envoy [18], Istio [32], HAProxy [29], Linkerd [53], Nginx [67], and Consul [15]. Although some policies could theoretically be supported by a feature-rich RPC runtime linked in with each application, that can slow deployment—Facebook recently reported that it can take *months* to fully roll out changes to one of its application communication libraries [21]. One use case that requires rapid deployment is to respond to a new application security threat, or to diagnose and fix a critical user-visible failure. Finally, many policies are mandatory rather than discretionary—the network operations team may not be able to trust the library code linked into an application. Example mandatory security policies include access control, authentication/encryption [15], and prevention of known exploits in widely used network protocols such as RDMA [79].

Although using a sidecar for policy management is functional and secure, it is also inefficient. The application RPC library marshals RPC parameters at runtime into a buffer according to the type information provided by the programmer. This buffer is sent through the operating system network stack and then forwarded back up to the sidecar, which typically needs to parse and unwrap the network, virtualization, and RPC headers, often looking inside the packet payload to correctly enforce the desired policy. It then re-marshals the data for transport. Direct application-level access to network hardware such as RDMA or DPDK offers high performance but precludes sidecar policy control. Similarly, network interface cards are increasingly sophisticated, but it is hard for applications or sidecars to take advantage of those new features, because marshalling is done too high up in the network stack. Any change to the marshalling code requires recompiling and rebooting each application and/or the sidecar, hurting end-to-end availability. In short, existing solutions can provide good performance, or flexible and enforceable policy control, but not both.

In this paper, we propose a new approach, called RPC as a managed service, to address these limitations. Instead of separating marshalling and policy enforcement across different domains, we combine them into a single privilege and trusted system service (Figure 1b) so that marshalling is done **after** policy processing. In our prototype, mRPC for managed RPC, the privileged RPC service runs at user level communicating with the application through shared memory regions [4,8,58]. However, mRPC could also be integrated directly into the operating system kernel with a dynamically replaceable kernel module [61].

Our goals are to be fast, support flexible policies, and provide high availability for applications. To achieve this, we need to address several challenges. First, we need to decouple marshalling from the application RPC library. Second, we need to design a new policy enforcement mechanism to process RPCs efficiently and securely, without incurring additional marshalling overheads. Third, we need to provide a way for operators to specify/change policies and even change the underlying transport implementation without disrupting running applications.

We implement mRPC, the first RPC framework that follows the RPC as a managed service approach. Our results show that mRPC speeds up DeathStarBench [23] by up to 2.5×, in terms of mean latency, compared with combining state-of-art RPC libraries and sidecars, i.e., gRPC and Envoy, using the same transport mechanism. Larger performance gains are possible by fully exploiting network hardware capabilities from within the service. In addition, mRPC allows for live upgrades of its components while incurring negligible downtime for applications. Applications do not need to be re-compiled or rebooted to change policies or marshalling code. mRPC has three important limitations. First, data structures passed as RPC arguments must be allocated on a special shared-memory heap. Second, while we use a language-independent protocol for specifying RPC type signatures, our prototype implementation currently only works with applications written in Rust. Finally, our stub generator is not as fully featured as gRPC.

In this paper, we make the following contributions:

- A novel RPC architecture that decouples marshalling/unmarshalling from RPC libraries to a centralized system service.
- An RPC mechanism that applies network policies and observability features with both security and low performance overhead, i.e., with minimal data movement and no redundant (un)marshalling. The mechanism supports live upgrade of RPC bindings, policies, transport, and marshalling without disrupting running applications.
- A prototype implementation of mRPC along with an evaluation on both synthetic workloads and real applications.

2 Background

In this section, we discuss the current RPC library architecture. We then discuss the emerging need for manageability and how manageability is implemented with existing RPC libraries.

2.1 Remote Procedure Call

To use RPC, a developer defines the relevant service interfaces and message types in a schema file (e.g., gRPC `.proto` file). A protocol compiler will translate the schema into program stubs that are directly linked with the client and server applications. To issue an RPC at runtime, the application simply calls the corresponding function provided by the stub; the stub is responsible for marshalling the request arguments and interacting with the transport layer (e.g., TCP/IP sockets or RDMA verbs). The transport layer delivers the packets to the remote server, where the stub unmarshals the arguments and dispatches the RPC request to a thread (eventually replying back to the client). We refer to this approach as *RPC-as-a-library*, since all RPC functionality is included in user-space libraries that are linked with each application. Even though the first RPC implementation [10] dates back to the 1980s, modern RPC frameworks (e.g., gRPC [26], eRPC [39], Thrift [84]) still follow this same approach.

A key design goal for RPC frameworks is efficiency. Google and Facebook have built their own efficient RPC frameworks, gRPC and Apache Thrift. Although primarily focused on portability and interoperability, gRPC includes many efficiency-related features, such as supporting binary payloads. Academic researchers have studied various ways to improve RPC efficiency, including optimizing the network stack [45,69,99], software hardware co-design [39, 41], and overload control [14].

As network link speeds continue to scale up [77], RPC overheads are likely to become even more salient in the future. This has led some researchers to advocate for direct application access to network hardware [5, 39, 73, 99], e.g., with RDMA or DPDK. Although low overhead, kernel bypass is largely incompatible with the need for flexible and enforceable layer 7 policy control, as we discuss next. In practice, multiple security weaknesses in RDMA hardware have led most cloud vendors to opt against providing direct access to RDMA by untrusted applications [48, 49, 58, 79, 95, 101].

2.2 The Need for Manageability

As RPC-based distributed applications scale to large, complex deployment scenarios, there is an increasing need for improved manageability of RPC traffic. We classify management needs into three categories: **1) Observability:** Provide detailed telemetry, which enables developers to diagnose and optimize application performance. **2) Policy Enforcement:** Allow operators to apply custom policies to RPC applications and services (e.g., access control, rate limits, encryption). **3) Upgradability:** Support software upgrades (e.g., bug fixes and new features) while minimizing downtime to applications.

One natural question to ask is: *is it possible to add these properties without changing existing RPC libraries?* For observability and policy enforcement, the state-of-the-art solution is to use a sidecar (e.g., Envoy [18] or Linkerd [53]). A sidecar is a standalone process that intercepts every packet an application sends, reconstructing the application-level data (i.e., RPC), and applying policies or enabling observability. However, using a sidecar introduces substantial performance overhead, due to redundant RPC (un)marshalling. This RPC (un)marshalling, for example, in gRPC+Envoy, including HTTP framing and protobuf encoding, accounts for 62-73% overhead in the end-to-end latency [102]. In our evaluation (§7), using a sidecar increases the 99th percentile RPC latency by 180% and decreases the bandwidth by 44%. Figure 1a shows the (un)marshalling steps invoked as an RPC traverses from a client to a server and back. Using a sidecar triples the number of (un)marshalling steps (from 4 to 12). In addition, the sidecar approach is largely incompatible with the emerging trend of efficient application-level access to network hardware. Using sidecars means data buffers have to be copied between the application and sidecars, reducing the benefits of having zero-copy kernel-bypass access to the network.

Finally, using sidecars with application RPC libraries does not completely solve the upgradability issue. While policy can often be changed dynamically (depending on the feature set of the sidecar implementation), marshalling and transport code is harder to change. To fix a bug in the underlying RPC library, or merely to upgrade the code to take advantage of new hardware features, we need to recompile the entire application (and sidecar) with the patched RPC library and reboot. gRPC has a monthly or two-month release cycle for bug fixes and new features [27]. Any scheduled downtime has to be communicated explicitly to the users of the application or has to be masked using replication; either approach can lead to complex application life-cycle management issues.

We do not see much hope in continuing to optimize this RPC library and sidecar approach for two reasons. First, a strong coupling exists between a traditional RPC library and each application. This makes upgrading the RPC library without stopping the application difficult, if not impossible. Second, there is only weak or no coupling between an RPC library and a sidecar. This prevents the RPC library and the sidecar from cross-layer optimization.

Instead, we argue for an alternative architecture in which RPC is provided *as a managed service*. By decoupling RPC logic, e.g., (un)marshalling, transport interface, from the application, the service can simultaneously provide high performance, policy flexibility, and zero-downtime upgrades.

3 Overview

Our system, mRPC, realizes the *RPC-as-a-managed-service* abstraction while maintaining similar end-to-end semantics as traditional RPC libraries (e.g., gRPC, Thrift). The goals for mRPC are to be fast, support flexible policy enforcement, and provide high availability for applications.

Figure 2 shows a high-level overview of the mRPC architecture and workflow, breaking it down into three major phases: initialization, runtime, and management. The mRPC service runs as a non-root, user-space process with access to the necessary network devices and a shared-memory region for each application. In each of the phases, we focus on the view of a single machine that is running both the RPC client application and the mRPC service. The RPC server may also run alongside an mRPC service. In this case, mRPC-specific marshalling can be used. However, we also support flexible marshalling to enable mRPC applications to interact with external peers using well-known formats (e.g., gRPC). In our evaluation, we focus on cases where both the client and server employ mRPC.

The initialization phase extends from building the application to how the application binds to a specific RPC interface. (1) Similar to gRPC, users define a protocol schema. The mRPC schema compiler uses this to generate stub code to include in their application. We illustrate this using a key-value storage service with a single `Get` function. (2) When the application is deployed, it connects with the mRPC service running on the same machine and specifies the protocol(s) of interest, which are maintained by the generated stub. (3) The mRPC service also uses the protocol schema to generate,

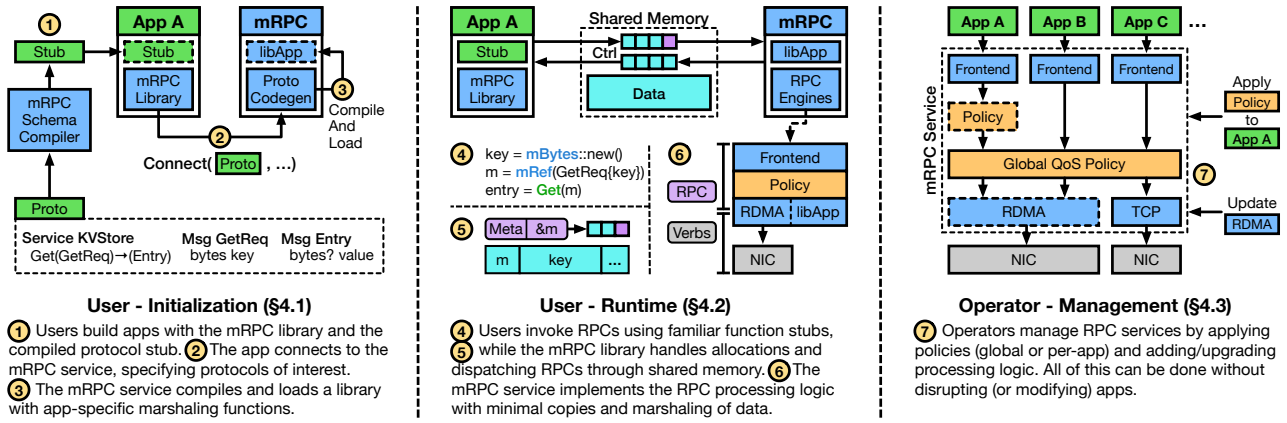


Figure 2: Overview of the mRPC workflow from the perspective of the users (and their applications) as well as infrastructure operators.

compile, and dynamically load a protocol-specific library containing the marshalling and unmarshalling code for that application's schemas[2]. This *dynamic binding* is a key enabler for mRPC to act as a long-running service, handling arbitrary applications (and their RPC schemas). [3]

At this point, we enter the runtime phase in which the application begins to invoke RPCs. Our approach uses *shared memory* between the application and mRPC, containing both control queues as well as a data buffer. (4) The application protocol stub produced by the mRPC protocol compiler can be called like a traditional RPC interface, with the exception that data structures passed as arguments or as return values must be allocated on a special heap in the shared data buffer. As an example, we show an excerpt of Rust-like pseudocode for invoking the `Get` function. (5) Internally, the stub and mRPC library manage RPC calls and replies in the control queues along with allocations and deallocations in the data buffer. (6) The mRPC service operates over the RPCs through modular *engines* that are composed to implement the per-application *datapaths* (i.e., sequence of RPC processing logic); each engine is responsible for one type of task (e.g., application interface, rate limiting, transport interface). Engines do not contain execution contexts, but are rather scheduled by *runtimes* in mRPC that correspond to kernel-level threads; during their execution, engines read from input queues, perform work, and enqueue outputs. External-facing engines (i.e., frontend, transport) use asynchronous control queues, while all other engines are executed synchronously by a runtime. Application control queues are contained in shared memory with the mRPC service.

This architecture, along with dynamic binding, enables mRPC to *operate over RPCs rather than packets*, avoiding the high overhead of traditional sidecar-based approaches. Additionally, the modular design of mRPC's processing logic enables mRPC to take advantage of fast network hardware (e.g., RDMA and smartNICs) in a manner that is transparent to the application. A key challenge, which we will address in §4.2, is how to securely enforce operator policies over RPCs in shared memory while minimizing data copies.

Finally, mRPC aims to improve the manageability of RPCs by infrastructure operators. Here, we zoom out to focus on the processing logic across all applications served by an mRPC service. (7) Operators may wish to apply a number of different policies to RPCs made by applications, whether on an individual basis (e.g., rate limiting, access control) or globally across applications (e.g., QoS). mRPC allows operators to add, remove, update, or reconfigure policies at runtime. This flexibility extends beyond policies to include those responsible for interacting with the network hardware. A key challenge, which we will address in §4.3, is in supporting the *live upgrade* of mRPC engines without interrupting running applications (and while managing engines sharing memory queues).

4 Design

In this section, we describe how mRPC provides dynamic binding, efficient policy and observability support, live upgrade, and security.

4.1 Dynamic RPC Binding

Applications have different RPC schemas, which ultimately decide how an RPC is marshalled. In the traditional RPC-as-a-library approach, a protocol compiler generates the marshalling code, which is linked into the application. In our design, the mRPC service is responsible for marshalling, which means that the application-specific marshalling code needs to be decoupled from an RPC library and run inside the mRPC service itself. Failing to ensure this separation would allow arbitrary code execution by a malicious user.

Applications directly submit the RPC schema (and not marshalling code) to the mRPC service. The mRPC service generates the corresponding marshalling code, then compiles and dynamically loads the library. Thus, we rely on our mRPC service code generator to produce the correct marshalling code

[2] Note that such libraries may be prefetched and/or cached to optimize the startup time.

[3] The dashed box of "Stub" and "libApp" means they are generated code.

for *any* user-provided RPC schema. For the initial handshake between an RPC client and an RPC server, the two mRPC services check that the provided RPC schemas match, and if not, the client's connection is rejected.

There are three remaining questions. First, **what are the responsibilities of the in-application user stub and mRPC library?** In mRPC, applications rely on user stubs to implement the abstraction as specified in their RPC schema. This means we still need to generate the glue code to maintain the traditional application programming interface. Our solution is to provide a separate protocol schema compiler, which is untrusted and run by application developers, to generate the user stub code that does not involve marshalling and transport. The application RPC stub (with the help of the mRPC library) creates a message buffer that contains the metadata of the RPC, with typed pointers to the RPC arguments, on the shared memory heap. The message is placed on a shared memory queue, which will be processed by the mRPC service. The receiving side works in a similar way.

Second, **does this approach increase RPC connect/bind time?** Implemented naively, this design will increase the RPC connect/bind time because the mRPC service has to compile the RPC schema and load the resulting marshalling library when an RPC client first connects to a corresponding server (or equivalently when an RPC server binds to the service). However, this latency is not fundamental to our design, and we can mitigate it in the following way. The mRPC service accepts RPC schemas before booting an application, as a form of prefetching. Given a schema, it compiles and caches the marshalling code. At the time of RPC connect/bind, the mRPC service simply performs a cache lookup based on the hash of the RPC schema. If it exists within the cache, the mRPC service will load the associated library; otherwise, the mRPC service will invoke the compiler to generate (and subsequently cache) the library. This reduces the connect/bind time from several seconds to several milliseconds.

Third, **when new applications arrive, do existing applications face downtime?** The multi-threaded mRPC service is a single process that serves many RPC applications; however, the marshalling engines for different RPC applications are not shared. They are in different memory addresses and can be (un)loaded independently. We will describe in §4.3 how to load/unload engines without disrupting running applications.

4.2 Efficient RPC Policy Enforcement and Observability

We have one key idea to allow efficient RPC policy enforcement and observability: senders should marshal once (as late as possible), while receivers should unmarshal once (as early as possible). On the sender side, we want to support policy enforcement and observability directly over RPCs from the application, and then marshal the RPC into packets. The receiver side is similar: packets should be unmarshalled into RPCs, applying policy and observability operations, and then delivered directly to the application. Compared to the traditional RPC-as-a-library approach with sidecars, this eliminates the redundant (un)marshalling steps (see Figure 1).

Data: DMA-capable shared memory heaps. Our design is centered around a dedicated shared memory heap between each application and the mRPC service. (Note that this heap is not shared across applications.) Applications directly create data structures, which may be used in RPC arguments, in a shared memory heap with the help of the mRPC library. Each application has a separate shared memory region, which provides isolation between (potentially mutually distrusting) applications. The mRPC library also includes a standard slab allocator for managing object allocation on this shared memory. If there is insufficient space within the shared memory, the slab allocator will request additional shared memory from the mRPC service and then map it into the application's address space. The mRPC service has access to the shared memory heap, allowing it to execute RPC processing logic over the application's RPCs, but also maintains a private memory heap for necessary copies.

Figure 3 shows an example workflow that includes access control for a key-value store service. Having the data structures directly in the shared memory allows an application to provide pointers to data, rather than the data itself, when submitting RPCs to the mRPC service. We call the message sent from an application to the mRPC service an *RPC descriptor*. If there are multiple RPC arguments, the RPC descriptor points to an array of pointers (each pointing to a different argument on the heap).

Let us say we have an ACL policy that rejects an RPC if the key matches a certain string. The mRPC service first copies the argument (i.e., `key`), as well as all parental data structures (i.e., `GetReq`), onto its private heap. This is to prevent time-of-use-to-time-of-check (TOCTOU) attacks. Since applications have access to DMA-capable shared memory at all times, an application could modify the content in the memory while the mRPC service is enforcing policies. Copying arguments is a standard mitigation technique, similar to how OS kernels prevents TOCTOU attacks by copying system call arguments from user- to kernel-space. This copying only needs to happen if the policy behavior is based on the content of the RPC. We demonstrate in §7.2 that even with such copying, mRPC's overhead for an ACL policy is much lower than gRPC + Envoy. The RPC descriptor is modified so that the pointer to the copied argument now points to the private heap. On the receiver side, the TOCTOU attack is not relevant, but we need to take care not to place RPCs directly in shared memory. If there is a receive-side policy that depends on RPC argument values, the mRPC service first receives the RPC data into a private heap; it copies the RPC data into the shared heap after policy processing. This prevents the application from reading RPC data that should have been dropped or modified by the policies. Note that we can bypass this copy when processing does not depend on RPC argument values (e.g., rate limits). During ACL policy enforcement, the RPC is dropped if the `key` argument is contained in a blocklist. Note that if an RPC is dropped, any further processing logic is never executed (including marshalling operations).

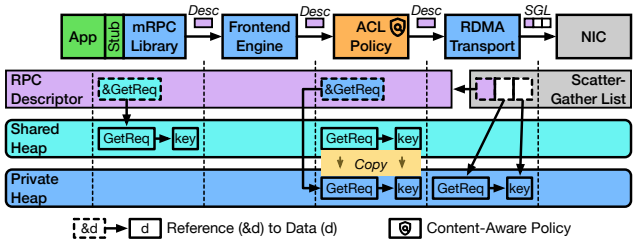


Figure 3: Overview of memory management in mRPC. Shows an example for the `Get` RPC that includes a content-aware ACL policy.

Finally, at the end of the processing logic, the transport adapter engine executes. mRPC currently supports two types of transport: TCP and RDMA. For TCP, mRPC uses the standard, kernel-provided scatter-gather (`iovec`) socket interface. For RDMA, mRPC uses the scatter-gather `verb` interface, allowing the NIC to directly interact with buffers on the shared (or private) memory heaps containing the RPC metadata and arguments. For both TCP and RDMA, mRPC provides disjoint memory blocks to the transport layer directly, eliminating excessive data movements.[4]

Control: Shared-memory queues. To facilitate efficient communication between an application and the mRPC service, we use shared memory control queues. mRPC allocates two unidirectional queues for sending and receiving requests from an application to the mRPC service. The requests contain RPC descriptors, which reference arguments on the shared memory heap. The mRPC service always copies the RPC descriptors applications put in the sending queue to prevent TOCTOU attacks. mRPC provides two options to poll the queues: 1) busy polling, and 2) `eventfd`-based adaptive polling. In busy polling, both the application-side mRPC library and the mRPC service busy poll on their ends of the queues. In the `eventfd` approach, the mRPC library and the mRPC service sends event notifications after enqueuing to an empty queue. After receiving a notification, the queue is drained (performing the necessary work) before subsequently waiting on future events. The `eventfd` approach saves CPU cycles when queues are empty. Other alternative solutions may involve dynamically scaling up (or down) the number of threads used to busy poll by the mRPC service; however, we chose the `eventfd` approach for its simplicity. In our evaluation, we use busy polling for RDMA and eventfd-based adaptive polling for TCP.

Memory management. We provide a memory allocator in the mRPC library for applications to directly allocate RPC data structures to be sent on a shared memory heap. The allocator invokes the mRPC service to allocate shared memory regions on behalf of the application (similar to how a standard heap manager calls `mmap` or `sbrk` to allocate memory from an OS kernel). We need to use a specialized memory allocator for RPC messages (and their arguments), since RPCs are shared between three entities: the application, the mRPC service, and the NIC. A memory block is safe to be reclaimed only when it will no longer be accessed by any entity.

We adopt a notification-based mechanism for memory management. On the sender side, the outgoing messages are managed by the mRPC library within the application. On the receiver side, the incoming messages are managed by the mRPC service. When the application no longer accesses a memory block occupied by outgoing messages, the memory block will not be reclaimed until the library receives a notification from mRPC service that the corresponding messages are already sent successfully through the NIC (similar to how zero-copy sockets work in Linux). Incoming messages are put in buffers on a separate read-only shared heap. The receiving buffers can be reclaimed when the application finishes processing (e.g., when the RPC returns). To support reclamation of receive buffers, the mRPC library notifies the mRPC service when specific messages are no longer in use by the application. Notifications for multiple RPC messages are batched to improve performance. If the receiver application code wishes to preserve or modify the incoming data, it must make an explicit copy. Although this differs from traditional RPC semantics, in our implementation of Masstree and DeathStarBench we found no examples where the extra copy was necessary.

Cross-datapath policy engines. mRPC supports engines that operate over multiple datapaths, which may span multiple applications. For instance, any global policy (e.g., QoS) will need to operate over all datapaths (see §5). For this type of engine, we instantiate replicas of the engine for each datapath that it applies to. Replicas can choose to either communicate through shared state, which requires managing contention across runtimes, or support runtime-local state that is contention-free.

4.3 Live Upgrades

Although our modular engine design for the mRPC service is similar to Snap [58] and Click [47], we arrive at very different designs for upgrades. Click does not support live upgrades, while Snap executes the upgraded process to run alongside the old process. The old process serializes the engine states, transfers them to the new process, and the new process restarts them. This means that even changing a single line of code within a single Snap engine requires a complete restart for all Snap engines. This design philosophy is fundamentally not compatible with mRPC, as we need to deal with new applications arriving with different RPC schemas, and thus our upgrades are more frequent. In addition, we want to avoid fate sharing for applications: changes to an application's datapath should not impact the performance of other applications. Ultimately, Snap is a network stack that does not contain application-specific code, where as mRPC needs to be application-aware for marshalling RPCs.

[4] For RDMA, if the number of disjoint memory blocks exceeds the limit of NIC's capability to encapsulate all blocks in one RDMA work request, mRPC coalesces the data into a memory block before transmission. This is because sending a single work request (even with a copy) is faster than sending multiple smaller work requests on our hardware.

We implement engines as plug-in modules that are dynamically loadable libraries. We design a live upgrade method that supports *upgrading, adding, or removing components of the datapath without disrupting other datapaths*.

Upgrading an engine. To upgrade one engine, mRPC first detaches the engine from its runtime (preventing it from being scheduled). Next, mRPC destroys and deallocates the old engine, but maintains the old engine's state in memory; note that the engine is detached from its queues and not running at this time. Afterwards, mRPC loads the new engine and configures its send and receive queues. The new engine starts with the old engine's state. If there is a change in the data structures of the engine's state, the upgraded engine is responsible for transforming the state as necessary (which the engine developer must implement). Note that this also applies to any shared state for cross-datapath engines. The last step is for mRPC to attach the new engine to the runtime.

Changing the datapath. When an operator changes the datapath to add or remove an engine, this process now involves the creation (or destruction) of queues and management of in-flight RPCs. Changes that add an engine are straightforward, since it only involves detaching and reconfiguring the queues between engines. Changes that remove an engine are more complex, as some in-flight RPCs may be maintained in internal buffers; for example, a rate limiter policy engine maintains an internal queue to ensure that the output queue meets a configured rate. Engine developers are responsible for flushing such internal buffers to the output queues when the engines are removed.

Multi-host upgrades or datapath changes. Some engine upgrades or datapath changes that involve both the sender and the receiver hosts need to carefully manage in-flight RPCs across hosts. For example, if we want to upgrade how mRPC uses RDMA, both the sender and the receiver have to be upgraded. In this scenario, the operator has to develop an upgrade plan that may involve upgrading an existing engine to some intermediate, backward-compatible engine implementation. The plan also needs to contain the upgrade sequence, e.g., upgrading the receiver side before the sender side. Our evaluation demonstrates such a complex live upgrade, which optimizes the handling of many small RPC requests over RDMA (see §7.3).

4.4 Security Considerations

We envision two deployment models for mRPC: (1) a cloud tenant uses mRPC to manage its RPC workloads (similar to how sidecars are used today); (2) a cloud provider uses mRPC to manage RPC workloads on behalf of tenants. In both models, there are two different classes of principals: operators and applications. Operators are responsible for configuring the hardware/virtual infrastructure, deploying the mRPC service, and setting up policies that mRPC will enforce. Applications run on an operator's infrastructure, interacting with the mRPC service to invoke RPCs. Applications trust operators, along with all privileged software (e.g., OS) and hardware that the operators provide; both applications and operators trust our mRPC service and protocol compiler. In both deployment models, applications are not trusted and may be malicious (e.g., attempt to circumvent network policies).

In the first deployment model, mRPC service runs on top of a virtualized network that is dedicated to the tenant. Running arbitrary policy and observability code inside the mRPC service cannot attack other tenants' traffic since inter-tenant isolation is provided by the cloud provider. In the second deployment model, our current prototype does not support running tenant-provided policy implementation inside mRPC service. How to safely integrate tenant-provided policy implementation and a cloud provider's own policy implementation is a future work.

From the application point of view, we want to ensure that **mRPC provides equivalent security guarantees as compared to today's RPC library and sidecar approach**, which we discuss in terms of: 1) dynamic binding and 2) policy enforcement. Our dynamic binding approach involves the generation, compilation, and runtime loading of a shared library for (un)marshalling application RPCs. Given that the compiled code is based on the application-provided RPC schema, this is a possible vector of attack. The mRPC schema compiler is trusted with a minimal interface: other than providing the RPC schema, applications have no control on the process of how the marshalling code is generated. We open source our implementation of the compiler so that it can be publicly reviewed.

As for all of our RPC processing logic, policies are enforced over RPCs by operating over their representations in shared memory control queues and data buffers. With a naive shared memory implementation, this introduces a vector of attack by exploiting a time-of-check to time-of-use (TOCTOU) attack; for instance, the application could modify the RPC message after policy enforcement but before the transport engine handles it. In mRPC, we address this by copying data into an mRPC-private heap prior to executing any policy that operates over the content of an RPC (as opposed to metadata such as the length). Similarly, received RPCs cannot be placed in shared memory until all policies have been enforced, since otherwise applications could see received RPCs before policies have a chance to drop (or modify) them. Shared memory regions are maintained by the mRPC service on a per-application basis to provide isolation.

5 Advanced Manageability Features

mRPC's architecture creates an opportunity for advanced manageability features such as cross-application RPC scheduling. In this section, we present two such features that we developed on our policy engine framework to demonstrate the broader utility of our RPC-as-a-managed-service architecture.

Feature 1: Global RPC QoS. mRPC allows centralized RPC scheduling of cross-application workloads based on a global view of current outstanding RPCs. For example, mRPC can enforce a policy that prioritizes RPCs with earliest deadlines [86] across applications to support latency SLO or prioritizes

latency-sensitive workloads [101]. One challenge here is that a naive implementation may attempt to apply the QoS policy for datapaths spread over multiple runtimes (i.e., execution thread contexts). This would require the (replicated) policy engines on each datapath to share the state on outstanding RPCs, and thus impose synchronization overheads. Therefore, we adopt a similar strategy as used in the Linux kernel to apply the QoS policy on a per-runtime basis, which instead can use runtime-local storage without the need for synchronization. In our implementation, we support a QoS strategy that prioritizes small RPCs based on a configurable threshold size.

Feature 2: Avoiding RDMA performance anomalies. It is well known that RDMA workloads may not fully utilize the capability of a specific RDMA NIC without fine-tuning, and that particular traffic patterns can even cause performance anomalies [40, 49] (e.g., low RDMA throughput, pause frame storms). Previous work such as ScaleRPC [13] and Flock [63] have proposed techniques to utilize the RNIC more efficiently. However, their approaches are library-based and only work for single applications; therefore, they do not handle scenarios in which the *combination* of multiple application workloads causes poor RDMA performance. mRPC's architecture enables us to have a global view of all RDMA requests and to avoid such performance anomalies.

We implement a global RDMA scheduler inside the RDMA transport engine, which translates RPC requests into RDMA messages and sends them to the RDMA NIC. In our implementation, we focus on addressing the performance degradation from interspersed small and large scatter-gather elements (which may be across RPCs as well as applications). We fuse such elements together with an explicit copy with an upper bound of 16 KB for the size of the fused element.

6 Implementation

mRPC is implemented in 32K lines of Rust: 3K lines for the protocol compiler, 6K for the mRPC control plane, 12K for engine implementations, and 11K for the mRPC library. The mRPC control plane is part of the mRPC service that loads/unloads engines.

The mRPC control plane is not live-upgradable. The mRPC library is linked into applications and is thus also not live-upgradable. We do not envision the need to frequently upgrade these components because they only implement the high-level, stable APIs, such as shared memory queue communication and (un)loading engines.

Engine interface. Table 1 presents the essential API functions that all engines must implement. Each engine represents some asynchronous computation that operates over input and output queues via `doWork`, which is similar in nature to Rust's `Future`. mRPC uses a pool of runtime executors to drive the engines by calling `doWork`, where each runtime executor corresponds to a kernel thread. We currently implement a simple scheduling strategy inspired by Snap [58]: engines can be scheduled to

Operations
`doWork`(in:[Queue], out:[Queue])
Operate over one or more RPCs available on input queues.
`decompose`(out:[Queue]) → State
Decompose the engine to its compositional states.
(Optionally output any buffered RPCs)
`restore`(State) → Engine
Restore the engine from the previously decomposed state.

Table 1: mRPC Engine Interface.

a dedicated or shared runtime on start. In addition, runtimes with no active engines will be put to slept and release CPU cycles. The engines also implement APIs to support live upgrading: `decompose` and `restore`. In `decompose`, the engine implementation is responsible for destructing the engine and creating a representation of the final state of the engine in memory, returning a reference to mRPC. mRPC invokes `restore` on the upgraded instance of the engine, passing in a reference to the final state of the old engine. The developer is responsible for handling backward compatibility across engine versions, similar to how application databases may be upgraded across changes to their schemas.

Transport engines. We abstract reliable network communication of messages into transport engines, which share similar design philosophy with Snap [58] and TAS [45]. We currently implement two transport engines: RDMA and TCP. Our RDMA transport engine is implemented based on OFED libibverbs 5.4, while our TCP transport engine is built on Linux kernel's TCP socket.

mRPC Library. Modern RPC libraries allow the user to specify the RPC data types and service interface through a language-independent schema file (e.g., `protobuf` for gRPC, `thrift` for Apache Thrift). mRPC implements support for `protobuf` and adopts similar service definitions as gRPC, except for gRPC's streaming API. mRPC also integrates with Rust's async/await ecosystem for ease of asynchronous programming in application development.

To create an RPC service, the developer only needs to implement the functions declared in the RPC schema. The dependent RPC data types are automatically generated and linked with the application by the mRPC schema compiler. The mRPC library handles all the rest, including task dispatching, thread management, and error handling. To allow applications to directly allocate data in shared memory without changing the programming abstraction, we implement a set of shared memory data structures that expose the same rich API as Rust's standard library. This is done by replacing the memory allocation of data structures such as `Vec` and `String` with the shared memory heap allocator.

7 Evaluation

We evaluate mRPC using an on-premise testbed of servers with two 100 Gbps Mellanox Connect-X5 RoCE NICs and two Intel 10-core Xeon Gold 5215 CPUs (running at 2.5 GHz

base frequency). The machines are connected via a 100 Gbps Mellanox SN2100 switch. Unless specified otherwise, we keep a single in-flight RPC to evaluate latency. To benchmark goodput and RPC rate, we let each client thread keep 128 concurrent RPCs on TCP and 32 concurrent RPCs on RDMA.

7.1 Microbenchmarks

We first evaluate mRPC's performance through a set of microbenchmarks over two machines, one for the client and the other for the server. The RPC request has a byte-array argument, and the response is also a byte array. We adjust the RPC size by changing the array length. RPC responses are an 8-byte array filled with random bytes. We compare mRPC with two state-of-the-art RPC implementations, eRPC and gRPC (v1.48.0). We deploy Envoy (v1.20) in HTTP mode to serve as a sidecar for gRPC. We use mRPC's TCP and RDMA backends to compare with gRPC and eRPC, respectively. There is no existing sidecar that supports RDMA. To evaluate the performance of using a sidecar to control eRPC traffic, we implement a single-thread sidecar proxy using the eRPC interface. We keep applications running for 15 seconds to measure the result.

Small RPC latency. We evaluate mRPC's latency by issuing 64-byte RPC requests over a single connection. Table 2 shows the latency for small RPC requests. Note that since the marshalling of small messages is fast on modern CPUs, the result in the table remains stable even when the message size scales up to 1 KB. We use `netperf` and `ib_read_lat` to measure raw round-trip latency.

mRPC achieves median latency of 32.8 µs for TCP and 7.6 µs for RDMA. Relative to netperf (TCP) or a raw RDMA read, mRPC adds 11.8 or 5.1 µs to the round-trip latency. This is the cost of the mRPC abstraction on top of the raw transport interface (e.g., socket, verbs).

We also evaluate latency in the presence of sidecar proxies. The sidecars do not enforce any policies, so we are only measuring the base overhead. Our results show that adding sidecars substantially increases the RPC latency. On gRPC, adding Envoy sidecars more than triples the median latency. The result is similar with eRPC. On mRPC, having a NullPolicy engine (which simply forwards RPCs) in the mRPC service has almost no effect on latency, increasing the median latency only by 300 ns.

Comparing the full solution (mRPC with policy versus gRPC/eRPC with proxy), mRPC speeds up the median latency by 6.1× (i.e., 33.4 µs against 203.4 µs) and the 99th percentile tail latency by 5.8×. On RDMA, mRPC speeds up eRPC by 1.3× and 1.4× in terms of median and tail latency (respectively). This is because the communication between the eRPC app and its proxy goes through the NIC, which triples the cost in the end-host driver (including the PCIe latency). In contrast, mRPC's architecture shortcuts this step with shared memory.

In addition, to separate the performance gain from system implementation difference, we evaluate the latency of mRPC with full gRPC-style marshalling (protobuf encoding and HTTP/2 framing) in the presence of NullPolicy engines as an ablation study. Under this setting, compared with gRPC + Envoy, mRPC speeds up the latency by 4.1× in terms of both median and tail latency. We also observe that the mRPC framework does not introduce significant overhead. Even with the cost of protobuf and HTTP/2 encoding, mRPC still achieves slightly lower latency compared with standalone gRPC. In mRPC, we can choose a customized marshalling format, because we know the other side is also an mRPC service. In other cases, e.g., when interfacing with external traffic or dealing with endianness differences, we can still apply full-gRPC style marshalling. When mRPC is configured to use full-gRPC style marshalling, we only need to pay (un)marshalling costs between mRPC services. For gRPC + Envoy, in addition to the (un)marshalling costs between Envoy proxies, the communication between applications and Envoy proxies also needs to pay this (un)marshalling cost. In the remaining evaluations, we will use mRPC's customized marshalling protocol. More results using gRPC-style marshalling are shown in §A.1.

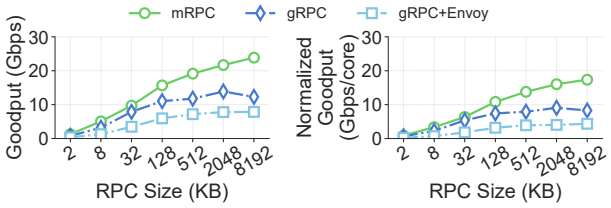
Transport	Solution	Median Latency (µs)	P99 Latency (µs)
TCP	Netperf	21.0	32.0
	gRPC	63.0	90.3
	mRPC	32.8	38.7
	gRPC+Envoy	203.4	251.1
	mRPC+NullPolicy	**33.4**	**43.3**
	mRPC+NullPolicy+HTTP+PB	49.8	61.9
RDMA	RDMA read	2.5	2.8
	eRPC	3.6	4.1
	mRPC	7.6	8.7
	eRPC+Proxy	11.3	15.6
	mRPC+NullPolicy	**7.9**	**9.1**

Table 2: **Microbenchmark [Small RPC latency]:** Round-trip RPC latencies for 64-byte requests and 8-byte responses.

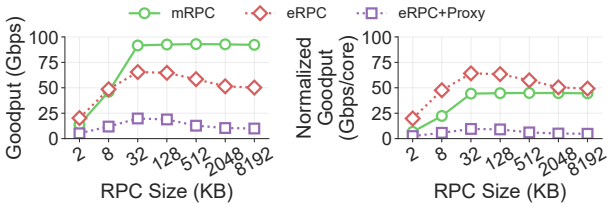
Large RPC goodput. The client and server in our goodput test use a single application thread. The left side of Figure 4 shows the result. From this point on, when we discuss mRPC's performance, we focus on the performance of mRPC that has at least a NullPolicy engine in place to fairly compare with sidecar-based approaches.

mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.1× and 9.3×, respectively, for 8KB RPC requests. mRPC is especially efficient for large RPCs[5], for which (un)marshalling takes a higher fraction of CPU cycles in the end-to-end RPC datapath. Having a sidecar substantially hurts RPC goodput both for TCP and RDMA. In particular, for RDMA, intra-host roundtrip traffic through the RNIC might contend with inter-host traffic in the RNIC/PCIe bus, halving the available bandwidth for inter-host traffic. mRPC even outperforms gRPC (without Envoy). mRPC is fundamentally more efficient in terms of marshalling format: mRPC uses `iovec` and incurs no data movement. §A.1 shows an ablation study that demonstrates that even if mRPC uses a full gRPC-style marshalling engine, mRPC outperforms gRPC + Envoy due to a reduction in the number of (un)marshalling steps.

CPU overheads. To understand the mRPC CPU overheads, we measure the per-core goodput. The results are shown on

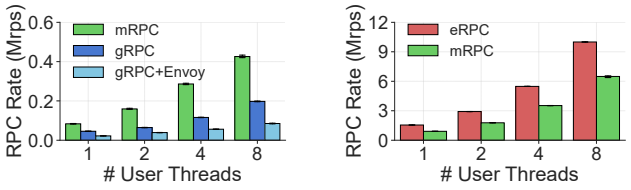


(a) TCP-based transport



(b) RDMA-based transport

Figure 4: **Microbenchmark [Large RPC goodput]:** Comparison of goodput for large RPCs. Note that different solutions demand different amounts of CPU cores, so we also normalized the goodput to their CPU utilization, as shown in the right figures. The error bars show the 95% confidence interval, but they are too small to be visible.



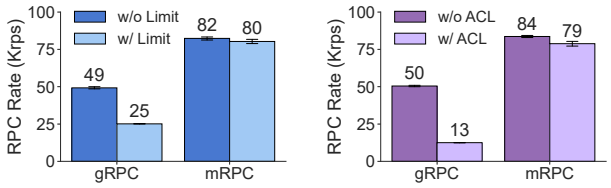
(a) TCP-based transport (b) RDMA-based transport

Figure 5: **Microbenchmark [RPC rate and scalability]:** Comparison of small RPC rate and CPU scalability. The bars show the RPC rate. The error bars show the 95% confidence interval.

the right side of Figure 4. mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.8× and 9.3×, respectively. This means mRPC is much more CPU-efficient than gRPC + Envoy and eRPC + Proxy. eRPC (without a proxy) is quite efficient, but converges to mRPC's efficiency as RPC size increases.

RPC rate and scalability. We evaluate mRPC's small RPC rate and its multicore scalability. We fix the RPC request size to 32 bytes and scale the number of client threads. We use the same number of threads for the server as the client, and each client connects to one server thread. Figure 5 shows the RPC rates when scaling from 1 to 8 user threads. All the tested solutions scale well. mRPC's RPC rates scale by 5.1× and 7.2×, on TCP and RDMA, from a single thread to 8 threads. As a reference, gRPC scales by 4.3×, gRPC + Envoy scales by 3.9×, and eRPC scales by 6.5×. mRPC achieves 0.43 Mrps on TCP and 6.5 Mrps on RDMA with 8 threads. gRPC + Envoy only has 0.09 Mrps, so mRPC outperforms it by 5×. We do not evaluate eRPC + proxy, because our eRPC proxy is only single-threaded. When we run eRPC + proxy with a single thread, it achieves 0.51 Mrps. So even if eRPC + proxy scales linearly to 8 threads, mRPC still outperforms it.

[5] Standalone eRPC exhibits relatively lower goodput on RoCE than on Infiniband. According to the eRPC paper [39], eRPC should achieve 75 Gbps on Infiniband for 8MB RPCs.



(a) Rate Limiting (b) Access Control

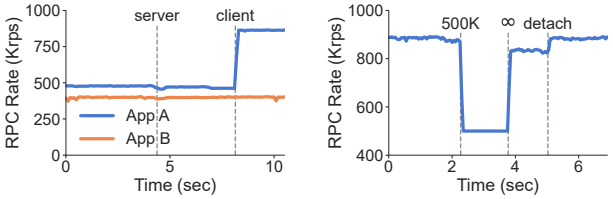
Figure 6: **Efficient Support for Network Policies**. The RPC rates with and without policy are compared. The bars of w/o Limit and w/o ACL for gRPC show its throughput when the sidecar is bypassed. The error bars show the 95% confidence interface.

7.2 Efficient Policy Enforcement

We use two network policies as examples to demonstrate mRPC's efficient support for RPC policies: (1) RPC rate limiting and (2) access control based on RPC arguments. RPC rate limiting allows an operator to specify how many RPCs a client can send per second. We implement rate limiting as an engine using the token bucket algorithm [91]. Our access control policy inspects RPC arguments and drops RPCs based on a set of rules specified by network operators. These two network policies differ greatly from traditional rate limiting and access control, which only limit network bandwidth and can only operate on packet headers.

We compare rate limit enforcement using an mRPC policy versus using Envoy's rate limiter on gRPC workloads. To evaluate the performance overheads, we set the limit to infinity so that the actual RPC rate is never above the limit (allowing us to observe the overheads). Figure 6a shows the RPC rate with and without the rate limits. gRPC's RPC rate drops immediately from 49K to 25K. This is because having a sidecar proxy (Envoy) introduces substantial performance overheads. For mRPC, the RPC rate stays the same at 82K. This is because having a policy introduces minimal overheads. The extra policy only adds tens to hundreds of extra CPU instructions on the RPC datapath.

We evaluate access control on a hotel reservation application in DeathStarBench [23]. The service handles hotel reservation RPC requests, which include the customer's name, the check-in date, and other arguments. The service then returns a list of recommended hotel names. We set the access control policy to filter RPCs based on the `customerName` argument in the request. We use a synthetic workload containing 99% valid and 1% invalid requests. We again compare our mRPC policy



(a) Transport adapter (b) Rate limit policy

Figure 7: **Live upgrade**. In (a), the annotations indicate when the client of App A and server of A and B are upgraded. In (b), the annotations denote the specified rate and when the policy is removed.

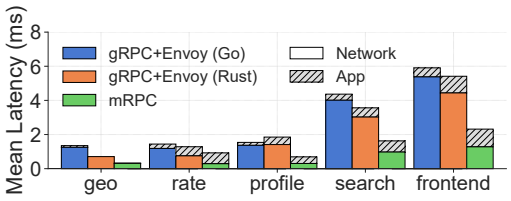


Figure 8: **DeathStarBench**: Mean latency of in-application processing and network processing of microservices. The latency of a microservice includes RPC calls to other microservices. The frontend latency represents complete end-to-end latency.

against using Envoy to filter gRPC requests. We implement the Envoy policy using WebAssembly. gRPC's rate drops from 50K to 13K. This is because of the same sidecar overheads and now Envoy has to further parse the packets to fetch the RPC arguments. On mRPC, the performance drop is much smaller, from 84K to 79K. Note that, on mRPC, the performance overhead of introducing access control is larger than rate limiting. For access control, the mRPC service has to copy the relevant field (i.e., `customerName`) to the private heap to prevent TOCTOU attacks on the sender side and has to copy the RPC from a private heap to the shared heap on the receiver side.

7.3 Live Upgrade

We demonstrate mRPC's ability to live upgrade using two scenarios.

Scenario 1. During our development of mRPC, we realized that using the RDMA NIC's scatter-gather list to send multiple arguments in a single RPC can significantly boost mRPC's performance. In this approach, even when an RPC contains arguments that are scattered in virtual memory, we can send the RPC using a single RDMA operation (`ibv_post_send`). We use these two versions of our RDMA transport engine to demonstrate that mRPC enables such an upgrade without affecting running applications. Note that all other evaluations already include this RDMA feature. This upgrade involves both the client side's mRPC service and the server side's mRPC service, because it involves how RDMA is used between machines (i.e., transport adapter engine). gRPC and eRPC cannot support this type of live upgrade.

We run two applications (App A and App B). Both applications are sending 32-byte RPCs, and the responses are 8 bytes. A and B share the mRPC service on the server side. A's and B's RPC clients are on different machines. We keep 8 concurrent RPCs for B, forcing it to send at a slower rate, while using 32 for A. We first upgrade the server side to accept arguments as a scatter-gather list, and we then upgrade the client side of A. Figure 7a shows the RPC rate of A and B. When the server side upgrades, we observe a negligible effect on A's and B's rate. Neither A nor B needs recompilation or rebooting. When A's client side's mRPC service is upgraded, A's performance increases from 480K to 860K. B's performance is not affected at all because B's client side's mRPC service is not upgraded.

Scenario 2. Enforcing network policies has performance overheads, even when they do not have any effect. For example, enforcing a rate limit of an extremely large throttle rate still introduces performance overheads just for tracking the current rate using token buckets. mRPC allows policies to be removed at runtime, without disrupting running applications.

We use the same rate limiting setup from §7.2 but on top of RDMA transport. Figure 7b shows the RPC rate. We start from not having the rate limit engine. We then load the rate limit engine and set the throttled rate to 500K. The RPC rate immediately becomes 500K. We then set the throttled rate to be infinite, and the rate becomes 840K. After we detach the rate limit engine, the rate becomes 890K.

Takeaways. There are two overall takeaways from these experiments. First, mRPC allows upgrades to the mRPC service without disrupting running applications. Second, live upgrades allow for more flexible management of RPC services, which can be used to enable immediate performance improvements (without redeploying applications) or dynamic configuration of policies.

7.4 Real Applications

We evaluate how the performance benefits of mRPC transform into end-to-end application-level performance metrics.

DeathStarBench. We use the hotel reservation service from the DeathStarBench [23] microservice benchmark suite. The reference benchmark is implemented in Go with gRPC and Consul [15] (for service discovery). Our mRPC prototype currently only supports Rust applications, and we thus port the application code to Rust for comparison. We use the same open-source services such as memcached [59] and MongoDB [64].

We distribute the HTTP frontend and the microservices on four servers in our testbed. The monolithic services (memcached, MongoDB) are co-located with the microservices that depend on them. We use a single thread for each of the microservices and the frontend. Further, we deploy an Envoy proxy as a sidecar on each of the servers (with no active policy). The pro-

	Median Latency	P99 Latency	Throughput
eRPC	16.8 µs	21.7 µs	8.7 MOPS
mRPC	22.5 µs	33.1 µs	7.0 MOPS

Table 3: **Masstree analytics**: Latency and the achieved throughput for GET operations. MOPS is Million Operations Per Second.

vided workload generator [23] is used to submit HTTP requests to the frontend. For a fair comparison, we also implemented a Rust version of the benchmark with Tonic [93], which is the de facto implementation of gRPC in Rust. We deploy the mRPC and Tonic implementations on bare metal, while the reference Go suite runs in Docker containers with a host network (which introduces negligible performance overheads compared to using bare metal [103]). All three solutions are based on TCP. We issue 20 requests per second for 250 seconds and record the latency of each request, breaking it down into the in-application processing time and network processing time for each microservice involved. In our evaluation, the dynamic bindings of the user applications are already cached in mRPC service, so the time to generate the bindings is not included in the result.

Figure 8 shows the latency breakdown. First, we validate that our own implementation of DeathStarBench on Rust is a faithful re-implementation. We can see that the original Go implementation and our Rust implementation have similar latency. Moreover, the amount of latency spent in gRPC is similar. Second, mRPC with a null policy outperforms by 2.5× gRPC with a sidecar proxy in average end-to-end latency. §A.2 contains more details about the tail latency and the scenario without a sidecar.

Masstree analytics. We also evaluate the performance of Masstree [56], an in-memory key-value store, over both mRPC and eRPC [39] using RDMA. We follow the exact same workload setup used in eRPC, which contains 99% I/O-bounded point GET request and 1% CPU-bounded range SCAN request. We run the Masstree server on one machine and run the client on another machine. Both the server and the client use 10 threads, with each client thread using 16 concurrent requests. The test runs for 60 seconds. The result in Table 3 shows that eRPC outperforms mRPC, which makes sense since eRPC is a well-designed library implementation that is focused on high performance. mRPC enables many other manageability features in exchange for a slight reduction in performance. In this case, using mRPC instead of eRPC means that median latency increases by 34% and throughput reduces by 20%.

7.5 Benefits of Advanced Manageability Features

Next, we demonstrate the performance benefits of having centralized RPC management, through two advanced manageability features that we developed (see §5). We use synthetic workloads to test the advanced manageability features.

	Latency App		B/W App
	P95 Latency	P99 Latency	Bandwidth
w/o QoS	45.1 µs	54.6 µs	22.2 Gbps
w/ QoS	19.5 µs	21.8 µs	22.0 Gbps

Table 4: **Global QoS**: Performance of latency- and bandwidth-sensitive applications with and without a global QoS policy.

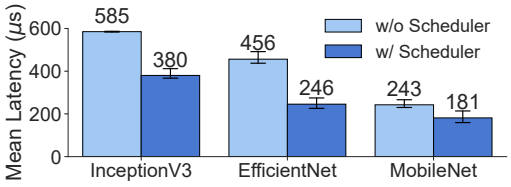


Figure 9: **RDMA Scheduler**: Mean RPC latency with or without RDMA scheduler. The error bars show the 95% confidence interval.

Global RPC QoS. We enable our cross-application QoS policy that reorders requests from multiple applications and prioritizes small RPC quests. We set up two applications and pin them to the same mRPC runtime. One application is latency-sensitive, sending 32-byte RPC requests with a single RPC in-flight; the other is bandwidth-sensitive, sending 32 KB requests with 64 concurrent RPCs. We measure the tail latency for the latency-sensitive application and the utilized bandwidth of the bandwidth-sensitive one.

Table 4 shows the result. Without the QoS policy, the bandwidth-sensitive application has a high bandwidth utilization; however, the latency-sensitive application suffers from a high tail latency. With the QoS policy in place, the small requests from the latency-sensitive application get higher priority and are sent first, improving P99 latency from 54.6 µs to 21.8 µs. Since small RPC requests consume negligible bandwidth, it barely affects the bandwidth-sensitive application (less than a 1% bandwidth drop).

RDMA Scheduler. Our RDMA scheduler batches small RPC requests into (at most) 16KB messages and sends requests using a single RMDA operation to reduce the load on the RDMA NIC. Our synthetic workload is based on BytePS [37], which uses RDMA for distributed deep learning. To synchronize a tensor to/from a server, BytePS prepends an 8-byte key and appends a 4-byte length to describe the tensor. The three disjoint memory blocks are placed in a scatter-gather list and submitted to the NIC, resulting in a small-large-small message pattern that triggers a performance anomaly [49]. This message pattern is quite common in real applications, as programs often need to describe a large payload with a small piece of metadata. We emulate BytePS's RPC request pattern and generate RPCs from three widely-used models: MobileNet, EfficientNetB0, and InceptionV3 [31, 89, 90]. Each RPC call consists of an

8-byte key, a payload of tensor, and a 4-byte length. We use a single thread to make RPCs. Figure 9 shows the average RPC latency. The RDMA scheduler provides 30-90% latency improvement. This improvement differs for different neural networks, because of different RDMA message patterns.

8 Related Work

Fast RPC implementations. Optimizing RPC has a long history. Birrell and Nelson's early RPC design [10] includes generating bindings via a compiler, interfacing with transport protocols, and various optimizations (e.g., implicit ACK). Bershad et al. showed how to use shared-memory queues to efficiently pass RPC messages between processes on the same machine [8]. mRPC's shared-memory region leverages this idea but extends it to allow for marshalling code to be applied after policy enforcement. A similar use of shared-memory queues can be found with recent Linux support for asynchronous system calls [3] combined with scatter-gather I/O [54]; unlike traditional system calls, however, mRPC protocol descriptions can be defined at runtime.

Another line of work uses RDMA to speed network RPCs [13, 39, 41, 63, 87, 88]. These studies assume direct application access to network hardware and are thus susceptible to RDMA's security weaknesses [79]. mRPC leverages ideas from RDMA RPC research but in a way that is compatible with policy enforcement and observability, by doing so as a service. Another line of work reduces the cost of marshalling, by using alternative formats [2, 9, 11, 20, 38, 66, 78, 92] or designing hardware accelerators [35, 43, 76, 97]. This work is largely orthogonal to our goal of removing unnecessary marshalling steps but could be applied to further improve mRPC performance.

Fast network stacks. Building efficient host network stacks is a popular research topic. MegaPipe [28], mTCP [36], Arrakis [73], IX [5], eRPC [39], and Demikernel [99] advocate building the network stack as a user-level library, bypassing the kernel for performance. In these systems, an application directly accesses the network interface, but they assume policy can be enforced by the network hardware and are thus vulnerable if the hardware has security weaknesses. mRPC can interpose policy on any RPC. Like mRPC, Snap [58] and TAS [45] implement the network stack as a service, but they stop at layer 4 (TCP and UDP) rather than layer 7 (RPC). Application RPC stubs must marshal data into shared memory queues to use Snap or TAS. Flexible policy engines are a key feature of Snap, but because Snap operates at layer 4, it can only apply layer 7 policies by unmarshalling and re-marshalling RPC data. A fast network stack like mRPC can also be implemented directly in the kernel. LITE [95] implements RDMA operations as system calls inside the kernel to improve manageability, and Shenango [69] interposes a specialized kernel packet scheduler for network messages.

Fast network proxies. There is a long line of work on improving the performance of network proxies [33, 34, 44, 46, 47, 51, 57, 60, 70, 71, 74, 75, 85, 100]. Much of this work considers the general case of a standalone proxy. Our work differs in two ways. First, our proposed technique is only for RPC traffic rather than generalized TCP traffic. Second, we co-design the application library stub and proxy, and thus, both must be co-located on the same machine for our shared memory queues to function. In today's sidecar proxies (our baseline), this assumption holds, but it does not hold for generalized network proxies.

Live upgrades of system software. Being able to update system software without disrupting or restarting applications is key to achieving end-to-end high availability. Snap [58] provides live upgrade of the network stack running as a proxy; Bento [61] provides similar functionality for kernel-resident file systems. Relative to these systems, mRPC upgrades are more fine-grained. For example, Snap targets a maximum outage during upgrades of 200 milliseconds, by spawning another instance of itself and moving all connections to the new process. By contrast, our goal is near instantaneous changes and upgrades to RPC protocol definitions, policy engines, and marshalling code. We accomplish this by keeping the control plane intact and performing updates by loading and unloading dynamic libraries. eBPF is a Linux kernel extensibility mechanism that supports dynamic updates [17]; unlike eBPF, mRPC can dynamically change the execution graph of policy engines as well as the individual engines themselves.

9 Conclusion

Remote procedure call has become the de facto abstraction for building distributed applications in datacenters. The increasing demand for manageability makes today's RPC libraries inadequate. Inserting a sidecar proxy into the network datapath allows for manageability but slows down RPC substantially due to redundant marshalling and unmarshalling. We present mRPC, a novel architecture to implement RPC as a managed service to achieve both high performance and manageability. mRPC eliminates the redundant marshalling overhead by applying policy to RPC data before marshalling and only copying data when necessary for security. This new architecture enables live upgrade of RPC processing logic and new RPC scheduling and transport methods to improve performance. We have performed extensive evaluations through a set of micro-benchmarks and two real applications to demonstrate that mRPC enables a unique combination of high performance, policy flexibility, security, and application-level availability. Our source code is available at `https://github.com/phoenix-dataplane/phoenix`.

Acknowledgement

We thank our shepherd Amy Ousterhout and other anonymous reviewers for their insightful feedback. Our work is partially supported by NSF grant CNS-2213387 and by gifts from Adobe, Amazon, IBM, and Meta.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] Apache Arrow. https://arrow.apache.org/, 2022.

[3] Jens Axboe. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2019.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *OSDI*, 2020.

[7] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.

[8] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 1991.

[9] Bincode. https://github.com/bincode-org/bincode, 2022.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 1984.

[11] Cap'n Proto. https://capnproto.org/, 2022.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[13] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *EuroSys*, 2019.

[14] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for µs-scale RPCs with Breakwater. In *OSDI*, 2020.

[15] Consul. https://www.consul.io/, 2022.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[17] eBPF. https://ebpf.io/, 2022.

[18] Envoy Proxy. https://www.envoyproxy.io/, 2022.

[19] etcd. https://etcd.io/, 2022.

[20] FlatBuffers. https://google.github.io/flatbuffers/, 2022.

[21] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *OSDI*, 2022.

[22] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS*, 2019.

[24] Gluster. https://www.gluster.org/, 2022.

[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[26] gRPC. https://grpc.io/, 2022.

[27] gRPC Release Schedule. https://grpc.github.io/grpc/core/md_doc_grpc_release_schedule.html, 2022.

[28] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O . In *OSDI*, 2012.

[29] HAProxy. http://www.haproxy.org/, 2022.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[31] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. https://arxiv.org/abs/1704.04861, 2017.

[32] Istio. https://istio.io/, 2022.

[33] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. SoftFlow: A Middlebox Architecture for Open vSwitch. In *ATC*, 2016.

[34] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *NSDI*, 2017.

[35] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W Lee. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *ISCA*, 2020.

[36] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[37] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[38] Introducing JSON. https://www.json.org/json-en.html, 2022.

[39] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*, 2016.

[41] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, 2016.

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *ISCA*, 2015.

[43] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A Hardware Accelerator for Protocol Buffers. In *MICRO*, 2021.

[44] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *NSDI*, 2018.

[45] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[46] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors. In *EuroSys*, 2015.

[47] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 2000.

[48] Xinhao Kong, Jingrong Chen, Wei Bai, Xu Yechen, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *NSDI*, 2023.

[49] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *NSDI*, 2022.

[50] Kubernetes. https://kubernetes.io/, 2022.

[51] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.

[52] Tianxi Li, Haiyang Shi, and Xiaoyi Lu. HatRPC: Hint-Accelerated Thrift RPC over RDMA. In *SC*, 2021.

[53] Linkerd. https://linkerd.io/, 2022.

[54] Rober Love. *Linux System Programming*. O'Reilly Media, 2007.

[55] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[56] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, 2012.

[57] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[58] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019.

[59] Memcached. https://memcached.org/, 2022.

[60] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM*, 2017.

[61] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High Velocity Kernel File Systems with Bento. In *FAST*, 2021.

[62] Jeffrey C. Mogul and John Wilkes. Nines Are Not Enough: Meaningful Metrics for Clouds. In *HotOS*, 2019.

[63] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *SOSP*, 2021.

[64] MongoDB. https://www.mongodb.com, 2022.

[65] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[66] MessagePack. https://msgpack.org/index.html, 2022.

[67] Nginx. https://www.nginx.com/, 2022.

[68] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *ATC*, 2014.

[69] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*, 2019.

[70] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[71] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 2019.

[73] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *OSDI*, 2014.

[74] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[75] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *NSDI*, 2019.

[76] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *ASPLOS*, 2020.

[77] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *SIGCOMM*, 2022.

[78] Protocol Buffers. https://developers.google.com/protocol-buffers, 2022.

[79] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *USENIX Security*, 2021.

[80] Russel Sandberg. The Sun Network File System: Design, Implementation and Experience. In *USENIX Summer ATC*, 1986.

[81] Mike Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transaction on Computer Systems*, February 1990.

[82] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[83] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *SIGCOMM*, 2021.

[84] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook white paper*, 5(8):127, 2007.

[85] Snabb: Simple and fast packet networking. https://github.com/snabbco/snabb, 2022.

[86] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium*, pages 2–11, 1994.

[87] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data Center RPC. In *SoCC*, 2014.

[88] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster than Server-Bypass with RDMA. In *EuroSys*, 2017.

[89] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *CVPR*, 2016.

[90] Mingxing Tan and Quoc Le. Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks. In *ICML*, 2019.

[91] Puqi Perry Tang and Tsung-Yuan Charles Tai. Network Traffic Characterization Using Token Bucket Model. In *INFOCOM*, 1999.

[92] Apache Thrift. https://thrift.apache.org/, 2022.

[93] Tonic. https://github.com/hyperium/tonic, 2022.

[94] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[95] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.

[96] Amin Vadhat. Coming of Age in the Fifth Epoch of Distributed Computing: The Power of Sustained Exponential Growth, 2020. Amin Vahdat - SIGCOMM Lifetime Achievement Award 2020 Keynote.

[97] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards Zero-Copy Serialization. In *HotOS*, 2021.

[98] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[99] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP*, 2021.

[100] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *SIGCOMM*, 2020.

[101] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI*, 2022.

[102] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng-Ju He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting Service Mesh Overheads. *ArXiv*, abs/2207.00592, 2022.

[103] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *NSDI*, 2019.

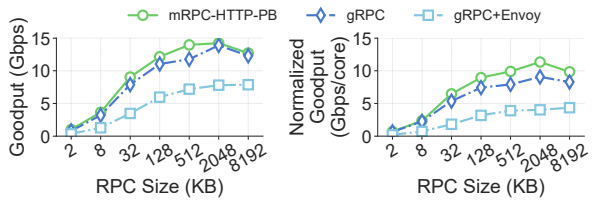


Figure 10: **Microbenchmark [Large RPC bandwidth]:** Comparison of large RPC bandwidth where we use HTTP/2 and protobuf (PB) marshalling for mRPC, on TCP transport. The error bars show the 95% confidence interval, but they are too small to be visible.

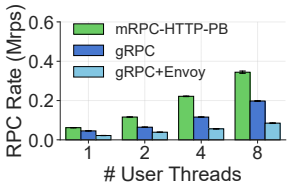


Figure 11: **Microbenchmark [RPC rate and scalability]:** Comparison of small RPC rate and CPU scalability where we use HTTP/2 and protobuf (PB) marshalling for mRPC, on TCP transport. The error bars show the 95% confidence interval.

A Appendix

A.1 mRPC with Full gRPC-style Marshalling

As gRPC uses `protobuf` [78] for encoding and HTTP/2 as the payload carrier, it has a memory copying and HTTP/2 framing cost. On the other hand, mRPC is agnostic to the marshalling format. Although mRPC's default marshalling is zero-copy and is generally faster than gRPC-style marshalling, our main goal of the paper is to show that we can eliminate the redundant (un)marshalling steps while enabling network policies and observability for RPC traffic.

To isolate the performance benefits of using zero-copy marshalling and reducing the number of (un)marshalling steps, we evaluate mRPC with full gRPC-style marshalling (protobuf + HTTP/2). We implement an mRPC variant that applies encoding (decoding) code generated by the `protobuf` compiler and HTTP/2 framing for inter-host mRPC service communication.

We conduct the same large RPC goodput experiment in §7.1. The results are presented in Figure 10. We find that mRPC achieves performance comparable to gRPC after switching to using protobuf + HTTP/2. With full gRPC marshalling, mRPC still performs 2.6× and 3.7× as fast as gRPC + Envoy in terms of goodput and goodput per core. This is because mRPC reduces the number of (un)marshalling steps. The small RPC rate and scalability of mRPC with gRPC marshalling is also shown in Figure 11. Since encoding small RPCs with protobuf is relatively fast, the trend to the rate and scalability is similar to Figure 5a.

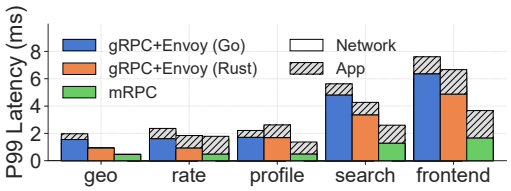


Figure 12: **DeathStarBench**: P99 latency of in-application processing and network processing of microservices, respectively. gRPC with Envoy and mRPC are compared. A null policy is applied for mRPC.

Figure 13: **DeathStarBench**: Mean latency of gRPC without proxy and mRPC.

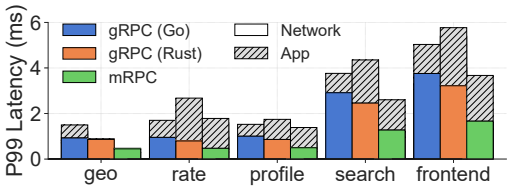


Figure 14: **DeathStarBench**: P99 latency of in-application processing and network processing of microservices, respectively. gRPC without proxy and mRPC are compared.

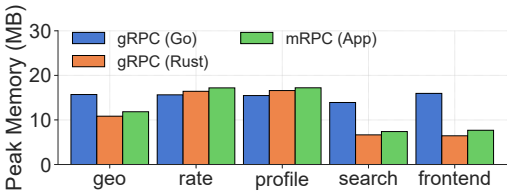


Figure 15: **DeathStarBench**: Peak memory usages of different services. gRPC without proxy and mRPC are compared.

A.2 Extended Evaluation for DeathStarBench

We report the P99 latency of DeathStarBench in Figure 12, comparing gRPC with Envoy and mRPC. The result is similar to the comparison of median latency in §7.4. mRPC speeds up gRPC+Envoy by 2.1× in terms of end-to-end P99 tail latency.

We also evaluate gRPC without proxy and mRPC without any policy enforced. Figure 13 and Figure 14 show the results for mean latency and P99 tail latency. We observe that mRPC speeds up gRPC by 1.7× and 1.6×, in terms of mean latency and P99 tail latency. Communication costs are substantial in the DeathStarBench applications, and thus reducing the communication latency can improve end-to-end application performance. This is consistent with the original

DeathStarBench paper's observation [23].

We further compare the memory usage of gRPC and mRPC. The peak memory consumption of gRPC and mRPC in DeathStarBench applications is illustrated in Figure 15. For mRPC, we report the user application side memory usage, which also includes all the memory pages shared with the mRPC service. We observe that mRPC does not incur notable memory overhead compared to gRPC. On the other hand, we find a small and constant memory footprint of mRPC service across all machines at around 9 MB.

Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory

Chenxi Wang[†*] Yifan Qiao[†*] Haoran Ma[†] Shi Liu[†] Yiying Zhang[‡]
Wenguang Chen[§] Ravi Netravali[♯] Miryung Kim[†] Guoqing Harry Xu[†]
UCLA[†] *UCSD*[‡] *Tsinghua University*[§] *Princeton University*[♯]

Abstract

Remote memory techniques for datacenter applications have recently gained a great deal of popularity. Existing remote memory techniques focus on the efficiency of a single application setting only. However, when multiple applications co-run on a remote-memory system, significant interference could occur, resulting in unexpected slowdowns even if the same amounts of physical resources are granted to each application. This slowdown stems from massive sharing in applications' swap data paths. Canvas is a redesigned swap system that fully isolates swap paths for remote-memory applications. Canvas allows each application to possess its dedicated swap partition, swap cache, prefetcher, and RDMA bandwidth. Swap isolation lays a foundation for adaptive optimization techniques based on each application's own access patterns and needs. We develop three such techniques: (1) adaptive swap entry allocation, (2) semantics-aware prefetching, and (3) two-dimensional RDMA scheduling. A thorough evaluation with a set of widely-deployed applications demonstrates that Canvas minimizes performance variation and dramatically reduces performance degradation.

1 Introduction

Techniques enabling datacenter applications to use far memory [36, 39, 8, 62, 73, 91, 104, 90, 19] have gained traction due to their potential to break servers' memory capacity wall, thereby improving performance and resource utilization. Existing far-memory techniques can be roughly classified into two categories: (1) clean-slate techniques [90, 19] that provide new primitives for developers to manage remote memory, and (2) swap-based techniques [39, 91, 8, 104, 2] that piggyback on existing swap mechanisms in the OS kernel. Clean-slate techniques provide greater efficiency by enabling user-space far memory accesses, while swap-based techniques offer transparency, allowing legacy code to run *as is* on a far-memory system. This paper focuses on swap mechanisms as they are more practical and easier to adopt.

A typical swap system in the OS uses a *swap partition* and *swap cache* for applications to swap data between memory and external storage. The swap partition is a storage-backed swap space. The swap cache is an intermediate buffer between the *local memory* and storage—it caches *unmapped pages* that were just swapped in or are about to be swapped out. Upon a page fault, the OS looks up the swap cache; a cache miss would trigger a *demand swap* and a number of *prefetching swaps*. Swaps are served by RDMA and all fetched pages are initially placed in the swap cache. The demand page is then mapped to a virtual page and moved out of the swap cache, completing the fault handling process.

Problems. Current swap systems run multiple applications over shared swap resources (*i.e.*, swap partition, RDMA, *etc.*). This design works for *disk-based swapping* where disk access is slow—each application can allow only a tiny number of pages to be swapped to maintain an acceptable overhead. This assumption, however, no longer holds under far memory because an application can place more data in far memory than local memory and yet still be efficient, thanks to RDMA's low latency and high bandwidth.

As such, applications have orders-of-magnitude more swap requests under far memory than disks. Millions of swap requests from different applications go through the same shared data path in a short period of time, leading to *severe performance interference*. Our experiments show that, with the same amounts of CPU and local-memory resources, co-running applications leads up to a 6× slowdown, an overhead unacceptable for any real-world deployment.

State of the Art. Interference is a known problem in datacenter applications and a large body of work exists on isolation of CPU [64, 16, 25], I/O [40, 96], network bandwidth [13, 37, 94, 87, 77, 53] and processing [59]. Most of these techniques build on Linux's `cgroup` mechanism, which focuses on isolation of traditional resources such as CPU and memory, *not* swap resources such as remote memory usage and RDMA. Prior swap optimizations such as Infiniswap [39] and Fastswap [8] focus on reducing remote access latency, overlooking the impact of swap interference in realistic settings. Justitia [113] isolates RDMA bandwidth between applications, but does not eliminate other types of interference such as locking and swap cache usage.

Contribution #1: Interference Study (§3). We conducted a systematic study with a set of widely-deployed applications on Linux 5.5, the latest kernel version compatible with Mellanox's latest driver (4.9-3.1.5.0) for our InfiniBand card. Our results reveal three major performance problems:

- **Severe lock contention:** Since all applications share a single swap partition, extensive locking is needed for swap entry allocation (needed by every swap-out), reduc-

∗ Contributed equally.

ing throughput and precluding full utilization of RDMA's bandwidth. Our experience shows that in windows of frequent remote accesses, applications can spend **70%** of the windows' time on swap entry allocation.

- **Uncontrolled use of swap resources (*e.g.*, RDMA):** The use of the shared RDMA bandwidth is often dominated by the pages fetched for applications with many threads simultaneously performing frequent remote accesses. For example, aggressively (pre)fetching pages to fulfill one application's needs can disproportionally reduce other applications' bandwidth usage. Further, even within one application, prefetching competes for resources with demand swaps, leading to either prolonged fault handling or delayed prefetching that fails to bring back pages in time.
- **Reduced prefetching effectiveness:** Applications use the same prefetcher, prefetching data based on *low-level (sequential or strided) access patterns* across applications. However, modern applications exhibit far more diverse access patterns, making it hard for prefetching to be effective across the board. For example, co-running Spark and native applications reduces Leap [73]'s prefetching contribution by **3.19**×.

These results highlight two main problems. First, interference is caused by sharing a combination of swap resources including the swap partition/cache, and RDMA (bandwidth and SRAM on RNIC). Although recent kernel versions added support [47] for charging prefetched pages into `cgroup`, resolving interference requires a *holistic* approach that can isolate all these resources. Furthermore, interference stems not only from resource racing, but also from fundamental limitations with the current design of the swap system. For instance, reducing interference between prefetching and demand swapping requires understanding whether a prefetching request can come back in time. If not, it should be dropped to give resources to demand requests, which are on the critical path. This, in turn, requires a redesign of the kernel's fault handling logic.

Second, cloud applications exhibit highly diverse behaviors and resource profiles. For example, applications with a great number of threads are more sensitive to locking than single-threaded applications. Furthermore, managed applications such as Spark often make heavy use of reference-based data structures while native applications are often dominated by large arrays. The *application-agnostic nature* of the swap system makes it hard for a one-size-fits-all policy (*e.g.*, a global prefetcher) to work well for diverse applications. Effective per-application policies dictates (1) holistic swap isolation and (2) understanding application semantics, which is currently inaccessible in the kernel.

Contribution #2: Holistic Swap Isolation (§4). To solve the first problem, we develop Canvas, a *fully-isolated* swap system, which enables each application to have its dedicated swap partition, swap cache, and RDMA usage. In doing so, Canvas can charge each application's `cgroup` for the usage of all kinds of swap resources, preventing certain applications from aggressively invading others' resources.

Contribution #3: Isolation-Enabled Adaptive Optimizations (§5). To solve the second problem, we develop a set of adaptive optimizations that can tailor their policies and strategies to application-specific swap behaviors and resource needs. Our adaptive optimizations bring a *further boost* on top of the isolation-provided benefits, making co-running applications even *outperform* their individual runs.

(1) Adaptive Swap Entry Allocation (§5.1) Separating swap partitions reduces lock contention at swap entry allocations to a certain degree, but the contention can still be heavy for multi-threaded applications. For example, Spark creates many threads to fully utilize cores and these threads need synchronizations before obtaining swap entries. The synchronization overhead increases dramatically with the number of cores (§6.4.1), creating a scalability bottleneck. We develop an adaptive swap entry allocator that dynamically balances between the degree of lock contention (*i.e.*, time) and the amount of swap space needed (*i.e.*, space) based on each application's memory behaviors.

(2) Adaptive Two-tier Prefetching (§5.2) Current kernel prefetchers build on low-level access patterns (*e.g.*, sequential or strided). Although such patterns are useful for applications with large array usages, many cloud applications are written in high-level, managed languages such as Java or Python; their accesses come from multiple threads or exhibit pointer-chasing behavior as opposed to sequential or strided patterns. As effective prefetching is paramount to remote-memory performance, Canvas employs a two-tier prefetching design. Our *kernel-tier prefetcher* prefetches data for each application into its private swap cache based on low-level patterns. Once this prefetcher cannot effectively prefetch data, Canvas adaptively forwards the faulty address up to the *application tier* via a modified `userfaultfd` interface, enabling customized prefetching logic at the level of reference-based or thread-based access patterns.

(3) Adaptive RDMA Scheduling (§5.3) Isolating RDMA bandwidth alone for each application is insufficient. As there could be many more *prefetching* requests than *demand swap requests*, naïvely sending all to RDMA delays demand requests, increasing fault-handling latency. On the other hand, naïvely delaying prefetching requests (as in FastSwap [8]) reduces their *timeliness*, making prefetched pages useless. We built a *two-dimensional* RDMA scheduler, which schedules packets not only between applications but also between prefetching and demand requests for each application.

Results. Our evaluation (§6) with a set of 14 widely-deployed applications (including Spark [109], Cassandra [10], Neo4j [79], Memcached [4], XGBoost [23, 22], Snappy [38], *etc.*) demonstrates that Canvas improves the overall application performance by up to **6.2**× (average **3.5**×) and reduces applications' performance variation (*i.e.*,

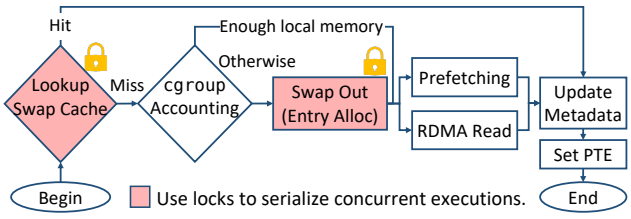


Figure 1: The kernel's remote-access data path.

standard deviation) by **7**×, from an overall of **1.72** to **0.23**. Canvas improves the overall RDMA bandwidth utilization by **2.8**× for co-run applications. Canvas is available at https://github.com/uclasystem/canvas.

2 Background

This section presents the necessary background in Linux 5.5, which is the latest kernel version compatible with Mellanox's latest driver for our InfiniBand adapter.

Figure 1 illustrates the kernel's remote access data path where remote memory is mapped into the host server as a swap partition where applications access remote memory via page faults. The swap partition is split into a set of 4KB *swap entries*, each mapping to an actual remote memory cell and has a unique entry ID. Upon a page fault, the kernel uses the swap entry ID contained in the corresponding page table entry (PTE) to locate the swap entry that stores the page.

The first step in handling the fault is to look up the swap cache, which is a set of radix trees, each containing a number of cached and unmapped pages for a block (*e.g.*, 64MB) of swap entries. These pages were either just swapped in due to demand swapping or prefetching, or are about to be swapped out. If a page can be found there, it gets mapped to a virtual page and removed from the swap cache. Otherwise, the kernel needs to perform a *demand swap-in*.

Before issuing the request, the kernel first does `cgroup` accounting to understand if there is enough physical memory to swap in the page. If there is, the kernel issues an RDMA read request, which is then pushed into RDMA's dispatch queue. As the demand swap occurs, the kernel prefetches a number of pages that will likely be needed in the future. This number depends on the swap history at the past few page faults. For example, if the pages fetched follow a sequential or strided pattern, the kernel will use this pattern to fetch a few more pages. If no pattern is found, the kernel reduces the number of prefetched pages until it stops prefetching completely. Once these demand and prefetched pages arrive, they are placed into the swap cache. Their swap entries in remote memory are then freed.

If `cgroup` accounting deems that local memory is insufficient for the new page, the kernel uses an LRU algorithm to evict pages. Evicting a page *unmaps* it and pushes it into the swap cache. When memory runs low, the kernel releases existing pages from the swap cache to make room for newly fetched pages. Clean pages can be removed right away and dirty pages must be written back. To write back a page, the swap system must first allocate a swap entry using a free-list-based allocation algorithm. Finally, an RDMA write request is generated and the page is written into the entry via RDMA.

In each remote access, extensive locking is needed for swap entry allocation—shared allocation metadata (*e.g.*, free list) must be protected when multiple applications/threads request swap entries simultaneously. Although there are active efforts [48, 46] in the Linux community to optimize swap entry allocation, their performance and scalability is unsatisfactory for cloud workloads (see Appendix B).

3 Motivating Performance Study

To understand the impact of interference, we conducted a study with a set of widely-deployed applications including Apache Spark [109], Neo4j [79], XGBoost [23] (*i.e.*, a popular ML library), Snappy [38] (*i.e.*, Google's fast compressor/decompressor), as well as Memcached [4]. Spark and Neo4j are managed applications running on the JVM, while the other three are native applications. They cover a spectrum of cloud workloads from data storage through analytics to ML. In addition, they include both batch jobs (such as Spark) and latency-sensitive jobs (such as Memcached). Co-running them represents a typical scenario in a modern datacenter where operators fill left-over cores unused by latency-sensitive tasks with batch-processing applications to improve CPU utilization [15]. For example, in a Microsoft Bing cluster, batch jobs are colocated with latency-sensitive services on over 90,000 servers [49]. Google also reported that 60% of machines in their compute cluster co-run at least five jobs [112].

We ran these programs, individually vs. together, on a machine with two Xeon(R) Gold 6252 processors, running Linux 5.5. Another machine with two Xeon(R) CPU E5-2640 v3 processors and 128GB memory was used for remote memory. Each machine was equipped with a 40 Gbps Mellanox ConnectX-3 InfiniBand adapter and interconnected by one Mellanox 100 Gbps InfiniBand switch. Using `cgroup`, the same amounts of CPU and local memory resources were given to each application throughout the experiments. RDMA bandwidth was *not* saturated for both application individual runs and co-runs. The amount of local memory configured for each application was 25% of its working set.

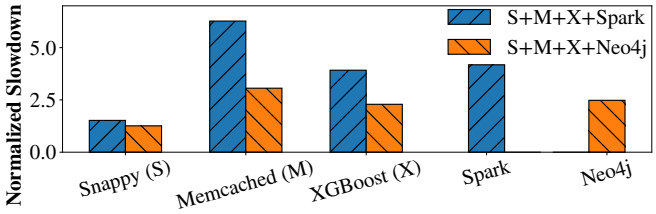


Figure 2: Slowdowns of co-running applications compared to running each individually.

Performance Interference and Degradation. To understand the overall performance degradation and how it changes with different applications, we used two managed applications: Spark and Neo4j. Figure 2 reports each application's performance degradation when co-running with other applications compared to running alone. The blue/orange bars show the slowdowns when the three native applications co-run with Spark/Neo4j. Clearly, co-running applications significantly reduces each application's performance. We observed an overall **3.9/2.2**× slowdown when native applications co-run with Spark/Neo4j. Spark persists a large RDD in memory and keeps swapping in/out different parts of the RDD, while Neo4j is a graph database and holds much of its graph data in local memory and thus does not swap as much as Spark.

Another observation is that the impact of interference differs significantly for different applications. Applications that generate high swap throughputs aggressively invade swap and RDMA resources of other applications. In our experiments, Memcached, XGBoost, and Spark all need frequent swaps. However, Spark runs many more threads (>90 application and runtime threads) than Memcached (4) and XGBoost (16), resulting in a much higher swap throughput. As such, Spark takes disproportionally more resources, leading to severe degradation for Memcached and XGBoost.

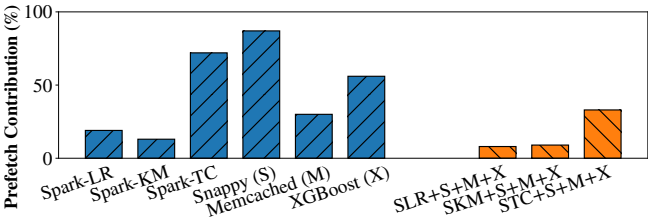
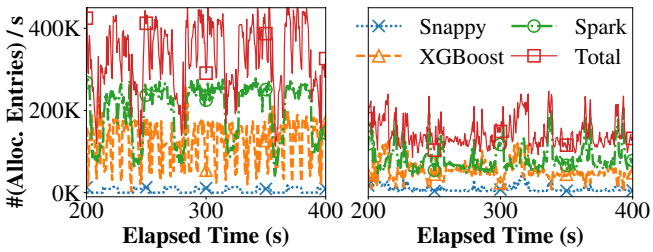


Figure 3: Prefetching contribution of Leap: the percentage of page faults served by Leap-prefetched pages (%).

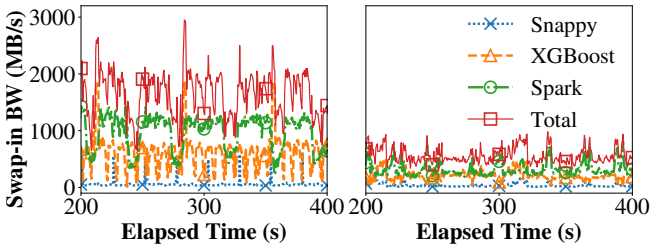
Reduced Prefetching Effectiveness. Sharing the same prefetching policy reduces the prefetching effectiveness when multiple applications co-run. Figure 3 reports *prefetching contribution*—the percentage of page faults served by prefetched pages—the higher the better; if a prefetched page is never used, prefetching it would only incur overhead. We used Leap [73] as our prefetcher. The left six bars report such percentages for the applications running individually. When applications co-run, the rightmost three bars report the average percentages across applications. As shown, co-running dramatically reduces the contribution.

Note that Leap [73] uses a majority-vote algorithm to identify patterns across multiple applications. However, when applications that exhibit drastically different behaviors co-run, Leap cannot adapt its prefetching mechanism and policy to each application. Furthermore, Leap is an aggressive prefetcher—even if Leap does not find any pattern, it always prefetches a number of contiguous pages. However, aggressive prefetching for applications such as Spark with garbage collection (GC) is ineffective—*e.g.*, prefetching for a GC thread has zero benefit and only incurs overhead. Detailed evaluation of prefetching can be found in §6.4.



(a) Running individually. (b) Co-running.

Figure 4: Swap entry allocation throughput when applications run individually (a) and together (b).



(a) Running individually. (b) Co-running.

Figure 5: RDMA swap-in bandwidth when applications run individually (a) and together (b).

Lock Contention. We observed severe lock contention in the swap system when applications co-run, particularly at swap entry allocation associated with each swap-out.

We experimented with Spark (Logistic Regression), XGBoost, and Snappy. Our results show that in windows of frequent remote accesses, co-running applications can spend up to **70%** of the window time on obtaining swap entries. Lock contention leads to significantly reduced swap-entry allocation throughput, reported in Figure 4. The total lines in Figure 4(a) and (b) show the total throughput (*i.e.*, the sum of each application's allocation throughput). The co-running throughput (b) is drastically reduced compared to the individual run's throughput (a) (*i.e.*, ∼450Kps to ∼200Kps).

Reduced RDMA Utilization. Figure 5 compares the RDMA read bandwidth (for swap-ins) when applications run individually and together. Similarly, the total line represents the sum of each application's RDMA bandwidth. The total RDMA utilization is constantly below ∼1000MBps in Figure 5(b), which is **3.28**× lower than that in Figure 5(a) due to various issues (*e.g.*, locking, reduced prefetching, *etc.*). The RDMA write bandwidth degrades by an overall of **2.80**×.

Demand v.s. Prefetching Interference. Optimizations such as Fastswap [8] improve swap performance by dividing the RDMA queue pairs (QP) into sync and async. The high-priority synchronous QP is used for demand swaps, while the low priority async QP is used for prefetching requests.

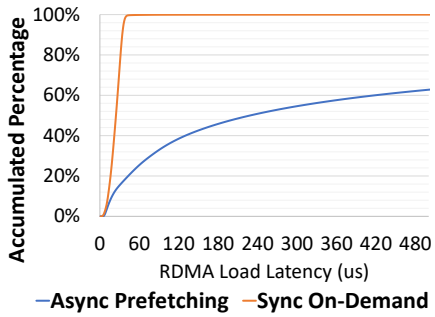


Figure 6: Latency of prefetching and on-demand swapping.

This separation reduces head-of-line blocking incurred by prefetching. However, when applications co-run, this design adds a delay for prefetching. Figure 6 depicts the CDF of the latency of RDMA packets from demand and prefetching requests, when the four applications co-run on Leap. As shown, 99% of the on-demand requests are served within 40μs. However, the latency of 36.9% of prefetching requests is longer than 512μs and it can reach up to 52ms! Long latency renders prefetched pages useless because prefetching is meant to load pages to be used soon. Our profiling shows that among the prefetched pages that are actually accessed by the application, *90% are accessed within 70*μs, indicating that ∼70% of the pages prefetched return too late. A late prefetch of a page would subsequently block a demand request of the page when it is accessed by the application. This problem motivates our two-dimensional RDMA scheduling (§5.3).

Takeaway. The root cause of performance degradation is that multiple applications, whose resource needs and swap behaviors are widely apart, all run on a global swap system with the same allocator and prefetcher. Table 1 summarizes these problems, their performance impact, and our solutions.

4 Swap System Isolation

Canvas extends `cgroup` for users to specify size constraints for swap partition, swap cache, and RDMA bandwidth. We discuss the kernel support to enforce these new constraints, laying a foundation for adaptive optimizations in §5.

Swap Partition Isolation. In Linux, remote memory is managed via a swap partition interface, shared by all applications. If there are multiple available swap partitions, they are used in a *sequential manner* according to their priorities. As a result, data of different applications are mixed and stored in arbitrary locations.

Canvas separates remote memory of each `cgroup` to isolate capacity and performance. The user creates a `cgroup` to set a size limit of remote memory for an application. Canvas allocates remote memory in a demand-driven manner—upon a pressure in local memory, Canvas allocates remote memory and registers it as a RDMA buffer. Canvas enables per-`cgroup` swap partitions by creating a swap partition interface and attaching it to each `cgroup`. For each `cgroup`, a separate swap-entry manager is used for allocating and freeing swap entries. Swap entry allocation can now be charged to the `cgroup`, which controls how much remote memory each application can use. Our adaptive swap entry allocation algorithm is discussed in §5.1.

Canvas explicitly enables a private swap cache for each `cgroup` (a default value of 32MB), whose size is charged to the *memory budget* specified in the `cgroup`. As a result, the size of an application's swap cache changes in response to its own memory usage, without affecting other applications.

For each demand swap-in, Canvas first checks the `mapcount` of the page, which indicates how many processes this page has been mapped to before. If the page belongs only to one process, it is placed in its private swap cache. Otherwise, it has to be placed in a global swap cache (discussed shortly). To release pages (*e.g.*, when the application's working set increases, pushing the boundary of the swap cache), Canvas scans the swap cache's page list, releasing a batch of pages to shrink the cache.

RDMA Bandwidth Isolation. For each `cgroup`, Canvas isolates RDMA bandwidth with a set of *virtual* RDMA queue pairs (VQPs) and a centralized packet scheduler. Users can set the swap-in/swap-out RDMA bandwidth of a `cgroup` with our extended interface. Our RDMA scheduler works in two dimensions. The *first dimension* schedules packets across applications, while the *second dimension* schedules on a per-application basis—each `cgroup` has its *sub-scheduler* that schedules packets that belong to the `cgroup` between demand swapping and prefetching.

VQPs are high-level interfaces, implemented with lock-free linked lists. Each `cgroup` pushes its requests to the head of its VQP, while the scheduler pops requests from their tails. At the low level, our scheduler maintains three physical queue pairs (PQP) per core, for *demand swap-in*, *prefetching*, and *swap-out*, respectively. The scheduler polls all VQPs and forwards packets to the corresponding PQPs, using a *two-dimensional* scheduling algorithm (see §5.3).

Handling of Shared Pages. Processes can share pages due to shared libraries or memory regions. These pages cannot go to any private swap cache. Canvas maintains a global swap partition and cache for shared pages. When a page is evicted and ummapped, Canvas checks its `mapcount` and adds it to the global swap cache if the page is shared between different processes. All pages in the global swap cache will be eventually swapped out to the global partition using the original lock-based allocation algorithm. Conversely, pages swapped in (and prefetched) from the global swap partition are all placed into the global swap cache. For typical cloud applications such as Spark, Cassandra and Neo4j, the number of shared pages is much smaller than process-private pages, using locks in a normal way would not incur a large overhead. We cannot charge applications' `cgroups` for pages in the global swap cache, because which process(es) share these pages is unknown before they get mapped into pro-

Problem Description	Performance Impact	Canvas's Solution
Unlimited use of swap and RDMA resources	Apps generating higher swap thruput use disproportionately more resources	Holistic isolation of swap system RDMA isolation and scheduling (§4, §5.3)
Lock conten. at swap entry alloc.	Reduced swap-out thruput	(1) Swap parti. isolation (§4); (2) adaptive entry alloc. (§5.1)
Single low-level prefetcher	Increased fault-handling latency	Two-tier adaptive prefetching (§5.2)
prefetching v.s. demand interfere	Increased fault-handling latency	Two-dimensional RDMA scheduling (§5.3)

Table 1: Summary of major issues and Canvas's solution.

cesses' address spaces. Canvas allows users to create a special `cgroup`, named `cgroup-shared`, to limit the size of the global swap cache/partition.

One limitation of our `cgroup`-based approach is that `cgroup` can only partition resources statically while applications' resource usage may change from time to time and static partitioning could lead to resource underutilization. However, the focus of this paper is to ensure isolation and future work could incorporate max-min fair allocation to improve resource utilization.

5 Isolation-Enabled Swap Optimizations

On top of the isolated swap system, we develop three optimizations, which dynamically adapt their strategies to each application's resource patterns and semantics.

5.1 Adaptive Swap Entry Allocation

As discussed in §3, swap entry allocation suffers from severe lock contention under frequent remote accesses—allocation is needed at every swap-out. To further motivate, we use a simple experiment by running Memcached alone on remote memory with different core numbers. As the number of cores increases, the average entry allocation time grows super-linearly—it grows from 10μs under 16 cores quickly to 130μs under 48 cores due to increased lock contention (see Figure 16). Creating a per-application swap partition mitigates the problem to a certain degree. However, applications like Spark run more than 90 threads; frequent swaps in these threads can still incur significant locking overhead.

To further reduce contention, we develop a novel swap entry allocator that adapts allocation strategies in response to each application's own memory access/usage. Our first idea is to enable a *one-to-one* mapping between pages and swap entries. At the first time a page is swapped out, we allocate a new swap entry using the original (lock-protected) algorithm. Once the entry is allocated, Canvas writes the entry ID into the page metadata (*i.e.*, `struct page`). This ID remains on the page throughout its life span. As a result, subsequent swap-outs of the page can write data directly into the entry corresponding to this ID. We pay the locking overhead *only once* for each page at its first swap-out.

This approach requires a swap entry to be reserved for each page. For example, if the local memory size is S and the remote memory allocation is 3S, with one-to-one mapping the remote memory allocation would be 4S (*i.e.*, each page residing in local memory also has a remote page, resulting in a 33% overhead). However, this overhead may not be necessary. For example, modern applications exhibit strong epochal behaviors. Under the original allocator, swap entries for pages accessed in one epoch can be reused for those in another epoch. Under this approach, however, all pages in all epochs must have their dedicated swap entries throughout the execution, which can lead to an order-of-magnitude increase in remote memory usage.

Our key insight is: we should trade off *space for time* if an application has much available swap space, but *time for space* when its space limit is about to be reached. As such, when the remote memory usage is about to reach the limit specified in `cgroup` (*i.e.*, 75% in our experiments), Canvas starts removing reservations to save space. The next question is which pages we should consider first as our candidates for reservation removal. Our idea is that we should first consider "hot pages" that always stay in local memory and are rarely swapped. This is because hot pages (*i.e.*, data on such pages are frequently accessed) are likely to stay in local memory for a long time; hence, locking overhead is less relevant for them. On the contrary, "cold" pages whose accesses are *spotty* are more likely to be swapped in/out and hence swap efficiency is critical. Here "hot" and "cold" pages are relatively defined as they are specific to execution stages—a cold page swapped out in a previous stage can be swapped in and become hot in a new stage.

To this end, we develop an *adaptive allocator*. Canvas starts an execution by reserving swap entries for *all* pages to minimize lock contention. Reservation removal begins when remote-memory pressure is detected. Canvas adaptively removes reservations for hot pages. We detect hot pages *for each application* by periodically scanning the application's *LRU active list*—pages recently accessed are close to the head of the active list. Each scan identifies a set of pages from the head of the list; a page is considered "hot" if it appears in a consecutive number of sets recently identified.

Removing the reservation for a hot page can be done by (1) removing the entry ID from the page metadata and (2) freeing its reserved swap entry in remote memory, adding the entry back to the free list. Once a hot page becomes cold and gets evicted, it does not have a reservation any more, and hence, it goes through the original (lock-protected) allocation algorithm to obtain an entry. In this case, the page

receives a new swap entry and remembers this new ID in its metadata.

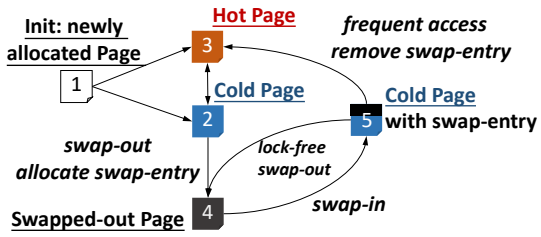


Figure 7: FSM describing our page management when remote-memory pressure is detected.

Figure 7 shows the page state machine, which describes the page handling logic. A cold page (to be evicted) can be in one of the two states: state 2 and state 5. A page comes to state 2 if it is (1) a brand new page that has never been swapped out or (2) previously a hot page but has not been accessed for long. Once it reaches state 2, the page does not have a reserved swap entry ID and hence, swapping out this page goes through the normal allocation path. In the case of swap-in (state 5), the swap entry ID is already remembered on the page. The next swap-out will directly use this entry and be lock-free. If the page becomes hot (from state 5 to 3), Canvas removes the entry ID and releases the entry reservation. The entry is then added back to the free list.

Performance Analysis. To understand the performance of the adaptive entry allocation algorithm, let us consider the following two scenarios. In the first scenario, the application performs uniformly random accesses. As a result, Canvas cannot clearly distinguish hot/cold pages, and thus randomly cancels their reservations. However, due to the random process, when a page is swapped out, it has a certain probability of still possessing a reserved swap entry (depending on the ratio of remaining reservations) and hence Canvas can still improve the allocation performance.

In the second scenario, the application follows a repetitive pattern of accessing a page a few times (making it hot) and then moving on to accessing another page; it will not come back to the page in a long while. Under our allocation algorithm, every page will be identified as a hot page, leading to the cancellation of its reservation. However, each page will be swapped out when it is cold enough; at each swap-out, the page has to go through the original allocation algorithm. This is the worst-case scenario, and even in this case, Canvas has the same (worst-case) performance as the original Linux allocator, which allocates an entry at each swap-out.

Some of the recent patches submitted to the Linux community also attempt to reduce lock contention for swap entry allocation. A detailed description of how Canvas differs from these patches can be found in Appendix B.

5.2 Two-tier Adaptive Prefetching

Problems with Current Prefetchers. Current prefetchers all focus on low-level (streaming or strided) access patterns. While such patterns exist widely in native array-based programs, applications written in high-level languages such as Python and Java are dominated by reference-based data structures—operations over such data structures involve large amounts of pointer chasing, making it hard for current prefetchers to identify clear patterns.

Furthermore, cloud applications such as Spark are heavily multi-threaded. Modern language runtimes, such as the JVM, run an additional set of auxiliary threads, *e.g.*, for GC or JIT compilation. How these user-level threads map to kernel threads is often implemented differently in different runtimes. Consequently, kernel prefetchers such as Leap [73] cannot distinguish patterns from different threads.

To develop an adaptive prefetcher, Canvas employs a two-tier design, illustrated in Figure 8. At the low (kernel) tier, Canvas uses an existing kernel prefetcher that prefetches data for each application into its own private swap cache (unless data comes from the global swap partition). A kernel prefetcher is extremely efficient and can already cover a range of (array-based) applications. For applications whose accesses are too complex for the kernel prefetcher to handle, we forward the addresses up to the application level, letting the application/runtime analyze semantic access patterns at the level of threads, references, arrays, *etc.*

Prefetching Logic. In Canvas, we adopt the sync/async separation design in Fastswap [8], which prevents head-of-line blocking. As stated earlier, we use three PQPs per core, one for swap-out, one for (sync) demand swap-in, and one for (async) prefetching. Canvas polls for completions of critical (demand) operations, while configuring *interrupt completions* for asynchronous prefetches.

Canvas determines whether to use an application-tier prefetcher based on *how successful kernel-tier prefetching is*. If the number of pages prefetched for an application is lower than a threshold at the most recent N (=3 in our evaluation) faults consecutively, Canvas starts forwarding the faulting addresses up to the application-tier prefetcher (discussed shortly) although the kernel-tier prefetcher is still used as the first-line prefetcher.

Canvas stops forwarding whenever the kernel-tier prefetcher becomes effective again. Our key insight is: the kernel-tier prefetcher is efficient without needing additional compute resources (as it uses the same core as the faulting thread), while the application-tier prefetcher needs extra compute resources to run. As such, we disable application-tier prefetchers as long as the kernel-tier prefetcher is effective. To pass a faulting address to the application, we modify the kernel's `userfaultfd` interface, allowing applications to handle faults at the user space. Our modification makes the kernel forward the faulting address only if the kernel's prefetcher continuously fails to prefetch pages.

Runtime Support for Application-tier Prefetching. A major challenge is how to develop application-tier prefetchers. On the one hand, application-tier prefetchers should conduct

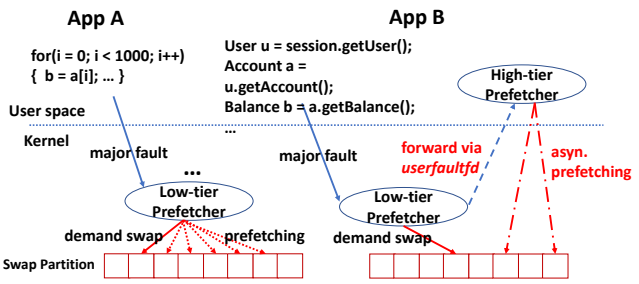


Figure 8: Canvas's two-tier prefetcher: App A is an array-based program while B is a modern web application that uses reference-based data structures. The low-tier prefetcher successfully prefetches pages for A, but not for B. Hence, Canvas forwards the addresses up to B's high-tier prefetcher.

prefetching based on *application semantics*, of which the kernel is unaware. On the other hand, application developers may not be familiar with a low-level activity like prefetching; understanding memory access patterns and developing prefetchers can be a daunting task for them.

Our insight is: applications that benefit from application-tier prefetching are mostly written in high-level languages and run on a managed runtime such as the JVM. Inspired by previous work on using language runtime to solve memory efficiency problems for data analytics applications [81, 78, 82, 80, 72], Canvas currently supports application-tier prefetching for the JVM as a platform. However its support could be easily extended to other managed runtimes for high-level languages like Go and C#. Leveraging language runtime solves both problems discussed above—it has access to semantic information such as how objects are connected and the number of application threads; furthermore, the burden of developing an application-tier prefetcher is shifted from application developers to runtime developers. Thus, it is not necessary to supply a custom application-tier prefetcher per application, but define it once for each language runtime.

In this work, we develop an application-tier prefetcher in Oracle's OpenJDK as a proof-of-concept. It works for all (Java, Scala, Python, *etc.*) programs that run on the JVM. Our JVM-based prefetcher considers two *semantic patterns*: (1) *reference-based* (*i.e.*, accessing an object brings in pages containing objects referenced by this object) and (2) *thread-based* (*i.e.*, accesses from different application threads are separately analyzed to find patterns).

For (1), we modify the JVM to add support that can quickly find, from a faulting address, *the object* in which the address falls. We use write barrier, a piece of code instrumented by the JVM at each object field write, as well as the garbage collector to record references between pages. For example, for each write of the form `a.f=b`, if the objects referenced by a and b are on different page groups, we record an edge on a *summary graph* where each node represents a consecutive group of pages and each edge represents references between groups. During prefetching, we traverse the graph from the node that represents the accessed page and prefetch pages that can be reached within 3 hops. The traversal does not follow cycles and its overhead is negligible. This approach is suitable for applications that store a large amount of data in memory, such as Spark and Cassandra.

For (2), we leverage the JVM's user-kernel thread map. For each faulting address, Canvas additionally forwards the thread information (*i.e.*, pid) to the JVM, which consults the map to filter out non-application (*e.g.*, GC, compilation, *etc.*) threads and segregate addresses based on Java threads (as opposed to kernel threads). Segregated addresses allow us to analyze (sequential/strided) patterns on a per-thread basis (using Leap's majority-vote algorithm [73]). Once patterns are found, the prefetcher sends the prefetching requests to the kernel via `async_prefetch`.

For native programs that directly use kernel threads (*e.g.*, `pthread`), the thread information is straightforward and immediately visible to Canvas. We can easily segregate addresses accessed from different threads and analyze patterns based upon addresses from each individual thread.

Policy. To improve effectiveness, the JVM uses a search tree to record information about large arrays. Upon the allocation of an array whose size exceeds a threshold (*i.e.*, 1MB in our experiments), the JVM records its starting address and size into the tree. The JVM runs a daemon prefetching thread. Once receiving a sequence of faulting addresses, we determine which semantic pattern to use based on *how many application threads are running* and *whether the faulting addresses fall into a large array*. If there are many threads and the faulting addresses fall into arrays, the JVM uses (2) to find per-thread patterns. If either condition does not hold, the JVM uses (1) to prefetch based on references. For native applications, we only enable (2), as we observed that our native programs do not use many deep data structures.

5.3 Two-Dimensional RDMA Scheduling

To provide predictable performance for applications sharing RDMA resources, our RDMA scheduling algorithm should provide four properties: (1) weighted fair bandwidth sharing [18, 30] across applications; (2) high overall utilization; (3) treating demand and prefetching requests with different priorities; and (4) timely handling of prefetching requests.

Canvas performs two-dimensional scheduling by extending existing techniques. Canvas uses max-min fair scheduling to assign bandwidth across applications, and priority-based scheduling with *timeliness* to schedule prefetching and demand requests within each application. Although these scheduling techniques are not new themselves, Canvas combines them in a unique way to solve the interference problem. Canvas maintains three PQPs on each core, respectively, for swap-outs, demand swap-ins, and prefetching swap-ins. Swap-outs are only subject to fair scheduling while swap-ins are subject to both fair and priority-based scheduling.

Vertical: Fair Scheduling. Under max-min fairness, each application receives a fair share of bandwidth. If there is extra bandwidth, we give it to the applications in the reverse order of their bandwidth demand until bandwidth is saturated. The high overall utilization of bandwidth is achieved by redistributing unconsumed bandwidth proportionally to the weights of unsatisfied applications. Canvas implements weighted fair queuing with virtual clock [84, 30, 110].

Horizontal: Priority Scheduling with Timeliness. Within each `cgroup`, Canvas schedules demand requests with a higher priority than prefetching requests. However, this could lead to long latency for prefetching requests. To bound the latency of prefetching, our scheduler employs a history-based heuristic algorithm to identify and drop outdated prefetching requests. In particular, Canvas maintains the *timeliness distribution* of prefetched pages per `cgroup`. Timeliness is a metric that measures the time between a page being prefetched and accessed. We attach a timestamp to each request when pushing it into a VQP. The scheduler maintains packets statistics on-the-fly to estimate the round-trip latency and arrival time of each prefetching request. Requests are dropped if the estimated arrival time exceeds the estimated timeliness threshold.

Special care must be taken to drop prefetching requests. Before issuing a prefetching request, the kernel creates a page in the swap cache and sets up its corresponding PTE. The page is left in a *locked* state until its data comes back. However, a thread that accesses an address falling into the page may find this locked page in the swap cache and block on it. Dropping prefetching requests may cause the thread to hang. To solve the problem, we detect threads that block on prefetching requests for too long and generate new *demand requests* for them.

We rely on a per-entry timestamp to efficiently detect threads that block on prefetching requests. In Canvas, we attach a timestamp field to the swap entry metadata. Canvas's scheduler records the timestamp every time it enqueues a prefetching request into VQP. If another thread faults on the same page later, it will retrieve the same swap entry from the PTE. If the swap entry contains a timestamp, the faulting thread knows that a prefetching request has already been issued. Next, the faulting thread calculates the time elapsed since the timestamp, and compares it with a timeout threshold (maintained by the RDMA scheduler based on page-fetching latencies). If it exceeds the timeout threshold, the faulting thread drops the prefetching request. The drop operation is elaborated below:

Before issuing each (demand or prefetching) request, the kernel first allocates a physical page in the swap cache and locks the page until the request returns. Upon the return of the data, the data is written into the page; the page is unlocked and mapped into the page table. In order to safely drop a request, we add another field *valid* in the swap entry metadata, indicating whether the prefetching request on the go is valid. Once a faulting thread identifies a delayed prefetching request (by using the timestamp as discussed above), it sets the *valid* field in the swap entry to `false` and then creates a new physical page in the swap cache. The thread goes ahead and issues another (demand) I/O request based on this new page. When the delayed prefetching request returns, it checks the *valid* field and discards itself once it sees the false value. The field is then set back to `true`.

When a demand request is issued, Canvas clears the timestamp field in its corresponding swap entry. If a thread faults on the same page, it will block on the request instead of issuing a new one due to the empty timestamp (indicating that the request on the go is a demand one).

6 Evaluation

It took us 17 months to implement Canvas in Linux 5.5. The application-tier prefetcher was implemented in OpenJDK 12.

Application	Workload	Dataset	Size / ($\|E\|$, $\|V\|$)
Managed			
Cassandra	5M read, 5M insert	YCSB[26]	10M records
Neo4j	PageRank	Baidu[5]	(17M, 2M)
Spark	PageRank (**SPR**)	Wikipedia[5]	(57M, 1.5M)
	KMeans (**SKM**)	Wikipedia[5]	188M points
	Logistic Regression (**SLR**)	Wikipedia[5]	188M points
	Skewed Groupby (**SSG**)	synthetic	256K records
	Triangle Counting (**GTC**)	synthetic	(1.5M, 384K)
MLlib	Bayes Classifiers (**MBC**)	KDD [3]	1.5M instances
GraphX	Connected Components (**GCC**)	Wikipedia[5]	(188M, 9M)
	PageRank (**GPR**)	Wikipedia[5]	(188M, 9M)
	Single Src. Shortest Path (**GSP**)	synthetic	2M vertices
Native			
XGBoost	Binary Classification	HIGGS[12]	22M instances
Snappy	Compression	enwik9 [1]	16GB
Memcached	45M gets, 5M sets	YCSB[26]	10M records

Table 2: Programs and their workloads.

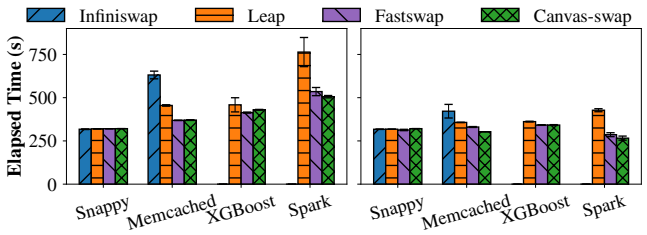
Setup. We included a variety of cloud applications in our experiments, including managed (Java) applications such as Spark [109], Cassandra [10] (a NoSQL database), Neo4j [79] (a graph database), as well as three native applications: XGBoost [23], Snappy [38], and Memcached [4]. Spark, Cassandra, Neo4j, Memcached, and XGBoost are multi-threaded while Snappy is single-threaded. The Spark applications span popular libraries such as GraphX and MLlib.

We co-ran different combinations of programs. The same application in different combinations receives the same amount of local (CPU and memory) resources. To simplify performance analysis, we let each combination of applications co-run contain one managed (Spark, Cassandra, or Neo4j) application and the three native programs, which consume less resources. These experiments were conducted on two machines, one used to execute applications and a second to provide remote memory. The configurations of these machines was reported earlier in §3. We carefully configured Linux with the following configuration to achieve the best performance for Linux: (1) SSD-like swap model, (2) per-VMA prefetching policy, and (3) cluster-based swap en-

try allocation. We disabled hyper-threads, CPU C-states, dynamic CPU frequency scaling, transparent huge pages, and the kernel's mitigation for speculation attacks.

For each combination, we limited the amounts of CPU resources for the managed application, XGBoost, Memcached, and Snappy to be 24, 16, 4, and 1 core(s). For local memory, we used two ratios: 50% and 25%, meaning each application has 50/25% of its working set locally. When using Canvas, we additionally limited the sizes of swap partitions in such a way that for each application the total size of its swap partition and assigned local memory is slightly larger than its working set. In doing so, each application has just enough (local and remote) memory to run and reservation cancellation (§5.1) is triggered in all executions.

The swap cache size for each application starts at 32MB and changes dynamically. The global swap cache size (configured by `cgroup-share`) was also set to 32MB. Canvas uses max-min fair scheduling to assign bandwidth across applications, and their initial weights are proportional to their swap partition assignments. We ran each application 10 times. Their average execution times (with error bars) are reported in all experiments throughput this section.



(a) 25% local memory. (b) 50% local memory.

Figure 9: Performance of different swap systems.

6.1 Basic Swap Systems

We used Fastswap [8] as our underlying swap system, with a small amount of code changes to port Fastswap (originally built against Linux 4.11) to Linux 5.5. We first compared the performance of each individual application running on basic swap systems including Infiniswap [39], Infiniswap with Leap [73], the original Fastswap [8], and Canvas's ported Fastswap (without isolation and optimizations). We could not run LegoOS [91] as it does not support network-related system calls, which are required for applications such as Spark. LegoOS implements swaps with RPCs as opposed to paging, but our idea (*i.e.*, isolation and adaptive swapping) is applicable to this approach as well.

We ran Infiniswap and Leap on Linux 4.4, and Fastswap on Linux 4.11. The results are reported in Figure 9. Infiniswap hung on XGBoost and Spark, and its corresponding bars are thus not reported in Figure 9. Since Canvas-swap was built off Fastswap, they have similar performance.

6.2 Overall Performance

Next, we demonstrate the overall performance when applications co-run together under Canvas. Each experiment ran the same set of three native programs with one managed application: Spark-LR, Spark-KM, Cassandra, or Neo4j. The results for the 25% and 50% local memory configurations are reported in Figure 10(a) and (b), respectively.

The four bars in each group represent an application's performance when running alone on Linux 5.5, co-running with other applications on Linux 5.5, co-running on the original Fastswap, and co-running on Canvas (with all optimizations enabled). Across all experiments, Canvas improves applications' co-run performance by up to **6.2**× (average **3.5**×) and up to **3.8**× (average **1.9**×) under the two memory configurations. Canvas enables Spark and Neo4j to even outperform their individual runs due to the optimizations that could also improve single-application performance.

6.3 Isolation Reduces Degradation and Variation

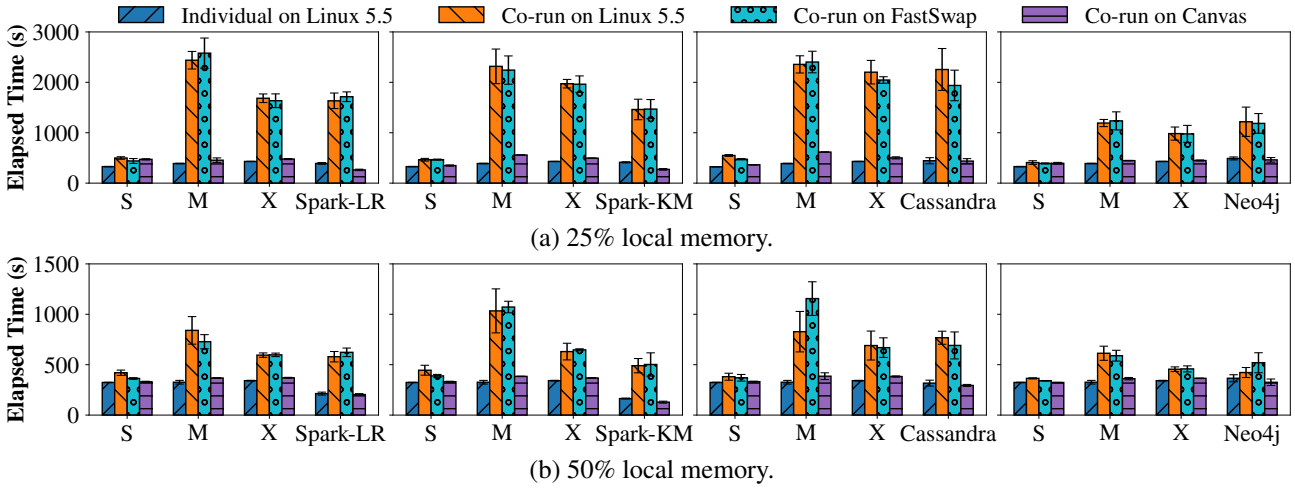
This experiment measures the effectiveness of isolation alone. We used a variant of Canvas with the isolated swap system and RDMA bandwidth (*i.e.*, vertical scheduling between applications) but without our swap-entry optimization, two-tier prefetcher, and horizontal RDMA scheduling.

Degradation Reduction. We ran the same set of experiments under 25% local memory. As shown in Figure 11, isolation reduces the running time by up to 5.2×, with an average of 2.5×. Isolation is particularly useful for applications that do not have many threads but need to frequently access remote memory, such as Memcached, which has 4 threads and cannot compete for resources with managed applications such as Spark and Cassandra, which have more than 90 (application and runtime) threads. As such, its performance is improved by **3.3**× with dedicated swap resources. Isolation improves the average RDMA utilization by 2.8× from 692MB/s to 1908MB/s, making the peak bandwidth reach 4494MB/s.

Table 3: Performance variations of three native applications when co-running with each of the 11 managed applications under 25% local memory (Canvas / Linux 5.5 / Fastswap).

Program	Mean			Min			Max			σ		
Snappy	**1.07**	1.28	1.23	**1.03**	1.10	1.08	**1.23**	1.69	1.46	**0.07**	0.20	0.14
Memcached	**1.45**	3.24	3.76	**1.30**	1.48	2.05	**1.91**	6.05	8.17	**0.20**	1.82	2.14
XGBoost	**1.05**	3.17	2.81	**1.01**	1.38	1.91	**1.13**	6.13	4.76	**0.04**	1.59	1.11
Overall	**1.21**	2.56	2.60	**1.01**	1.10	1.08	**1.91**	6.13	8.17	**0.23**	1.64	1.72

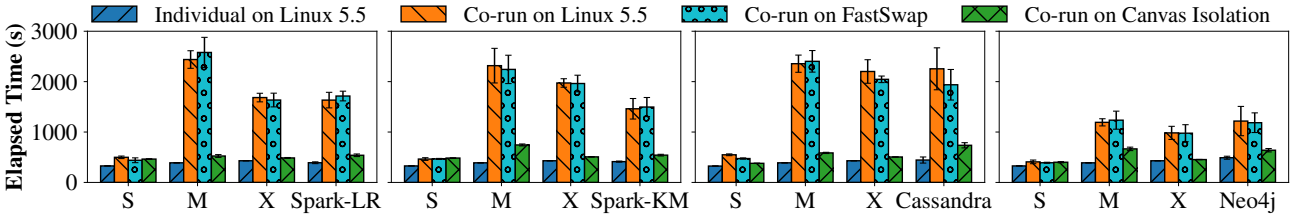
Variation Reduction. One significant impact of interference is performance variation—the same application has drastically different performance when co-running with different applications (as shown in Figure 2). To demonstrate our benefits, we co-ran the three native applications with each of the eleven managed applications listed in Table 2, which cover a wide spectrum of computation and memory access behaviors. Table 3 reports various statistics of their perfor-



(a) 25% local memory.

(b) 50% local memory.

Figure 10: Performance of each program under 25% and 50% local memory when the three native programs, Snappy (S), Memcached (M), and XGBoost (X), co-run with a managed application. Canvas ran with all optimizations enabled.



(a) Co-run with Spark-LR. (b) Co-run with Spark-KM. (c) Co-run with Cassandra. (d) Co-run with Neo4j.

Figure 11: Performance of native applications co-run with different managed applications under 25% local memory; for Canvas, only isolation was enabled (*i.e.*, without adaptive optimizations).

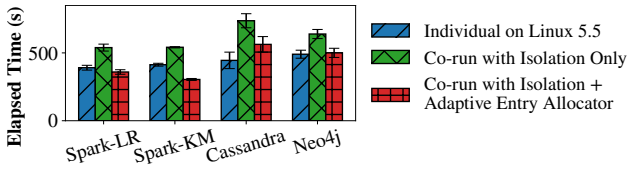


Figure 12: Benefit of adaptive swap entry allocation. Compared are the times of the application running individually on Linux 5.5, co-running on Canvas with adaptive entry allocation disabled, and enabled.

mance including the mean, minimum, maximum, and standard deviation of their slowdowns (compared to their individual runs). Clearly, the performance of the three programs is much more stable (indicated by a small σ) under Canvas than Linux—variations are reduced by 7× overall.

6.4 Effectiveness of Adaptive Optimizations

This subsection evaluates the benefit of each swap optimization *on top of the isolated swap system* by turning it on/off.

6.4.1 Adaptive Swap Entry Allocator

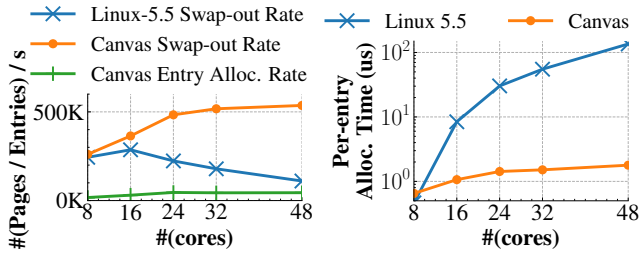
Isolation already reduces lock contention at swap entry allocation because each process has its own swap entry manager. However, for multi-threaded applications such as Spark and Cassandra, their processing threads still have to go through the locking process. In this subsection, we focus on managed applications due to their extensive use of threads. Figure 12 shows the performance of Spark LR, Spark KM, Cassandra, and Neo4j when they each co-run with the other three native programs. On average, our adaptive allocation enables an *additional* boost of 1.50× for Spark LR, 1.77× for Spark KM, 1.31× for Cassandra, and 1.28× for Neo4j.

Table 4: Swap-out throughput w/ and w/o adaptive swap-entry allocation when native programs co-run with Spark.

Thruput (KPages/s)	Linux 5.5	Canvas w/o adap. alloc.	Canvas w/
Avg. Spark apps	98	164	**295**
Avg. all apps	185	309	**468**

Table 4 reports the swap-out throughput when the native applications co-run with Spark. As shown, isolation improves the throughput by 1.67× while adaptive allocation provides an additional boost of 1.51×. This benefit is obtained after applying all optimizations in Linux 5.5.

Effectiveness of Entry Reservation. We compared our adaptive allocation algorithm with the original allocator in Linux 5.5 by running Memcached with varying (8 – 48) cores under 25% local memory. As shown in Figure 13(a), for Canvas,



(a) Swap-out and entry alloc rates. (b) Per-entry alloc time.

Figure 13: Entry allocation comparison between the allocation algorithm in Canvas and Linux 5.5 for Memcached under 25% local memory. The Y-axis in (b) is log-scaled.

(1) the swap-out rate increases with the core number (showing good scalability) and (2) the swap entry allocation rate remains low. This is due to Canvas's entry reservation algorithm that effectively reuses a significant number of swap entries for page swap-outs. On the contrary, in Linux 5.5, the swap-out rate (which is the same as its entry allocation rate) decreases when more cores are used. This is because each entry allocation takes significantly longer, reducing the swap-out throughput. A comparison of per-entry allocation time can be seen in Figure 13(b). We additionally compared the allocation algorithm between Canvas, Linux 5.5, and Linux 5.14; these results are reported in Appendix B.

6.4.2 Prefetching Effectiveness

Our baseline is the kernel's default prefetcher on the isolated swap system with adaptive swap allocator *enabled*. Since application-tier prefetching is designed primarily for high-level languages, here we focus on managed programs.

Time. We compare the running time for three Spark applications LR, KM, TC, and Neo4j, between the kernel's prefetcher over Canvas's isolated swap system and Canvas's two-tier prefetcher, when each managed application co-runs with the three native applications under the 25% local memory configuration. Application-tier prefetching brings **33%**, **17%**, **19%**, and **8%** additional performance benefits on top of the kernel prefetching with the isolated swap system. All the four managed applications benefit from the thread-level pattern analysis while the managed applications have seen 5-9% contributions from using the reference-based pattern. The thread-level pattern analysis we added for native programs brings a 5% and 11% improvement for Memcached and XGBoost.

We have also run Leap [73], a prefetcher that aggressively prefetches a number of contiguous pages if it cannot find any pattern. This approach may work for native applications because these applications access arrays; hence, the contiguous pages aggressively prefetched are likely to be useful for array accesses. However, it works poorly for high-level language applications such as Spark and Neo4j, which use deep data structures and run graph-traversal GC tasks (which exhibit neither sequential nor strided patterns). Aggressively prefetching useless pages wastes the RDMA bandwidth and the swap cache. Leap slows down our managed applications by 1.4×, compared to the kernel's default prefetcher.

Table 5: Prefetching contribution and accuracy when different Spark and Neo4j co-run with native applications.

Contribution	Spark-LR	Spark-KM	Spark-TC	Neo4j
Leap	23.4%	25.8%	42.2%	**67.0%**
Kernel	63.3%	68.0%	65.9%	41.1%
Canvas Two-tier	**79.2%**	**79.3%**	**75.3%**	45.0%
Accuracy	**Spark-LR**	**Spark-KM**	**Spark-TC**	**Neo4j**
Leap	16.8%	17.2%	35.9%	6.1%
Kernel	95.6%	96.4%	93.9%	80.4%
Canvas Two-tier	**94.3%**	**94.8%**	**94.9%**	**87.1%**

Prefetching Contribution and Accuracy. Table 5 compares prefetching *contribution* and *accuracy* for the four managed applications when each of them co-runs with the same three native applications. Contribution is defined as a ratio between the number of page faults hitting on the swap cache and the total number of page faults (including both cache hits and demand swap-ins). Accuracy is defined as a ratio between the number of page faults hitting on the swap cache and the total number of prefetches. Clearly, contribution has a strong correlation with performance while accuracy measures the pattern recognition ability of a prefetcher. For example, for a conservative prefetcher that prefetches pages only if a pattern can be clearly identified, it can have a high accuracy (*i.e.*, prefetched pages are all useful) but a low contribution (*i.e.*, the number of prefetches is small).

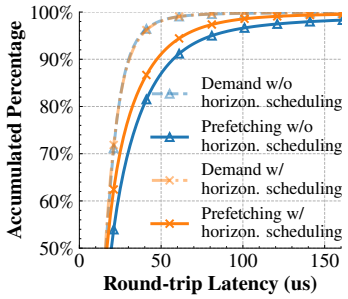
Here we report prefetching contribution and accuracy for three prefetchers: Leap (on our isolated swap system), the kernel prefetcher (also on our isolated swap system), and Canvas's two-tier prefetcher. Among the three prefetchers, for all but Neo4j, Leap has the lowest accuracy and contribution because it is an aggressive prefetcher. Leap keeps prefetching pages even when it cannot detect any patterns, which greatly reduces the prefetching accuracy. Second, due to the limited swap cache, the useless pages prefetched can cause previously prefetched pages to be released before they are accessed, hurting contribution. The kernel prefetcher and Canvas have comparable accuracy because the kernel prefetcher is much more conservative than Leap. It stops prefetching when no clear pattern can be observed. However, Linux has lower contribution than our two-tier prefetcher since Canvas prefetches more useful pages using semantics.

6.4.3 RDMA Scheduling

We evaluate our two-dimensional RDMA scheduling. For the vertical dimension, we use the weighted min-max ratio (WMMR) $\frac{\min(x_i/w_i)}{\max(x_i/w_i)}$ [96] as our bandwidth fairness metric (the closer to 1, the better), where x_i is the bandwidth consumption of the application i, and w_i is its weight. We set the

weight proportionally to the average bandwidth of each application when running individually. Our vertical scheduling achieves an overall of **0.88** WMMR.

The horizontal dimension (*i.e.*, priority scheduling with timeliness) is our focus here because interference between prefetching and demand swapping is a unique challenge we overcome in this work. We ran GraphX Connected Components (GraphX-CC) with the three native applications. Figure 14 compares the latency of sync vs. async swap-in requests with and without the horizontal scheduling of RDMA.



w/o horizontal scheduling
Contribution: 52.1%
Accuracy: 46.9%
w/ horizontal scheduling
Contribution: **62.8%**
Accuracy: **52.4%**

(a) Latency CDF. (b) Prefetching effectiveness.

Figure 14: Horizontal scheduling effectiveness for GraphX-CC: (a) prefetching latency reduced, and (b) prefetching contribution and accuracy improved.

As shown, our scheduler does *not* incur overhead for the synchronous, demand requests but reduces the (90th percentile) latency of the asynchronous prefetching requests by ∼5%. Note that these results were obtained with Canvas's two-tier prefetcher enabled, which already generates precise prefetching requests. With the Leap prefetcher, the (90th percentile) latency reduction can be as high as **9**×. To understand how the latency reduction improves prefetching effectiveness, we have also compared the prefetching contribution and accuracy with and without the horizontal scheduling, as shown in Figure 14(b). Due to the high timeliness requirement of prefetching requests, even 5% latency reduction can lead to noticeable improvements in prefetching—*e.g.*, the contribution/accuracy of GraphX-CC increases by **10.7%** and **5.5%** on top of the two-tier prefetcher—which ultimately translate to a **7-12%** overall improvement.

7 Related Work

Remote Memory. The past few years have seen a proliferation of remote-memory systems that built on the kernel's swap mechanisms (including recent works such as LegoOS [91], Infiniswap [39], Fastswap [8], and Semeru [104] as well as earlier attempts [32, 6, 31, 34, 28, 45, 61, 105]). Remote memory is part of a general trend of resource disaggregation in datacenters [43, 21, 36, 14, 11, 66, 65, 58, 7, 9, 83, 95], which holds the promise of improving resource utilization and simplifying new hardware adoption. Under disaggregated memory, application data are stored on memory servers, making swap interference a more serious problem.

Resource Isolation. Interference exists in a wide variety of settings [29, 69, 111] and resource isolation is crucial for delivering reliable performance for user workloads. There is a large body of work on isolation of various kinds of resources including compute time [64, 16, 25], processor caches [35, 57, 106], memory bandwidth [67, 68, 71, 50, 107], I/O bandwidth [40, 96, 70, 74, 97, 103, 108], network bandwidth [13, 41, 37, 94, 87, 77, 53], congestion control [27, 44], as well as CPU involved in network processing [59]. Techniques such as IX [17] and MTCP [52] isolate data-plane and application processing at the core granularity.

Prefetching. Prefetching has been extensively studied, in the design of hardware cache [101, 42, 114, 100, 76], compilers [98, 63, 89, 86, 60, 33], as well as operating systems [102, 73]. Detecting spatial patterns [75] is a common way to prefetch data. For example, various hardware techniques [93, 54, 51] have been developed to identify patterns (*i.e.*, sequential or stride) in addresses accessed. Leap [73] is a kernel prefetcher designed specifically for applications using remote memory. Swap interference can reduce the effectiveness of any existing prefetchers, let alone that none of them consider complex (semantic) patterns. Early work such as [85, 20] proposes application-level prefetching for efficient file operations on slow disks. Our prefetcher is, however, designed for a new setting where applications trigger page faults frequently and read pages from fast remote memory, with much tighter latency budgets.

RDMA Optimizations. There is a body of recent work on RDMA scheduling [88, 92] and scalability improvement [99, 24, 56, 55, 113]. These techniques focus more on scalability when RDMA NICs are shared among multiple clients.

8 Conclusion

We observed swap resources must be isolated when multiple applications use remote memory simultaneously. As such, Canvas isolates swap cache, swap partition, and RDMA bandwidth to prevent applications from invading each other's resources. Now that resource accounting is done separately for applications, Canvas offers three optimizations that adapt kernel operations such as swap-entry allocation, prefetching, and RDMA scheduling to each application's resource usage, providing additional performance boosts.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Danyang Zhuo for his feedback. This work is supported by NSF grants CNS-1703598, CCF-1723773, CNS-1763172, CCF-1764077, CNS-1907352, CHS-1956322, CNS-2007737, CNS-2006437, CNS-2128653, CCF-2106404, CNS-2106838, CNS-2147909, CNS-2152313, CNS-2151630, and CNS-2140552, CNS-2153449, ONR grant N00014-18-1-2037, a Sloan Research Fellowship, and research grants from Cisco, Intel CAPA, and Samsung.

References

[1] Large Text Compression Benchmark.

[2] NVMe over fabrics. http://community.mellanox.com/s/article/what-is-nvme-over-fabrics-x.

[3] Libsvm data: Classification. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets, 2012.

[4] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[5] Konect networks data. http://konect.cc/networks/, 2021.

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: A simple abstraction for remote memory. In *USENIX ATC*, pages 775–787, 2018.

[7] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing far memory data structures: Think outside the box. In *HotOS*, pages 120–126, 2019.

[8] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[9] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *HotCloud*, 2020.

[10] Apache. Apache cassandra. https://cassandra.apache.org, 2021.

[11] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

[13] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, pages 242–253, 2011.

[14] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

[15] L. A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture, 2018.

[16] D. B. Bartolini, F. Sironi, D. Sciuto, and M. D. Santambrogio. Automated fine-grained cpu provisioning for virtual machines. *ACM Trans. Archit. Code Optim.*, 11(3), July 2014.

[17] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI*, pages 49–65, 2014.

[18] D. Bertsekas and R. Gallager. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., USA, 1992.

[19] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[20] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, Nov. 1996.

[21] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets-XVI*, pages 164–170, 2017.

[22] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

[23] T. Chen and C. Guestrin. extreme gradient boosting for applied machine learning. https://xgboost.readthedocs.io/en/latest/, 2021.

[24] Y. Chen, Y. Lu, and J. Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *EuroSys*, 2019.

[25] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[27] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized congestion control. In *SIGCOMM*, pages 230–243, 2016.

[28] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *OSDI*, 1994.

[29] C. Delimitrou and C. Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *ASPLOS*, pages 599–613, 2017.

[30] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, Aug. 1989.

[31] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, pages 201–212, 1995.

[32] E. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. In *University of Washington CSE TR CSE TR*, 1991.

[33] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *MICRO*, pages 152–162, 2011.

[34] M. D. Flouris and E. P. Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing*, 2(4), Dec 1999.

[35] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-driven LLC allocation. In *USENIX ATC*, pages 295–308, 2016.

[36] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[37] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, pages 323–336, 2011.

[38] Google. Google's fast compressor/decompressor. https://github.com/google/snappy, 2020.

[39] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[40] A. Gulati, A. Merchant, and P. J. Varman. MClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, pages 437–450, 2010.

[41] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Co-NEXT*, 2010.

[42] Y. Guo. *Compiler-Assisted Hardware-Based Data Prefetching for next Generation Processors*. PhD thesis, 2007.

[43] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[44] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *SIGCOMM*, pages 244–257, 2016.

[45] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *Digest of Papers. Compcon Spring*, pages 538–547, Feb 1993.

[46] Intel. Batch allocation for swap entries. https://github.com/torvalds/linux/commit/ed43af10975eef7e, 2020.

[47] Intel. Memcontrol: Charge swap-in pages to cgroup. https://github.com/torvalds/linux/commit/4c6355b25e8bb83c, 2020.

[48] Intel. Per-core cluster allocation. https://github.com/torvalds/linux/commit/490705888107c3ed, 2020.

[49] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *USENIX ATC*, pages 519–532, 2018.

[50] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007.

[51] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, pages 247–259, 2013.

[52] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. MTCP: A highly scalable user-level TCP stack for multicore systems. In *NSDI*, pages 489–502, 2014.

[53] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, and C. Kim. EyeQ: Practical network performance isolation for the multi-tenant cloud. In *HotCloud*, 2012.

[54] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, pages 252–263, 1997.

[55] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *USENIX ATC*, pages 437–450, 2016.

[56] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, pages 185–201, 2016.

[57] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *ASPLOS*, pages 729–742, 2014.

[58] K. Keeton. The Machine: An architecture for memory-centric computing. In *ROSS*, 2015.

[59] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating network-based cpu in container environments. In *NSDI*, pages 313–328, 2018.

[60] A. Kolli, A. Saidi, and T. F. Wenisch. RDIP: Return-address-stack directed instruction prefetching. In *MICRO*, pages 260–271, 2013.

[61] S. Koussih, A. Acharya, and S. Setia. Dodo: a user-level system for exploiting idle memory in workstation clusters. In *HPDC*, pages 301–308, Aug 1999.

[62] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, pages 317–330, 2019.

[63] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, pages 129–142, 2005.

[64] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, pages 65–74, 2009.

[65] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, pages 267–278, 2009.

[66] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, pages 1–12, 2012.

[67] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, pages 367–376, 2012.

[68] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *ISCA*, pages 169–180, 2014.

[69] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2), 2016.

[70] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *OSDI*, pages 81–96, 2014.

[71] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, H. Wang, L. Zhang, and Y. Bao. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *ASPLOS*, pages 131–143, 2015.

[72] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, pages 457–471, 2016.

[73] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[74] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *USENIX ATC*, 2010.

[75] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[76] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), 2016.

[77] Y. Mundada, A. Ramachandran, and N. Feamster. Silverline: Data and network isolation for cloud services. In *HotCloud*, 2011.

[78] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *SOSP*, pages 538–553, 2019.

[79] Neo4j. Neo4j graph data platform. https://neo4j.com, 2021.

[80] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ASPLOS*, pages 56–69, 2018.

[81] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[82] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, pages 675–690, 2015.

[83] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[84] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[85] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, pages 79–95, 1995.

[86] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, pages 285–297, 2015.

[87] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *SIGCOMM*, pages 187–198, 2012.

[88] H. Qiu, X. Wang, T. Jin, Z. Qian, B. Ye, B. Tang, W. Li, and S. Lu. Toward effective and fair RDMA resource sharing. In *APNet*, pages 8–14, 2018.

[89] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS*, pages 189–198, 2004.

[90] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, pages 315–332, 2020.

[91] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[92] D. Shen, J. Luo, F. Dong, X. Guo, K. Wang, and J. C. S. Lui. Distributed and optimal rdma resource scheduling in shared data center networks. In *INFOCOM*, pages 606–615, 2020.

[93] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *MICRO*, pages 42–53, 2000.

[94] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[95] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[96] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, pages 349–362, 2012.

[97] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *SOSP*, pages 182–196, 2013.

[98] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[99] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *SOSP*, pages 306–324, 2017.

[100] S. P. Vander Wiel and D. J. Lilja. When caches aren't enough: data prefetching techniques. *Computer*, 30(7):23–30, 1997.

[101] S. P. Vander Wiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 372–377, 1999.

[102] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *SIGMETRICS*, pages 33–43, 1998.

[103] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[104] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, Nov. 2020.

[105] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *OSDI*, pages 35–53, 2022.

[106] X. Wang and J. F. Martínez. ReBudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. In *ASPLOS*, pages 19–32, 2016.

[107] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-Flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*, pages 607–618, 2013.

[108] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-level i/o scheduling. In *SOSP*, pages 474–489, 2015.

[109] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[110] L. Zhang. A new architecture for packet switching network protocols. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1989.

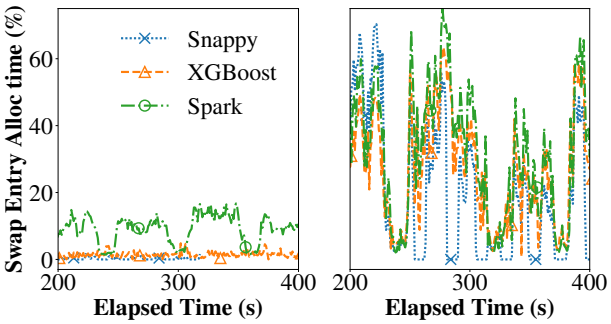
[111] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhuy. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev.*, 42(4):62–71, 2015.

[112] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, pages 379–391, 2013.

[113] Y. Zhang, Y. Tan, B. E. Stephens, and M. Chowdhury. RDMA performance isolation with justitia. In *NSDI*, 2022.

[114] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782–796, 2000.

A Extended Motivation



(a) Running individually. (b) Corunning.

Figure 15: Percentage of time spent on swap entry allocation when applications run individually (a) and together (b).

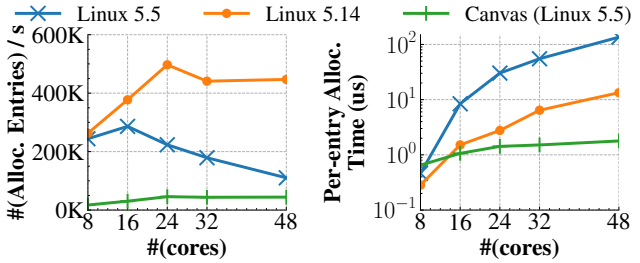
Figure 15 compares the percentage of time spent on swap entry allocation between individual runs and co-runs under Linux 5.5. As shown, each application, when co-run with other applications, spend significantly more time on allocating swap entries due to the increased locking time.

B Recent Kernel Development

As an optimization in Linux 5.5, the kernel keeps swap entries for clean pages—when clean pages are evicted, they do not need to be written back if their swap entries are not released for other allocations. Once a page becomes dirty, its swap entry must be immediately released. Clearly, this approach works for read-intensive applications where most pages are clean, but not for write-intensive workloads such as Spark. We tried various entry-keeping thresholds (*i.e.*, entry keeping starts when the percentage of available swap entries exceeds this threshold) between 25% and 75%, and saw only marginal performance differences (<5%) across our programs.

We have closely followed the kernel development since the release of Linux 5.5 and found two recent patches related to our approach. These two patches, submitted by Intel and merged into the kernel at 5.8, also attempt to optimize locking overhead at swap entry allocation. The idea of the first patch [48] is using fine-grained locking—dividing swap entries into *clusters* and assigning each core a random cluster upon an allocation request. The second patch [46] performs batch entry allocation by scanning more swap entries while holding the lock to make each batch larger. Note that our adaptive allocation algorithm solves a much bigger problem than these patches—Canvas *avoids* allocating entries for most swap-outs, while these patches reduce the overhead of locking for each allocation. As such, Canvas is completely lock-free for reserved entries while these patches must still go through the allocation path, requiring locking if multiple cores are assigned the same cluster (*i.e.*, core collision).

In fact, the probability of collision increases quickly with the number of cores. As shown below in Figure 16, the allocation performance of these patches degrades super-linearly when the number of cores exceeds 24. Another major drawback is that none of these patches build on isolated swap partitions. Lack of swap partition isolation makes applications search for swap entries globally, which can still result in interference—applications such as Spark can quickly saturate these clusters with all its executor threads, making other applications wait before they can obtain the locks. By reserving entries, our algorithm significantly reduces the number of entry allocation requests (due to entry reusing) and the cost of each allocation (due to reduced lock contention).



(a) Swap entry allocation rate. (b) Per-entry allocation time.

Figure 16: Entry allocation comparison between Canvas and the allocation algorithm when Memcached runs on Linux 5.14 on RAMDisk.

Comparison with Linux 5.5 and Linux 5.14. As the kernel is fast evolving and our latest InfiniBand driver is only compatible with Linux 5.5, we compared the swap-entry allocation performance between Canvas, Linux 5.5, and the latest Linux 5.14 over RAMDisk, by running Memcached with varying (8 – 48) cores.

As Figure 16(a) shows, our adaptive entry reservation algorithm reduces the allocation rate by several orders of magnitude compared to Linux 5.14. Note that the allocation rate under Linux 5.5 drops as the number cores increases because each allocation takes much longer and hence the swap-out throughput (*i.e.*, allocation throughput) reduces (*i.e.*, the application runs slower).

Figure 16(b) compares our algorithm with Linux 5.5 and Linux 5.14 on per-entry allocation time. As shown, the optimization in [48, 46] is unscalable—as the number of cores increases, the per-entry allocation cost increases significantly. In fact, the allocation cost grows superlinearly after 24 cores due to core collision. On the contrary, Canvas's per-entry allocation cost remains low and stable. With 48 cores, our algorithm outperforms Linux 5.14's entry allocator (that uses [48, 46]) by **13**×.

Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony

Yifan Qiao[†*] Chenxi Wang[†◇] Zhenyuan Ruan[‡] Adam Belay[‡] Qingda Lu[♯]
Yiying Zhang[§] Miryung Kim[†] Guoqing Harry Xu[†◇]
[†]*UCLA* [‡]*MIT CSAIL* [♯]*Alibaba Group* [§]*UCSD*

Abstract

Remote memory techniques are gaining traction in datacenters because they can significantly improve memory utilization. A popular approach is to use kernel-level, page-based memory swapping to deliver remote memory as it is transparent, enabling existing applications to benefit without modifications. Unfortunately, current implementations suffer from high software overheads, resulting in significantly worse tail latency and throughput relative to local memory.

Hermit is a redesigned swap system that overcomes this limitation through a novel technique called *adaptive, feedback-directed asynchrony*. It takes non-urgent but time-consuming operations (*e.g.*, swap-out, `cgroup` charge, I/O deduplication, *etc.*) off the fault-handling path and executes them asynchronously. Different from prior work such as Fastswap, Hermit collects runtime feedback and uses it to direct how asynchrony should be performed—*i.e.*, whether asynchronous operations should be enabled, the level of asynchrony, and how asynchronous operations should be scheduled. We implemented Hermit in Linux 5.14. An evaluation with a set of latency-critical applications shows that Hermit delivers low-latency remote memory. For example, it reduces the 99th percentile latency of Memcached by **99.7% from 36 ms to 91 µs**. Running Hermit over batch applications improves their overall throughput by **1.24**× on average. These results are achieved without changing a single line of user code.

1 Introduction

Techniques enabling datacenter applications to use remote memory [10, 17, 28, 29, 36, 43, 52, 53, 57] have gained traction due to their potential to break servers' memory capacity wall, thereby significantly improving datacenters' resource utilization. Compared to clean-slate techniques [17, 52] that provide new primitives for developers to efficiently manage remote memory, swap-based techniques [3, 10, 29, 53, 57, 58] that piggyback on existing paging/swap mechanisms in the OS kernel are more practical as they offer transparency, allowing legacy code to run *as is* on a far-memory system.

The main drawback of swap-based remote access is the overhead incurred by the kernel's paging system. For example, when running Memcached using Fastswap [10], the current state-of-the-art swapping system for Linux, a remote access takes an average of *14 µs*, of which only *9 µs* are spent on network (RDMA) operations—the software-induced overhead is *above 50%*! This large fault-handling overhead significantly increases operation latency, precluding the use of remote memory with latency-critical applications.

In addition, long remote-access time can further block subsequent instructions dependent on these accesses, leading to substantial reductions in application throughput. For example, the performance of garbage collection in a managed language runtime is highly sensitive to memory access latency due to its pointer-chasing nature. Reductions in GC performance can lead to delayed object creations, dramatically reducing the application's overall throughput [42, 57, 58].

The underlying reason for such high overhead is a mismatch in the design of today's swap-based paging systems, which originally targeted slow, disk-based storage, and modern datacenter networks (*e.g.*, 100-400 GbE) that can deliver pages much faster. For example, through profiling, we reveal the following performance bottlenecks that persist in Linux (§3):

- **Page reclamation blocks the critical path**: To make room to fault in new pages, the OS must reclaim memory by swapping out cold pages. Linux is designed to handle this asynchronously by swapping out pages in a separate thread. However, when Linux fails to keep up with the demand for new pages, the page fault handler must block and wait for reclamation to finish.
- **Duplication checks are too conservative**: Linux is designed to never make duplicate I/O requests for the same page. Although this occasionally prevents wasted bandwidth, it comes at a high cost in terms of synchronization overhead, such as during swap cache lookup and insertion.
- **Opportunities for batching are not exploited**: Batching can be an effective optimization when it does not harm page fault handling latency. For example, when Linux performs page reclamation, it first selects a set of victim pages and then swaps out each page individually. A better strategy would be to process victim pages in batches, reducing the cost of TLB shootdowns, I/O writes, and `cgroup` accounting.

State of the Art. The conventional wisdom is that software overheads can be overcome by bypassing the ker-

[*] Part of the work was done when Yifan Qiao interned at Alibaba Group.
[◇] Corresponding authors.

nel [48, 52, 63]. This approach typically requires application-level modifications or the use of custom APIs, making it impractical to deploy transparently across all applications. Our aim is to answer the following question instead: *Can we eliminate performance bottlenecks in the kernel directly, allowing the benefits of fast remote memory to be exposed to all applications transparently?*

Recent work, such as Fastswap [10] and InfiniSwap [29], has made some progress in optimizing the kernel's swap subsystem, such as the use of RDMA to deliver remote pages more efficiently. Fastswap, the current state-of-the-art, also modifies the Linux Kernel to offload page reclamation to a dedicated core and executes it asynchronously. This increases swap-out efficiency, and reduces the time that a page fault handler must block waiting for reclamation to finish. However, Fastswap leaves other opportunities for asynchrony on the table. In addition, a single, dedicated core is insufficient to accommodate changes in demand for swap-out throughput under time-varying memory pressure, limiting the conditions where Fastswap can perform well (§3).

Insights. This work builds on three insights, all centering around asynchrony. First, asynchrony can be used to reduce the latency of page fault handling. For example, during a page fault, the kernel first looks for the page in the swap cache. If the page is present, it will be mapped at the faulting address and the kernel does not need to issue a fetch. However, this check is protected by a lock, which incurs a non-trivial overhead. Instead, fetching a page via RDMA, even if the page is already in the swap cache, is extremely fast: its only penalty is slightly wasted network bandwidth (*i.e.*, bandwidth is rarely saturated). By always issuing the fetch asynchronously and overlapping it with the check, we can reduce the fault-handling latency.

Second, only page faults handlings are latency critical, so it is safe to aggressively optimize all other operations for throughput via batching. For instance, when TLB shootdowns are batched, it reduces the number of interrupts that have to be sent across cores. As another example, RDMA writes of multiple swapped-out dirty pages can be batched into a single transfer. These opportunities are only possible because such operations are conducted asynchronously; otherwise, batching would delay critical swap-in operations.

Third, to achieve optimal performance, the use of asynchrony (e.g., number of cores) must be adjusted dynamically. For example, it is critical that swap-out throughput is perfectly balanced with swap-in throughput. If swap-out throughput is too low, the page fault handler will block and delay the application. If it is too high, it will leave a substantial portion of local memory underutilized, impacting application performance. This is especially challenging because the swapping rate depends on the workload, its inputs, and even the different phases within its execution.

Hermit. This paper presents Hermit, a new paging/swap system that exploits these (previously-unknown) opportunities for asynchrony. Hermit employs *feedback-directed asynchrony* as the major principle in the paging system design, simultaneously enabling full code transparency (*i.e.*, any legacy code can run *as is*), low remote access latency, and high application throughput. Hermit employs different types of asynchrony to tackle the three bottlenecks (*i.e.*, blocked swap-ins, conservative checks, lack of batching), as elaborated below:

First, page reclamation is moved into a set of reclaim threads, which eagerly evict (least-recently used) pages and aggressively batch expensive operations involved in each swap-out (§4.2). In particular, Hermit batches page unmapping, TLB shootdown, RDMA writes, polling, and `cgroup` uncharging in swap-out threads, reducing the amounts of computation involved in swap-outs and improving their throughput (§4.4).

Second, Hermit opportunistically bypasses the swap-in duplication check and issues I/O read requests eagerly, delaying such checks to the synchronous PTE update stage. Since only one thread can successfully update the PTE, all other competing threads will eventually release their duplicate pages, guaranteeing safety (§4.3).

Third, inspired by optimistic locking [4], Hermit makes page I/O fully asynchronous during swap-in to further reduce latency. We split the swap-in procedure into two components: one that can still successfully run and is reversible even if there are concurrent updates, and a second that may either abort or create irreversible side effects in the presence of concurrent updates. Hermit moves the first component out of the critical section to overlap it with the page I/O (details are in Figure 4). Hermit checks the validity before the critical section finishes (*i.e.*, whether concurrent updates have occurred) and if they have, reverts the speculatively executed operations.

Finally, we create a feedback control system for each type of asynchronous operation, using execution profiles to adjust whether and how asynchrony should be applied. In particular, we use (1) *page turnaround* (*i.e.*, time between a page's swap-in and previous swap-out), (2) *page-in/-reclamation throughput*, and (3) *conflict rates* (*i.e.*, how often concurrent updates occur), as metrics to adjust our asynchrony in dealing with reclamation timing, reclamation intensity, eager swap-in, conservative checks, respectively. Hermit profiles and collects these signals throughout the execution to dynamically adapt to the application's changing behaviors.

Results. Hermit was implemented in Linux 5.14 (released August 2021). We have carefully inspected all relevant kernel patches made since then and confirmed that none of them are directly related to Hermit.

We evaluated Hermit with a set of real-world applications including both latency critical (Memcached, SocialNet, and Gdnsd) and batch processing applications (Apache Spark, XGBoost, and Apache Cassandra). Our evaluation on Memcached demonstrates that Hermit outperforms Fastswap [10]

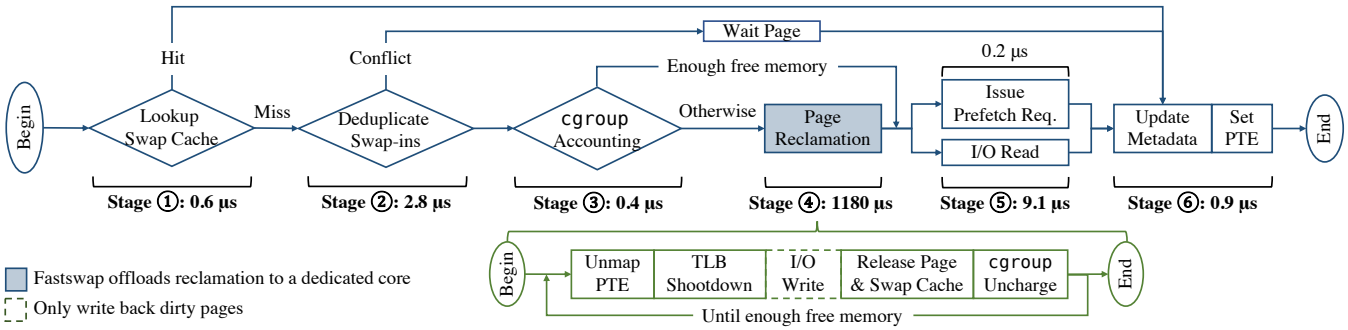


Figure 1: The life cycle of a remote memory page fault in Linux swap.

by **99.7%** in latency, reducing the 99th percentile latency **from 36 ms to 91 µs**. For batch processing applications, Hermit improves throughput by up to **1.87**× with a geometric mean of **1.24**×. Hermit also scales much better with the number of cores than Fastswap. These results demonstrate that low tail latency and high throughput can be achieved at the same time without bypassing kernel, making Hermit a practical solution for enabling remote memory. Hermit is available at https://github.com/uclasystem/hermit.

2 Background

Today's datacenter applications expose high load variability and diurnal patterns [13, 14, 47]. Despite low average load, operators have to provision resources for peak demand to avoid violating service-level agreements. Memory is an especially challenging resource because it is *uncompressable*, meaning that running out of it causes tasks to be killed, which can be very disruptive to overall performance [56]. This is a major contributing factor to low memory utilization in today's datacenters [41, 56].

Remote memory offers a promising solution to improving memory utilization. Its key idea is to break the hardware boundary and unstrand the idle memory of remote machines through fast datacenter networking. Existing systems have demonstrated the feasibility of utilizing remote memory with good performance [10, 17, 52]. Among different approaches to realizing remote memory, the kernel-based approach offers a unique advantage of transparency. It enables existing applications to run as is over remote memory using commodity hardware. This is very attractive to datacenter operators as it significantly lowers the bar for adoption.

The kernel-based approach achieves transparency through paging, an idea that dates back to the 1960s. Originally, paging was designed to extend the addressable memory space with a slow but large secondary storage (usually a mechanical disk). Under memory pressure, the kernel pages out cold pages to disk and marks them as absent from memory. Later, if a process accesses any of those pages, the memory management unit (MMU) raises a page fault exception which transparently traps the control flow into the kernel to page in the data and update the corresponding page table entry (PTE).

Linux implements paging in its swap subsystem, which is often used as the last resort for preventing out-of-memory (OOM) killing. Swap can serve as a temporary mechanism that buys operators time to solve memory pressure, *e.g.*, by migrating or killing processes. The architecture of the paging/swap subsystem has remained relatively stable since its inception. However, in the context of remote memory, fast network-attached memory (4 µs, 12 GB/s) can be used as a secondary storage device as opposed to a slower disk (10 ms, 200 MB/s). Due to this huge performance gap, the legacy swap system is a bottleneck in accessing remote memory. For example, when running Memcached on Fastswap (*i.e.*, the state-of-the-art swap system) with a high local memory ratio (70%), we see a 4× throughput drop.

3 Understanding Existing Swap Systems

3.1 The Life Cycle of Remote Memory Access

The legacy design of Linux swap imposes high overheads on accessing remote memory. To better understand the root cause of its inefficiencies, we conducted a performance study by running Memcached on Fastswap [10] (the state-of-the-art swap system). Figure 1 shows the stages of a remote memory access and breaks down their costs. We discuss each stage in more detail as follows:

① Lookup swap cache. The swap cache serves as a centralized component that prevents race conditions. It tracks the information of swapped-in pages and ongoing swap-out requests. First, the faulting page may have been fetched by another process or the OS prefetcher. By looking up the swap cache, Linux detects this and jumps to stage ⑥. Second, it is possible that the faulting page is being swapped out by another process. In this case, naïvely fetching the remote page will see the stale copy. With the swap cache, Linux detects the race and cancels the ongoing swap-out. Looking up the swap cache takes an average of 0.6 µs.

② Deduplicate swap-ins. At the same time, there can be multiple threads swapping in the same page. Linux guarantees that only one thread can succeed by synchronizing with lock primitives. The remaining threads will be busy waiting until the page gets fetched. This design saves I/O bandwidth

but impacts latency and hurts scalability. This stage takes an average of 2.8 µs.

③ **`cgroup` accounting.** Before fetching the page, Linux must ensure that the current process has sufficient free memory by performing `cgroup` accounting. For the lucky process with enough memory, it jumps to stage ⑤ directly. The accounting stage takes an average of 0.4 µs. Otherwise, Linux must go through stage ④ to reclaim pages to make room, as elaborated below.

④ **Direct page reclamation.** Linux iteratively reclaims pages until the size of the available local memory is above the low-water mark. Linux swaps out a single page for each iteration. Swap-out is expensive as it involves operations such as TLB shootdown, PTE unmapping, *etc.* This stage exists only when the local memory runs low, but it is also the longest one that takes an average of 1180 µs. To reduce direct reclamation, Fastswap performs this stage asynchronously with a dedicated core.

⑤ **Fetch and prefetch page.** Linux issues an I/O request to fetch the faulted page. Meanwhile, it may issue multiple prefetching requests. This stage takes an average of 9.1 µs.

⑥ **Update metadata.** Finally, Linux updates kernel metadata, including page table entries (PTEs), swap entries, and page reverse mapping (`rmap`). This stage takes 0.9 µs.

3.2 Root Causes of Inefficiencies

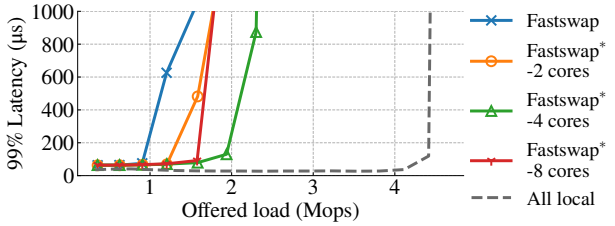


Figure 2: 99th percentile latency with respect to offered load of Memcached on Fastswap under 70% local memory.

To understand the bottleneck imposed by Fastswap's single, dedicated reclamation core, we ran several experiments with Memcached. Figure 2 shows Memcached's 99th percentile latency with respect to its offered load when running with 70% local memory. The baseline for comparison is Memcached running locally (100% local memory without swapping), which is the rightmost curve and achieves >4.4 Mops load throughput with good tail latency. Memcached on Fastswap (the blue curve), however, can only offer ≈1 Mops load before the dedicated core gets saturated and its latency increases dramatically. The reason is that Fastswap's single dedicated core cannot keep up with the increasing demand for page reclamation. We then modified Fastswap's original implementation to offload page reclamation onto multiple cores, denoted as `Fastswap*` in the figure, as a naïve strawman approach.

Using more dedicated cores can indeed help reduce the direct reclamation ratio, as shown in Figure 3. With 4 dedicated cores, `Fastswap*` is able to eliminate direct page

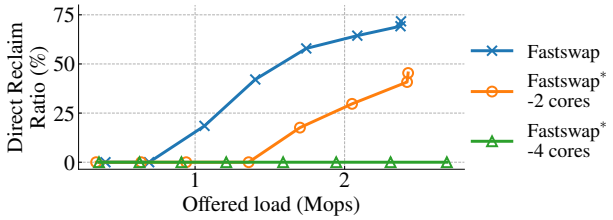


Figure 3: Direct page reclamation ratio of Memcached on Fastswap under 70% local memory.

reclamation, thus providing the highest throughput among all Fastswap variants. However, Fastswap uses static core provisioning, which is insufficient in practice due to the phased behaviors and shifts in load that occur within datacenter applications. First, the number of required dedicated cores depends on the application's working set, the available local memory, and the swap-in intensity, making it impossible for a statically determined number to work universally for different applications or even different phases of the same application. Second, over-provisioning dedicated cores does not always lead to greater end-to-end performance; in many cases, using more cores only shifts the bottleneck from page reclamation to the application itself, as more dedicated cores for reclamation imply fewer available cores for application threads. As shown in Figure 2, increasing the number of dedicated cores in Fastswap from 1 to 4 (`Fastswap*-4 cores`) improves performance, but further allocating cores degrades performance (`Fastswap*-8 cores`). Furthermore, although `Fastswap*-4 cores` eliminates direct page reclamation (*i.e.*, reducing latency), it still loses ∼45% performance (*i.e.*, reducing throughput). The performance loss is due to three major kinds of inefficiencies induced by Linux swap, as elaborated below.

Swap-out blocks swap-in. As explained earlier, Memcached experiences high memory access latency when running short of local memory, as it has to reclaim pages. Page reclamation is expensive as it requires finding victim pages and unmapping them, followed by a number of expensive operations for consistency such as TLB shootdown. This significantly impacts its tail latency, leading to violations of the service-level agreement (SLA).

Fastswap tackles this issue by allocating a dedicated core to reclaim pages asynchronously in the background. However, as discussed earlier, it is nearly impossible to statically identify the optimal number of cores due to load variability.

Unoptimized for fast I/O. Linux swap was designed for slow secondary storage like hard-disk drives whose performance is two to three orders of magnitude lower than today's remote memory in both bandwidth and latency. Since disk bandwidth is often the bottleneck, Linux applies aggressive optimizations in its page fault handling path to reduce I/O traffic (stage ②). While they were effective in the era of slow disks, these optimizations become irrelevant in the context of remote memory whose bandwidth is close to the bandwidth of main memory. Even worse, the outdated op-

timization generates an adversarial performance impact; it prolongs remote memory access latency, hurting scalability (*e.g.*, due to synchronization). For latency-critical applications like Memcached, prolonged remote memory accesses can significantly increase the time for serving incoming requests, imposing super-linear effects on tail latency. Modeled by queueing theory [24], for instance, 10% longer service time can potentially double the 99th percentile latency, leading to vast SLA violations.

Additionally, since the disk latency (ms-scale) is significantly higher than the CPU time in page fault handling (µs-scale), Linux adopts a serial-execution model for simplicity. As shown in Figure 1, the I/O read stage is executed separately from other stages; after issuing the I/O read request, Linux either busy waits for the I/O response or re-schedules the faulting thread (which hurts latency of fast I/O requests), relinquishing the opportunity of overlapping the waiting period with other stages.

Unoptimized for CPU overhead. Linux swap is a mechanism aimed at avoiding OOM killing. Inherently, treating swapping as a rare event, it was designed to optimize for responsiveness, *not* for CPU efficiency. For example, during page reclamation (stage ④), Linux swaps out only one page at a time, under the assumption that by releasing the space more timely it can unblock the OOM process sooner. Unfortunately, this amplifies the CPU usage as it must invoke expensive operations such as TLB shootdown for *every reclaimed page*. While overhead is acceptable when swapping is rare, it grows significantly in the scenario of remote memory (which is swapping-intensive). In the case of Memcached, 12.6% of the total CPU time is spent on reclaiming pages, not on application tasks. To make matters worse, Linux swap heavily relies on locks to synchronize page reclamation and scales poorly. Hence, the overhead will further increase with the number of concurrent swapping operations.

Key takeaway. Linux swap imposes high overheads to remote memory access primarily due to the above three issues. Fastswap, the state-of-the-art swap system, partially tackles the first issue, but neglects the last two. For the first issue, Fastswap uses statically provisioned cores to run swap-out tasks; as shown in Figure 2, static core provisioning cannot adapt to dynamic load changes, leading to either insufficient or wasted CPU resources.

4 Hermit Design

4.1 Design Overview

To overcome the aforementioned inefficiencies, we developed Hermit, a new swap system based on the principle of *feedback-directed asynchrony*. Our key insight is that asynchrony should be used aggressively (to overlap nonurgent and urgent operations to reduce latency), but this must be done in a controlled manner—whenever asynchrony cannot bring benefits, we should switch back to the conventional synchronous design. Figure 4 illustrates Hermit's design.

First, Hermit optimizes tail latency of accessing remote memory by moving page reclamation from the critical path into the background (§4.2). Instead of following the design of Fastswap, which statically reserves a certain number of dedicated cores, Hermit relies on a *reclaim scheduler* to dynamically schedule reclaim threads. The scheduler leverages feedback from `cgroup` counters to determine the right timing and the appropriate number of cores for reclamation.

Second, the swap-in path of Hermit was designed with fast remote memory in mind (§4.3)—for remote memory, it is reasonable to trade off network usage for end-to-end performance as modern datacenter network offers abundant bandwidth (100-400 Gbps). In the common case, Hermit detects idle network bandwidth and opportunistically bypasses swap-in duplication checks (stage ② in §3) to improve scalability and reduce latency. This bypassing has a consequence: in the (rare) case that multiple threads are fetching the same page at the same time, they will all transfer the same page over the network. Note that this will *not* lead to correctness issues because only one copy will be mapped by the PTE in the last stage, and any other requests will abort and release their page. However, it may potentially waste some network bandwidth when duplicate pages are requested. Therefore, instead of bypassing blindly, we use the conflict rate (in the last stage) as a control signal to determine whether it is beneficial to enable bypassing. To further optimize the critical-path latency, Hermit also overlaps the I/O read stage with other swap-in operations (*e.g.*, `cgroup` accounting, metadata updating, *etc.*).

Finally, we structured Hermit to operate in a swap-intensive environment to match the reality of using remote memory (§4.4). Hermit carefully optimizes the CPU usage of page reclamation so that more CPU resources are available for applications. Enabled by Hermit's reclaim scheduler, which reduces the "urgency" of reclamation tasks, Hermit opportunistically handles reclamation requests *in batches* to amortize the overhead. In addition, Hermit bypasses the expensive reverse mapping operation when swapping out a private page (which is common). As a result, Hermit not only reduces the remote access latency but also significantly improves the application's throughput.

4.2 Reclaim Scheduling

In Linux swap, the direct page reclamation in the swap-in path significantly impacts the tail latency of accessing the remote memory. To reduce tail latency, Hermit moves reclamation off the critical path into background threads; the reclaim scheduler monitors the free memory size and *proactively* starts reclamation before memory exhaustion. The scheduler uses the application's swap throughput as a feedback signal to auto-tune the number of reclaim threads.

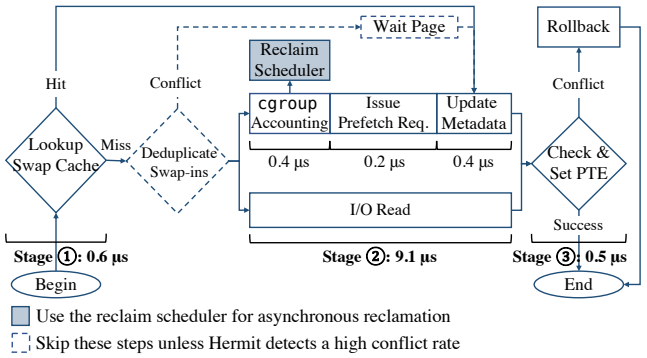


Figure 4: The life cycle of a remote memory page fault in Hermit.

Designing such a scheduler is challenging because it must determine both the right *timing* and the appropriate *amount of CPU resources* for reclamation. (1) As for the timing, if the scheduler starts reclamation too early, a substantial portion of local memory would be underutilized, impacting application performance; on the flip side, if the scheduler starts reclamation too late, the application would exhaust the local memory and suffer from direct reclamation. (2) As for CPU resources, under-provisioning cores for reclamation (*i.e.*, the case of Fastswap) make it unable to keep up with the local memory consumption rate, leading to memory exhaustion, while over-provisioning cores is also undesired as it contends with the application and reduces its performance.

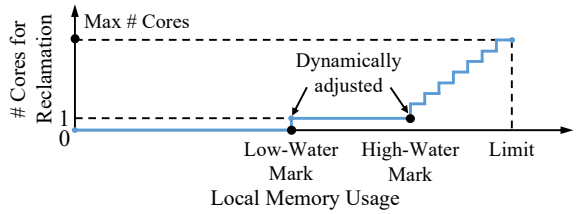


Figure 5: Adaptive reclaim scheduler.

Figure 4 shows the design of the reclaim scheduler, which leverages counters from `cgroup` to schedule reclamation. Since the timing for reclamation is critical to performance, our reclaim scheduler has to be very reactive to free memory size changes (in µs-level). Instead of using a dedicated core to poll the memory usage which waste CPU cycles, Hermit adopts a *decentralized* reclaim scheduler; it inlines the scheduler code into the `cgroup` charging, an indispensable step for swap-ins. This design enables us to discover any sudden change in the free memory size with only a few CPU cycles.

Hermit's scheduling policy follows the conventional wisdom of random early detection [26] to gradually increase its asynchronous reclaim throughput. Specifically, Hermit starts asynchronous reclamation when application's memory budget is running low, but Hermit will only enable a small number of reclaim threads first and gradually increase the number of reclaim threads after observing constantly increasing memory usage. The intention of the design is to handle a burst of swap-ins within the memory limit with as few reclaim threads as possible, and thus minimizing asynchronous reclamation's interference to the application.

On the other hand, when the application is about to run out of memory, Hermit must unleash the full power of asynchronous reclaim threads to match the reclaim throughput to swap-in throughput to avoid direct reclamation, offering the application maximum swap performance. Figure 5 depicts Hermit's adaptive scheduling policy, which determines the number of cores for page reclamation given the application's current local memory usage. The curve can be divided into three phases, marked by the low-water mark and the high-water mark to differentiate the urgency of asynchronous reclamation.

When the application does not swap intensively and its local memory usage is below the low-water mark, the number of reclamation cores is zero, indicating that the asynchronous page reclamation is disabled now to let application threads have all CPU cores. When the application's local memory usage is between the low-water mark and the high-water mark, it indicates that the application is under memory pressure, and the scheduler will assign one core for asynchronous reclamation to relieve the memory pressure with minimal compute to minimize its interference to application's threads.

Finally, when the application hits the high-water mark, it indicates that the application is about to run out of memory. Page reclamation is an urgent task now to prevent the application from triggering direct page reclamation. As such, the reclaim scheduler must assign more cores for reclamation to match the reclaim throughput with application's swap-in throughput. As Figure 5 shows, during this phase, the number of cores assigned for reclamation is proportional to the local memory usage, reaching the maximum value when the local memory usage equals the memory limit.

Hermit leverages the kernel's runtime statistics to autotune the low and high memory watermarks, as elaborated below.

High memory watermark. Hermit dynamically adjusts the high memory watermark based on the application's current *swap intensity*. We define swap intensity as the overall swap-in throughput divided by the per-core page reclamation throughput, representing the number of cores needed for reclamation to match the swap-in speed. Intuitively, when the swap intensity increases, we should lower the high-water mark to start ramping up reclamation earlier; and when the swap intensity decreases, we should raise the high-water mark accordingly. Hermit sets the high-water mark as $MEM_LIMIT - \alpha \cdot SWAP_INTENSITY$, where $\alpha = 128$ works well in practice.

Low memory watermark. Initially, Hermit sets the low-water mark to be the same as the high-water mark. Then it gradually probes its optimal value based on the average page turnaround time (APT), defined as the average duration for swapped-out pages to remain untouched. When APT does not increase, Hermit attempts to lower the low-

water mark, as now it can potentially start reclamation earlier without impacting the application performance. However, when APT increases, Hermit immediately raises back the low-water mark to revert the negative impact on the application performance.

4.3 Adapt Swap-in to Fast Remote Memory

As shown in Figure 4, Hermit re-architects the swap-in path for the fast remote memory with two main innovations.

Eager swap-in. Hermit opportunistically bypasses the swap-in duplication check to minimize latency. As such, it is now possible that multiple threads issue swap-in requests for the same page. To ensure that only one of them will succeed, Hermit synchronizes them in the final stage (updating PTE) using a fine-grained lock. All other failed threads will release their swapped-in pages—CPU cycles consumed by them are wasted and considered as penalty. Hermit collects the conflict rate and the penalty as feedback to reassess whether it is still beneficial to enable eager swap-in and disable it if it impacts performance.

Asynchronous I/O. Hermit further shortens the critical path of swap-ins by overlapping the I/O read with other operations, for example, `cgroup` charging. If later the `cgroup` check shows no memory, Hermit discards the I/O read response and updates the failure counter. Hermit falls back into synchronous I/O when the failure ratio is high. This happens very rarely in practice thanks to Hermit's asynchronous reclamation (§4.2).

4.4 CPU-Efficient Page Reclamation

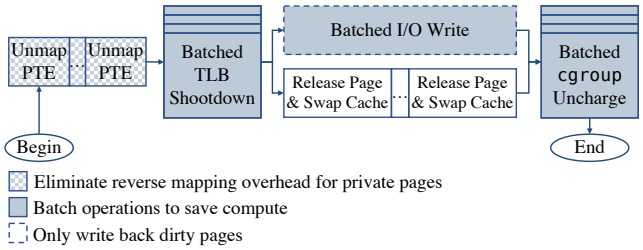


Figure 6: Hermit's asynchronous page reclamation path.

As shown in Figure 6, Hermit carefully optimizes the CPU overheads of page reclamation to minimize its performance impact to applications.

Batched reclamation. As illustrated in §3, Linux's page reclamation is mainly designed for slow disk devices where swapping occurs infrequently—it trades off CPU efficiency for responsiveness by only swapping out one page at a time. However, Hermit overcomes the responsiveness loss with its asynchronous reclamation design, which relaxes the responsiveness requirement of page reclamation, thereby creating opportunities for batching. As depicted in Figure 6, Hermit batches expensive operations, including TLB shootdowns, I/O writes, and `cgroup` accountings—to amortize their overheads in the asynchronous page reclamation path.

Reverse mapping elimination. To avoid race conditions during reclamation, Linux has to ensure that the page is immutable before writing it back to remote memory. Linux achieves this goal by using `rmap` (reverse page mapping) to identify and unmap all the virtual pages mapped to the reclaimed physical page. `rmap` walk is expensive as it involves several memory accesses and lock synchronizations. A key observation in Hermit is that most reclaimed pages are private pages (*i.e.*, only referenced by one virtual page). For private pages, Hermit eliminates the expensive `rmap` walk by inlining the virtual page address into the physical page metadata in Linux. This approach trades a tiny portion of local memory (0.2% in the worst case) for better performance.

5 Implementation

We implemented Hermit atop Linux 5.14, the latest release when we started the project. We have been carefully examining every new release to ensure that no patch is directly related to our techniques. We added or modified 9704 lines of kernel code, mainly re-implementing Linux's swap-in and swap-out code paths.

We built our RDMA-based swap backend atop Fastswap's implementation. The original Fastswap uses Linux's `frontswap` interface which only supports blocking I/O. We extended it with an asynchronous I/O interface to enable asynchronous batched I/O writes during page reclamation.

For the swap-in path, we stored the feedback signals `swap_stats`, used by Hermit to decide whether to bypass the swap-in deduplication, in Linux's process context `mm_struct`. `swap_stats` contains two atomic counters representing the numbers of successful and aborted swap-ins respectively. The page fault handler reads and updates `swap_stats` when swapping in the page.

For the swap-out path, we implemented per-`cgroup` reclaim threads as Linux kernel threads. We stored the feedback signals `swap_ctrl`, used by Hermit to decide the swap-out timing, in Linux's memory cgroup `mem_cgroup`. `swap_ctrl` contains two counters representing the total number of charged pages and reclaimed pages. Hermit updates `swap_ctrl` during `cgroup` charging and page reclamation. The reclaim scheduler reads `swap_ctrl` periodically (per 128 charges in our implementation) to calculate the swap intensity for updating the high-water mark. We use Linux's existing mechanism of tracking the page re-fault distance to calculate the average page turnaround (APT) for updating the low-water mark. Hermit batches 32 pages per NUMA node for its asynchronous page reclamation to keep low amortized overheads while ensuring most reclamations can finish timely (within 1 ms). To batch reclamation while ensuring consistency, we carefully ordered the operations (see Figure 6). Hermit first selects and unmaps a batch of pages, and then issues a single TLB flush before writing all dirty pages to remote memory. After which, Hermit rechecks each page to ensure it remains untouched and free it. Other-

wise, the page must have been faulted on and re-mapped into the process' page table, so Hermit skips freeing this page and returns it back to the application. To bypass the `rmap` walk, we stored the virtual address of private pages using a global array. We did not directly embed the virtual address into Linux's per-page metadata to avoid breaking its cache alignment.

6 Evaluation

Our evaluation seeks to answer the following questions:

1. Can Hermit maintain low tail latency (§6.2) and high throughput (§6.3) while delivering remote memory?
2. How does Hermit's performance compare to standard Linux and Fastswap [10]? (§6.2-§6.3)
3. What contributes to Hermit's better performance? (§6.4)

Setup. We ran experiments in a cluster with one CPU server and one memory server, connected by a 100 GbE network. Each server equips a 24-core AMD 7402P CPU and 128 GB memory. Both Hermit and Fastswap ran on Ubuntu 20.04 with Linux 5.14. For latency-critical applications, we generated load from another server, which connects to the CPU server via a 25 GbE network. We followed common practices to tune these servers for low latency [47], including disabling CPU frequency scaling, machine-check exceptions, and transparent hugepages. We also disabled OS security mitigations as recent CPUs have fixed these vulnerabilities. We enabled hyperthreading as it improves the performance of remote memory systems.

Methodology. We compared Hermit with the ideal system that only uses local memory and the state-of-the-art kernel-based remote memory system, Fastswap [10]. To enable a fair comparison, we also ported Fastswap to Linux 5.14, the same kernel version that Hermit uses.

6.1 Real-world Applications

We used six real-world datacenter applications for evaluation, as shown in Table 1.

Category	Application	Dataset	Size
Latency-Critical	Memcached [7]	Facebook's `USR` [14] like	32M KVs
	SocialNet [27]	Socfb-Penn94 [51]	41.5K nodes, 1.4M edges
	Gdnsd [1]	Custom	75M sites
Batch	Spark [62]	Wikipedia EN [8]	188M points
	XGBoost [21]	HIGGS [15]	21M instances
	Cassandra [9]	YCSB [22]	20M records

Table 1: Applications used in the evaluation.

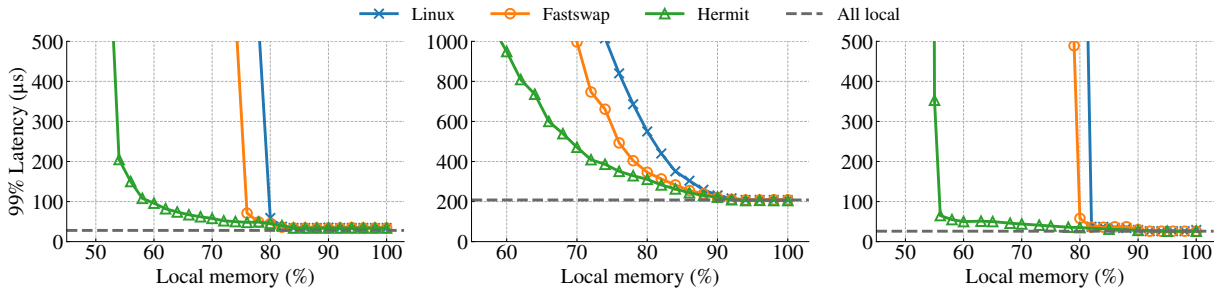
Latency-critical applications. Memcached [7] is a popular in-memory key-value store. It only performs a hash table lookup for each request, leading to a small per-request memory footprint. It has low compute intensity and poor spatial locality. We followed Facebook's `USR` distribution to generate load with 99.8% GET and 0.2% PUT [14]. SocialNet (a part of the DeathStarBench [27]) is a twitter-like interactive web application built with microservices. It has a fan-out pattern in which each client request is served by multiple microservice instances. This leads to a larger per-request memory footprint than Memcached. It has medium compute intensity and poor spatial locality. We rewrote DeathStarBench's python-based load generator using C++ to increase its throughput. Gdnsd is an authoritative-only DNS server. It performs a tree lookup for each DNS query. It has a small per-request memory footprint and low compute intensity. Different from previous applications, Gdnsd has good spatial locality. We generated queries with random domain names for evaluation. For all three applications, we generated requests with keys followed Zipf distribution using the skewness parameter $s = 0.99$, to be consistent with the standard YCSB benchmark suite [22].

Batch applications. Apache Spark [62] is a big data analytics engine. We used the logistic regression model from its official example suite for evaluation, in which Spark trains the model iteratively by scanning the dataset to update the model parameters. It has high compute intensity and a large memory footprint. XGBoost is a gradient boosting library for machine learning. We ran binary classification for evaluation. It initializes a group of decision trees and trains them iteratively by splitting the tree leaves with input data. It has dynamic parallelism and a medium memory footprint. Apache Cassandra [9] is a large-scale NoSQL database. It uses a storage structure similar to a log-structured merge tree, which has medium compute intensity and good spatial locality. Different from other batch applications, it also periodically persists in-memory data to disk. We used YCSB [22] as its workload for evaluation. Both Spark and Cassandra are Java-based and run atop OpenJDK-11. Java's garbage collection makes them more memory intensive. XGBoost is a native C++ application.

6.2 Tail Latency of Latency-Critical Applications

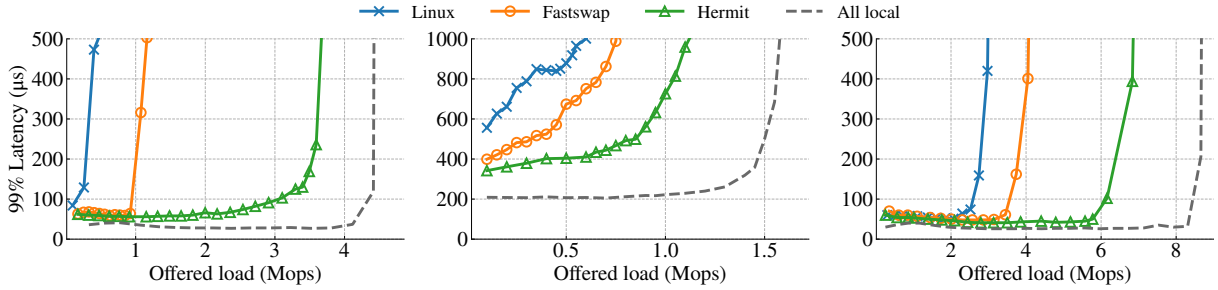
To better quantify the tail latency overhead introduced by Hermit, we use low-latency applications enabled by Shenango (a recent datacenter library OS) [47], for evaluation. With Shenango's low-latency threading runtime and network stack, these applications achieve sub-millisecond tail latency, making it an extremely challenging case for swap systems. We also rerun the same applications with their vanilla (Linux-based) versions. The results (in Appendix A) show similar trends but with higher tail latency for all systems, including the ideal local-only case. This stems from the higher overhead of the kernel's threading and network stack. Following previous studies [34, 49, 63], we primarily focus on applications' 99th percentile latency in our evaluation. The results of other percentiles (including median and 99.9th) can be found in Appendix C.

We first ran applications with a fixed load (50% of load capacity measured with only using local memory) and varying



(a) Memcached (2 Mops) (b) SocialNet (0.75 Mops) (c) Gdnsd (4 Mops)

Figure 7: Hermit significantly outperforms Fastswap and Linux in terms of 99% latency under the same fixed load and varying local memory ratio. Hermit enables applications to operate in a more challenging regime of less local memory while still maintaining < 500 µs 99% latency.



(a) Memcached (b) SocialNet (c) Gdnsd

Figure 8: Hermit achieves significantly lower 99% latency than Fastswap and Linux under the same fixed local memory ratio and varying load. For Memcached and Gdnsd, Hermit achieves 99% latency close to the ideal local-only case. SocialNet is more challenging due to its higher per-request memory footprint, but Hermit still achieves 74% load capacity of the ideal case.

local memory ratios. We measured the application performance on Linux, Fastswap, Hermit, and the ideal setup that only uses local memory (see Figure 7). The original Linux does not have an RDMA-based swap backend. To enable a fair comparison, we extended it to use Fastswap's RDMA backend. On Figure 7, the X-axis shows the ratio of the local memory provisioned; the Y-axis shows the 99th percentile latency achieved by Linux, Fastswap, Hermit, and the ideal setup.

Intuitively, both Fastswap and Hermit achieve ideal performance when only using local memory. When we decrease the local memory ratio, latency increases as remote accesses become more frequent. However, Hermit's latency increases slower than Fastswap, revealing it is more tolerant to remote accesses. This is because Hermit's overhead of accessing remote memory is lower, thanks to its shorter swap-in path and its reclaim scheduler that eliminates direct reclamation (§6.4.1). As Hermit adaptively changes the number of reclaim threads to match the reclamation rate with the swap-in rate, it can result in competition for CPU resources if the local memory ratio is small enough. Eventually, both systems encounter a "hockey-stick" when they cannot handle the excessive remote memory accesses. Compared to Fastswap, Hermit enables applications to operate in a more challenging regime of less local memory while still maintaining < 500 µs 99th percentile latency.

Specifically, the low compute intensity of Memcached and Gdnsd aligns with Hermit's optimizations well; they only require a few CPU cores for serving load, leaving the rest of the cores for reclamation. Moreover, thanks to their small per-request memory footprints, they only require a small number of reclaim threads. For Memcached, Hermit has to rely on more than four reclaim threads to keep up with frequent swap-ins when Memcached runs under < 60% local memory ratio. The CPU contention gets more severe when local memory gets smaller, and the system reaches 70% CPU utilization under 58% local memory ratio. Afterward, Hermit's reclaim threads can heavily interfere and block Memcached's threads, thus ramping up the tail latency. Similarly, Gdnsd on Hermit used ∼72% CPU cycles when running under 56% local memory ratio, and the system can no longer maintain low 99th percentile latency afterward. Fastswap's single dedicated core fails to keep up with the increasing page reclamation demand when local memory ratio is lower than 76% and 82% for Memcached and Gdnsd, respectively, which ramps up their 99th percentile latency. To conclude, Hermit pushes the operating regime in terms of local memory ratio from 75% (*i.e.*, Fastswap) to 55% for Memcached, and from 80% to 55% for Gdnsd. Gdnsd has a slightly better result due to its better spatial locality. SocialNet is a more challenging application that has a higher compute intensity and a larger per-request memory footprint. It requires more reclaim threads which compete with application threads more heavily under low local memory ratios. The system used 70% of its CPU resources under 65% local memory ratio, and saturated all CPU cores under 60% local memory ratio. Hermit pushes its

regime from 75% local memory ratio to 65%. In summary, Hermit enables applications to store an average of 20% more working set in remote memory without breaking the tail latency target, thereby harnessing stranded memory resources more efficiently.

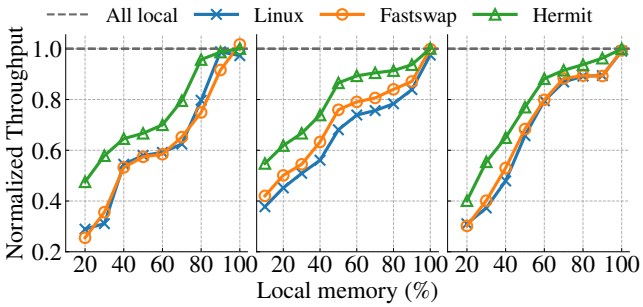
Next, we fixed the local memory ratio to 70% and measured the tail latency of applications with varying load (see Figure 8). Under low load, both Fastswap and Hermit encounter higher latency than the local-only case due to additional remote memory accesses. Hermit delivers lower latency than Fastswap due to the cheaper remote accesses it offers. For Memcached and Gdnsd whose per-request memory footprint is smaller, Hermit reduces 99th percentile latency by 3–9 µs, whereas for SocialNet, Hermit reduces latency by 43–86 µs.

Under high load, the latency gap becomes wider because of the CPU contention between application and asynchronous reclaim threads. In this case, application threads access remote memory intensively, therefore triggering memory reclamation frequently. The asynchronous reclaim threads impact application performance by contending CPU resources. Hermit experiences lower performance degradation because of its asynchronous and more CPU-efficient design of memory reclamation (§6.4.2). By eliminating blocking induced by direct reclamation and shifting more CPU resources from reclamation to application, Hermit handles higher load than Fastswap under the same local memory ratio while still maintaining $<$ 500 µs 99th percentile latency. Hermit improves the load capacity by 3.2× (from 1.1 Mops to 3.5 Mops) for Memcached, and 1.7× (from 4.0 Mops to 6.8 Mops) for Gdnsd. Notably, compared to the ideal local-only case, Hermit enables these applications to enjoy the benefit of remote memory with only an average of 20% decrease in their load capacity. It is more challenging to handle SocialNet well due to its larger per-request memory footprint and higher compute intensity. As a result, the number of reclaim threads needed increases quickly with the load, deteriorating the contention with application threads. Even though, Hermit still improves SocialNet's capacity by 1.5× (from 0.75 Mops to 1.15 Mops).

6.3 Throughput of Batch Applications

In this section, we evaluate the throughput of batch applications under varying local memory ratios (see Figure 9). Hermit outperforms both Fastswap and Linux. It only requires 45%–70% local memory to achieve at least 80% of the ideal throughput for all applications. In contrast, Fastswap (*i.e.* the better baseline) has to use an average of 20% more local memory to achieve the same throughput. Even under the extremely challenging case of 20% local memory, Hermit is still able to preserve 40%–60% of applications' ideal throughput. This leads to 1.23×–1.87× improvement over Fastswap.

When Spark runs atop Fastswap, its throughput drops significantly when running with $<$ 40% local memory. Our profiling reveals that swapping becomes extremely frequent in this case, triggering the scalability bottleneck in kernel's page reclamation path. Hermit does not suffer from the same issue due to two reasons. First, Hermit significantly reduces the direct reclamation ratio by performing reclamation asynchronously and timely. Therefore, it confines reclamation into a small number of reclaim threads rather than all the application threads (in direct reclamation). Second, Hermit's CPU-efficient reclamation design reduces the number of threads needed, further alleviating the scalability issue.



(a) Spark (68.4s) (b) XGBoost (42.2s) (c) Cassandra (72.6s)

Figure 9: We measured the throughput of batch applications achieved by different swap systems normalized to the ideal local-only setup. Hermit outperforms other baselines. The number in the parenthesis shows the ideal execution time.

6.4 Design Drill-Down

We now evaluate specific aspects of Hermit's design to understand their individual contributions to overall performance.

6.4.1 Remote Memory Access Latency

Hermit reduces remote memory access latency by shortening the critical path of swap-ins. Figure 10 breaks down the improvements brought by specific optimizations, including bypassing deduplication and using asynchronous I/O. The results are measured using Memcached. Without Hermit's optimizations, the original Linux spends 2.8 µs on swap-in deduplication. Hermit eliminates this overhead entirely by opportunistically bypassing the deduplication, see Figure 11. After enabling asynchronous I/O, Hermit further overlaps I/O read with other swap-in operations (*e.g.*, `cgroup` accounting and metadata updating), reducing the swap-in latency by another 0.9 µs. With both optimizations turned on, Hermit reduces the page fault handling latency by 35%, from 13.8 µs to 10.2 µs. The RDMA backend spends 9 µs on performing a 4KiB-page I/O. This indicates that Hermit reduces the overhead of the swap system by a factor of four, from 4.8 µs to only 1.2 µs.

6.4.2 Page Reclamation Efficiency

To demonstrate Hermit's improvements on page reclamation efficiency, we ran Memcached and measured the per-thread

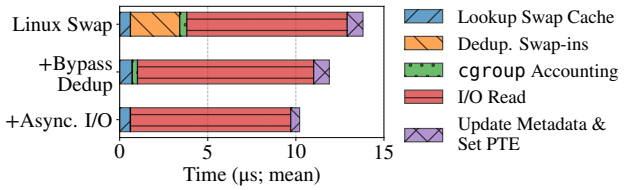


Figure 10: Hermit reduces the remote memory access latency in Memcached from 13.8 µs to 10.2 µs with two optimizations, *i.e.*, bypassing deduplication and using asynchronous I/O.

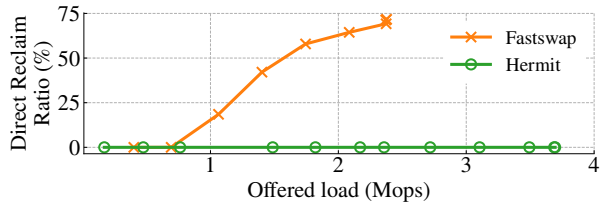


Figure 11: Hermit entirely eliminates direct reclamation for Memcached, thanks to its asynchronous reclamation design. Fastswap fails to serve > 2.4 Mops load due to CPU congestion.

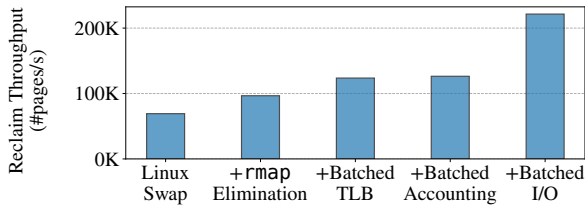


Figure 12: Eliminating reverse mappings and enabling more batching makes reclamation 2.9× more efficient.

reclamation throughput, see Figure 12. As shown by the leftmost bar, the original Linux achieves 77K pages/s reclamation throughput. Hermit's `rmap` elimination optimization effectively improves the throughput by 37%, as most of pages are private in Memcached. Batching TLB shootdowns and `cgroup` accountings amoritizes their overheads and brings an additional 27% and 3% improvement, respectively. Finally, Hermit batches I/O writes for dirty pages and overlaps them with the page release phase. This significantly reduces the time wasted on polling for the write completion, generating a 75% further improvement. Our further profiling reveals that Hermit reduces the per-page overhead of `rmap` by 59% from 1.70 µs to 0.69 µs, TLB shootdown by 92% from 2.45 µs to 0.20 µs, and I/O writes by 88% from 6.47 µs to 0.76 µs. To summarize, Hermit improves the single-thread page reclamation throughput from 77K pages/s to 221K pages/s, making reclamation 2.9× more efficient.

6.4.3 Effectiveness of Feedback-directed Asynchrony

To demonstrate the importance of Hermit's feedback-directed asynchrony, we modified Hermit's reclaim scheduler to use Fastswap's static scheduling policy. The new version `Hermit*` uses a fixed number of reclaim threads and starts reclamation only when the free local memory size falls below 8 MiB. Figure 13 shows the results of Memcached.

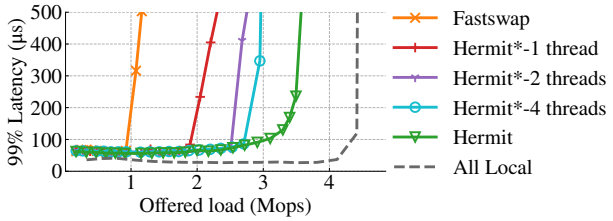


Figure 13: Hermit's feedback-directed asynchrony is indispensable for achieving superior performance. Hermit considerably outperforms all `Hermit*`s—the modified versions that adopt Fastswap's static scheduling policy for reclamation.

Hermit consistently outperforms all variants of `Hermit*`, regardless of the number of reclaim threads statically configured. Our further profiling reveals that the memory pressure during Memcached's execution varies over time. In most cases, it only requires ≤2 reclaim threads to mitigate the pressure. However, upon sudden bursts of requests, it needs up to 4 threads to fully keep up with the demand. Hermit's reclaim scheduler dynamically adjusts the number of reclaim threads to adapt to the changes in demand, thereby achieving superior performance to its static counterparts.

6.4.4 Breaking Down End-to-End Speedup

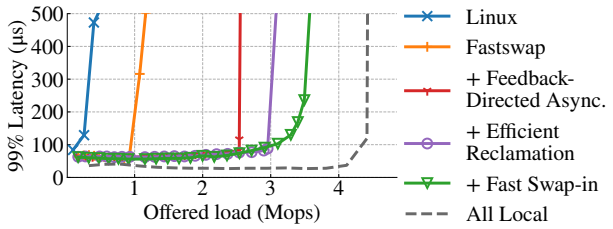


Figure 14: All three of Hermit's optimizations work in tandem to improve Memcached's latency and throughput. Results are measured with 70% local memory.

We evaluated the individual contribution of each of the three optimizations (§6.4.1-§6.4.3) to the overall application performance.

For latency-critical applications, we used Memcached as the representative. We re-ran Memcached with the same configuration as Figure 8 (a) with optimizations enabled incrementally. Figure 14 reports the results. Linux even fails to handle low load of 0.5 Mops under 70% local memory, as it frequently triggers direct reclamation which can easily prolong Memcached's 99th percentile latency by hundreds of microseconds. Fastswap outperforms Linux by offloading reclamation to a dedicated core. However, the application quickly saturates the core's reclamation capacity once the load reaches 1.1 Mops, and starts to trigger direct reclamation again (see Figure 11). This prevents Fastswap from maintaining low 99th percentile latency afterward.

With the reclaim scheduler (§4.2), Hermit can handle a much higher load, 2.5 Mops, before the latency starts to spike. This is because Hermit's reclaim scheduler proactively and timely starts asynchronous reclamation, eliminat-

ing the blocking caused by direct reclamation. Optimizations in the reclamation path (§4.4) reduce the amount of CPU resources required. This alleviates the contention between reclaim threads and application threads, adding 0.4 Mops to the load capacity. Finally, optimizations in the swap-in path (§4.3) make remote memory accesses faster and reduce the per-request processing time, thereby enabling Memcached to achieve higher load with the same amount of compute. Putting them all together, Hermit helps Memcached reach 3.5 Mops using 70% local memory while maintaining 99th percentile latency under 250 µs.

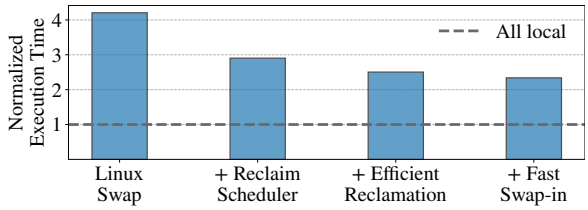


Figure 15: All three of Hermit's optimizations collectively improve Spark's throughput. The Y-axis shows the execution time normalized to the ideal local-only time (68.4s). Results are measured under 20% local memory.

For batch applications, we used Spark as the representative and re-ran it under 20% local memory with the same configuration as Figure 9(a). Figure 15 breaks down the performance improvements. Our reclaim scheduler again improves the application throughput by a large margin (31%) due to the following reasons. First, batch applications usually follow the epochal hypothesis [46], whose compute and memory behaviors vary during an epoch but repeat across epochs. Asynchronous reclamation unleashes the hidden parallelism by speculatively reclaiming pages, making it possible for reclaim threads to efficiently harness idle compute resources in each epoch. Second, Linux swap frequently triggers massive direct reclamations instantaneously, causing severe lock contentions between page faults handlings (swap-in) and reclamation. Hermit avoids the burst of reclamation and greatly alleviates the contention by reclaiming asynchronously and proactively. Further, optimizations on the page reclamation path and the swap-in path collectively improve the swap efficiency: they yield an additional 10% and 4% throughput improvement, respectively.

6.4.5 Resource Consumption of Swap Operations

Network Bandwidth. Hermit performs swap operations eagerly to improve performance. It opportunistically bypasses swap-in deduplication to reduce swap-in latency (§4.3) and proactively schedules asynchronous reclaim threads to avoid direct reclamation (§4.2). These optimizations offer performance benefits potentially at the cost of additional network usage. For example, Hermit might swap in the same page several times in the presence of concurrent page faults. To confirm that Hermit does not incur excessive network traffic, we measure the network bandwidth used for swap-ins and swap-outs, and compare it with Fastswap's usage.

Figure 16 shows the results when running Memcached. The X-axis shows the offered load while the Y-axis shows the average network bandwidth. The error bar quantifies the bandwidth fluctuation during the application's execution. With higher offered load, both Fastswap and Hermit use more network bandwidth as Memcached swaps memory more frequently. The bandwidth usage in swap-outs is lower than in swap-in as clean pages do not need to be written back during reclamation.

For swap-in, Hermit incurs similar network bandwidth usage compared to Fastswap. This is consistent with our further investigation which reveals that the conflict rate (*i.e.* the ratio of concurrent page faults that swap in the same page) is less than 0.07%. Therefore, Hermit's swap-in optimization barely introduces any extra network overhead in practice.

For swap-out, we break down the total bandwidth consumption into the usage of asynchronous swap-out and direct swap-out. Hermit is able to constantly perform asynchronous reclamation without using additional network bandwidth compared to Fastswap. This makes sense as Hermit's optimizations to reclamation timing and efficiency do not inflate the number of reclaimed pages.

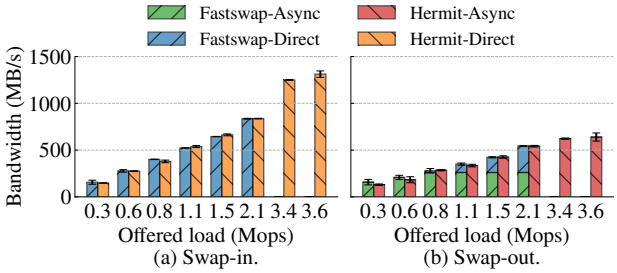


Figure 16: Hermit's optimizations do not incur additional network usage during swap-ins/-outs compared to Fastswap.

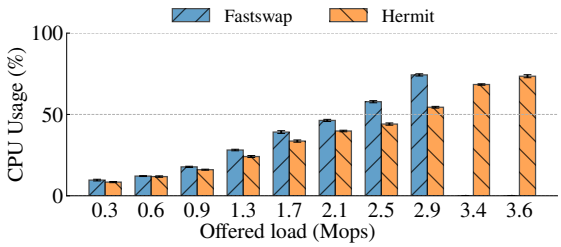


Figure 17: Hermit saves ∼30% CPU cycles under varying load compared with Fastswap, which is the key enabler to achieve low 99th percentile latency under high load.

CPU Cycles. We also profiled the CPU usage of applications running on Fastswap and Hermit, revealing that Hermit can serve much higher load with the same amount of CPU resources. Figure 17 depicts the total CPU usage of Memcached and Hermit's reclaim threads under 70% local memory ratio and varying load. When increasing load, both Fastswap and Hermit use more CPU cycles as Memcached

swaps more frequently. We observed that Memcached fails to use > 70% CPU cycles due to its internal lock contention on hot slabs under skewed workloads. Even though Hermit can spawn more reclaim threads than Fastswap (when needed), it uses 20%–30% fewer CPU resources overall, thanks to its feedback-directed asynchrony and more effective use of batching. Therefore, Hermit is able to offer 32% higher load capacity for Memcached compared to Fastswap.

7 Related Work

Resource Disaggregation. Datacenters today suffer from poor average resource utilization due to overprovisioning [41, 56]. Resource disaggregation, an idea that dates back to 1990s [12, 23, 31, 45, 61], has gained renewed interest, thanks to the high performance of modern datacenter networks [16, 19, 28]. Its key idea is to break the server boundary and unstrand idle resources of remote servers over the network. Existing systems have demonstrated the feasibility of disaggregating various types of resources, including storage [32, 35, 37], accelerators [6, 20], and memory [10, 43, 52, 57]. Some other systems focus on improving the reliability of disaggregated datacenters [33, 64]. We focus on memory disaggregation (*i.e.*, remote memory) in this paper.

Kernel-based Remote Memory. To provide transparency to existing applications, the kernel-based approach leverages OS paging to access and manage remote memory. Most kernel-based systems build upon Linux, including Hermit. Infiniswap [29] is an early work that integrates Linux's swap subsystem with an RDMA-based block device backend. Later, Fastswap [10] leverages the lightweight `frontswap` interface of Linux to reduce overhead and offloads page reclamation to a dedicated core. Leap [43] improves Linux's prefetcher to achieve a higher local memory hit ratio. Canvas [59] isolates swap paths for co-running applications. The ongoing advances of Linux's virtual memory subsystem from the kernel community also benefit Linux-based remote memory. These include, but are not limited to multi-generational LRU [55], speculative page faults [4], maple-tree-based VMAs [2], and DAMON-based proactive page reclamation [5]. Finally, LegoOS [53] makes larger changes to both the kernel and hardware with the goal of achieving better performance through a clean-slate approach.

Library-based Remote Memory. Library-based approaches bypasses the OS to reduce kernel overhead and overcome the granularity restrictions imposed by paging. They trade application transparency for performance; application developers often have to modify their code to use new remote memory APIs. FaRM [25] and KVDirect [38] expose remote memory with an external key-value store interface which mismatches with the construction of existing applications. Distributed shared memory (*e.g.*, [40, 44]), on the other hand, provides an object-oriented interface that is more user-friendly. AIFM [52] proposes a higher-level abstraction of remote-able data structure, but but it still requires effort to port applications. Semeru [57], Mako [42], and MemLiner [58] are JVM-based remote memory runtimes, offering transparency to Java applications by co-designing the JVM with the kernel.

Hardware-accelerated Remote Memory. Another type of work proposes novel hardware designs, thereby unlocking new opportunities for optimizing remote memory. While Hermit focuses on the software layer, it could benefit from advances to the underlying hardware. PBerry [18] and Kona [17] overcome the granularity restriction of paging and enable cache-line-level remote memory access. Clio [30], StRoM [54], and RMC [11] reduce the expensive network traffic by offloading tasks into the customized hardware of the memory server. Finally, the emerging CXL bus [39] may lower the performance cost of accessing remote memory by delivering lower latency and near-local-DRAM throughput.

Multi-tiered Memory System. Recent research has focused on overcoming DRAM's capacity wall through the use of slower memory/storage devices—such as compressed memory, non-volatile memory (NVM), NVMe SSD, *etc.* Two examples of such systems are TMO [60] and HeMem [50], which transparently offload main memory to slower tiers. TMO focuses on developing a policy for determining which data to offload and how much, whereas Hermit aims at building an efficient offloading mechanism. HeMem targets improving throughput for batch applications. Therefore, it treats page offloading as a time-insensitive operation and performs it in the background. In contrast, Hermit optimizes for both batch and latency-critical applications by conducting page reclamation timely and proactively.

8 Conclusion

In this paper, we presented Hermit, a re-architected swap system that is based on adaptive, feedback-directed asynchrony. Our evaluation shows that Hermit significantly outperforms Fastswap (the state-of-the-art swap system) in real data center applications; it reduces the 99th percentile tail latency by 99.7% and improves the throughput by 1.24× on average. Hermit defies the conventional wisdom about kernel-based remote memory, demonstrating that it is possible to achieve both full transparency and high performance simultaneously.

Acknowledgements

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Michael Wei for his feedback. This work is supported by NSF grants CNS-1703598, CCF-1723773, CNS-1763172, CCF-1764077, CNS-1907352, CHS-1956322, CNS-2007737, CNS-2006437, CNS-2128653, CCF-2106404, CNS-2106838, CNS-2147909, CNS-2104398, ONR grant N00014-18-1-2037, research grants from Cisco, Intel CAPA, VMware, and Samsung, and a gift from Amazon.

References

[1] gdnsd - an authoritative-only dns server. https://gdnsd.org/.

[2] Introducing maple trees. https://lwn.net/Articles/845507/.

[3] NVMe over fabrics. http://community.mellanox.com/s/article/what-is-nvme-over-fabrics-x.

[4] Speculative page faults. https://lwn.net/Articles/851853/.

[5] Using damon for proactive reclaim. https://lwn.net/Articles/863753/.

[6] Virtual gpu (vgpu) | nvidia. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[7] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[8] Wikipedia networks data. http://konect.uni-koblenz.de/networks/, 2020.

[9] Apache cassandra. https://cassandra.apache.org, 2021.

[10] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[11] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, pages 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[12] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[13] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, Apr. 2018. USENIX Association.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[15] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

[16] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

[17] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. *Rethinking Software Runtimes for Disaggregated Memory*, pages 79–92. Association for Computing Machinery, New York, NY, USA, 2021.

[18] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS '19, pages 127–135, New York, NY, USA, 2019. Association for Computing Machinery.

[19] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets-XVI*, pages 164–170, 2017.

[20] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[21] T. Chen and C. Guestrin. extreme gradient boosting for applied machine learning. https://xgboost.readthedocs.io/en/latest/, 2021.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[23] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, pages 19–es, USA, 1994. USENIX Association.

[24] C. Delimitrou and C. Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, jul 2018.

[25] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[26] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[27] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[28] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[29] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[30] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pages 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[31] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, oct 2010.

[32] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP ≈ RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, Feb. 2020. USENIX Association.

[33] S. Kadekodi, F. Maturana, S. Athlur, A. Merchant, K. V. Rashmi, and G. R. Ganger. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, Carlsbad, CA, July 2022. USENIX Association.

[34] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, pages 185–201, 2016.

[35] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

[36] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, pages 317–330, 2019.

[37] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding rack-scale disaggregated storage. In *HotStorage*, 2017.

[38] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[39] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[40] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[41] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Big Data*, pages 2884 – 2892, 2017.

[42] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, 2022.

[43] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[44] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, pages 291–305, 2015.

[45] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters. In *European Conference on Parallel Processing*, 2003.

[46] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[47] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[48] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, pages 1–16, 2014.

[49] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[50] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[51] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[52] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, Nov. 2020.

[53] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[54] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[55] The OCP Foundation. Multi-generational lru: the next generation. https://lwn.net/Articles/856931/.

[56] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *EuroSys*, 2020.

[57] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, Nov. 2020.

[58] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, 2022.

[59] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

[60] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[61] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. *SIGPLAN Not.*, 46(7):205–216, Mar 2011.

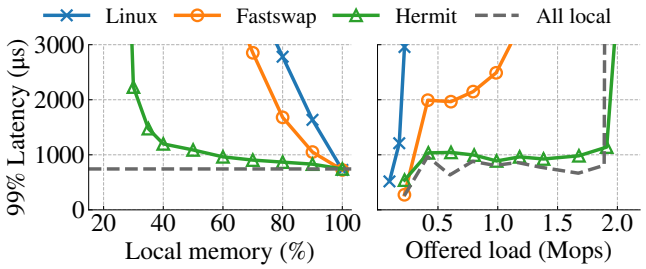
[62] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[63] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *SOSP*, pages 195–211, 2021.

[64] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

A Tail Latency of Linux-Based Applications

In this section, we evaluate Hermit using the vanilla Linux-based Memcached as opposed to the Shenango-enhanced Memcached (§6.2). Figure 18(a) shows 99th percentile latency with fixed load (1 Mops) and varying local memory ratios. Figure 18(b) shows latency with a fixed local memory ratio (70%) and varying load. Hermit still significantly outperforms other baseline systems. The results show a similar trend to the results of Shenango-enhanced Memcached.



(a) Under 1 Mops load. (b) Under 70% local memory.

Figure 18: For the vanilla Linux-based Memcached, Hermit still significantly outperforms other baseline systems.

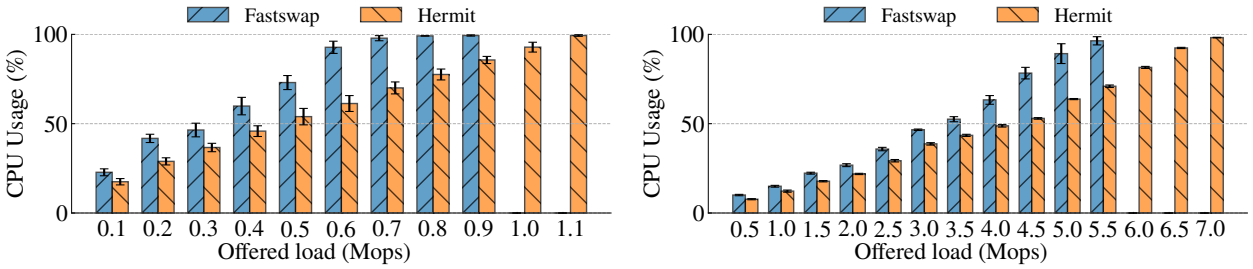
B CPU Usage of Other Applications

This section reports and compares the CPU usage of SocialNet and Gdnsd running on Fastswap and Hermit under the same setting as in Figure 8. Figure 19 shows the results. Thanks to its efficient swap design, Hermit consistently uses 10%–40% fewer CPU cycles than Fastswap, even though it invokes more reclaim threads.

C Tail Latency in Other Percentiles

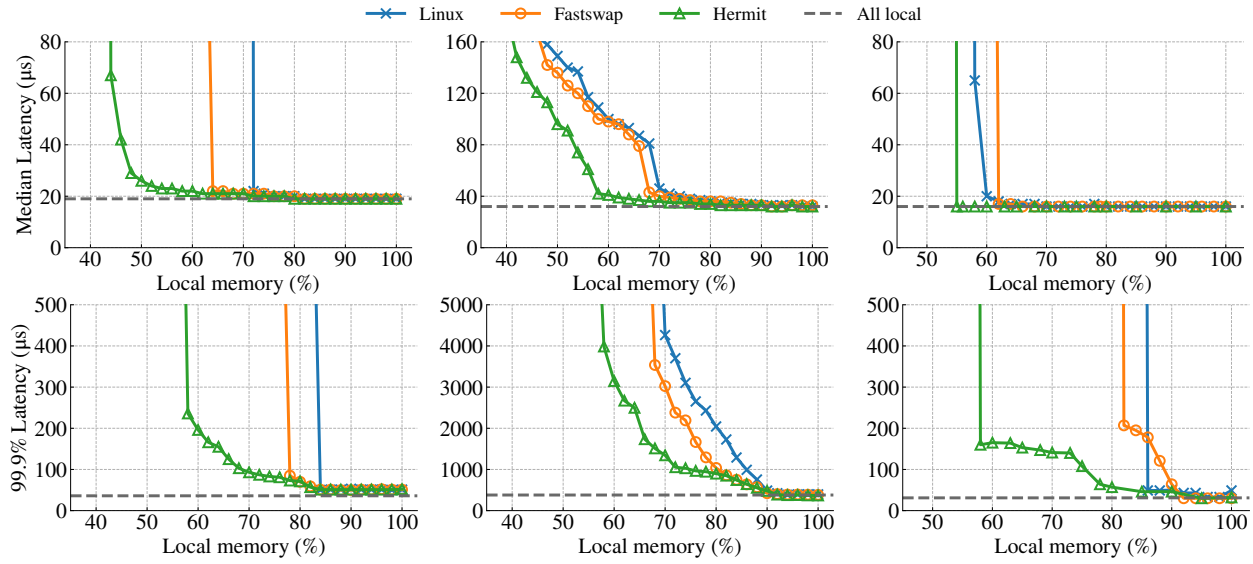
This section reports the median and 99.9th percentile latency of all three latency-critical applications. Figure 20 depicts the results when running applications with a fixed load and varying local memory ratios. The results exhibit a similar trend to Figure 7. All three systems sustain low median latency when local memory is not too scarce, while Hermit slightly outperforms Fastswap and Linux. When local memory continues to decrease, applications have to spend more CPU cycles on frequent remote memory accesses. The CPU congestion consequently ramps up the median latency. Thanks to the CPU-efficient swap design, Hermit's median latency increases slower than Fastswap, allowing applications to serve higher load, particularly when local memory is scarce. With regards to 99.9th percentile latency, Hermit again significantly outperforms Fastswap and Linux. It enables applications to put on average 20% more working set in remote memory without violating the tail latency agreement.

Next, we repeated the experiment shown in Figure 8 by fixing the local memory ratio to 70% and measured the median and 99.9th percentile latency of applications with varying load (see Figure 21). Hermit is able to deliver low median latency close to the ideal setup and much lower tail latency. Since 99.9th percentile latency is more susceptible to direct page reclamation, Fastswap experiences significant performance degradation once its single dedicated core gets saturated. Hermit, in contrast, is able to offer high load (> 60% compared to the ideal all-local setup) and maintain low 99.9th percentile latency under 70% local memory ratio.



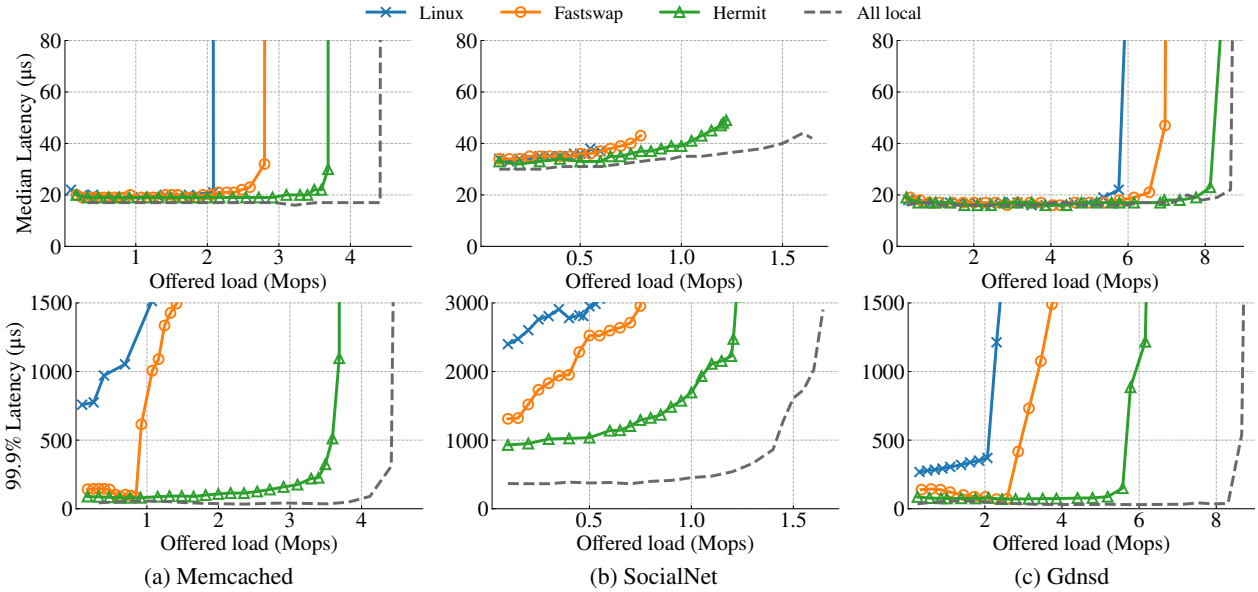
(a) SocialNet (b) Gdnsd

Figure 19: For SocialNet and Gdnsd, Hermit still saves 10%–40% CPU cycles under varying load compared with Fastswap, which is the key enabler it can achieve low tail latency under high load.



(a) Memcached (1.5 Mops) (b) SocialNet (0.75 Mops) (c) Gdnsd (4 Mops)

Figure 20: Hermit still significantly outperforms Fastswap and Linux in terms of median and 99.9% latency under the same load and varying local memory ratio.



(a) Memcached (b) SocialNet (c) Gdnsd

Figure 21: Hermit also achieves significantly lower median and 99.9% latency than Fastswap and Linux under 70% local memory ratio and varying load. As we used a closed-loop load generator for SocialNet, it reaches the maximum load capacity before its median latency spikes.

NetRPC: Enabling In-Network Computation in Remote Procedure Calls

Bohan Zhao
Tsinghua University

Wenfei Wu
Peking University

Wei Xu
Tsinghua University

Abstract

People have shown that in-network computation (INC) significantly boosts performance in many application scenarios include distributed training, MapReduce, agreement, and network monitoring. However, existing INC programming is unfriendly to the normal application developers, demanding tedious network engineering details like flow control, packet organization, chip-specific programming language, and ASIC architecture with many limitations. We propose a general INC-enabled RPC system, NetRPC. NetRPC provides a set of familiar and lightweight interfaces for software developers to describe an INC application using a traditional RPC programming model. NetRPC also proposes a general-purpose INC implementation together with a set of optimization techniques to guarantee the efficiency of various types of INC applications running on a shared INC data plane. We conduct extensive experiments on different types of applications on the real testbed. Results show that using only about 5% or even fewer human-written lines of code, NetRPC can achieve performance similar to the state-of-the-art INC solutions.

1 Introduction

The recent programmable switches like Barefoot Tofino [16] and Cisco Silicon One [5] can execute user-specified stateful packet processing at line rate. The evolution has sparked a surge of proposals to offload application functions into the network. The trend is called *in-network computation (INC)*.

INC has been widely applied in various applications including distributed ML training [11, 22, 29, 31, 37], cache [19, 25], agreement [6, 18, 40], and network monitoring [12, 17, 26, 27]. The tremendous bandwidth and low latency on switches lead to huge performance gains. For example, ATP [22] accelerates distributed training throughput by 38% $\sim$ 66%; P4xos [6] reduces the end-to-end delay by more than 90%; NetCache [19] improves throughput by 3-10 times compared with a host-only software solution. However, developing INC applications involves too much arcane knowledge in networking that is far from application programmers' (we refer to them as *users* in this paper) expertise and willingness to learn.

First, the INC program centers on individual packets. Users need to handle network functions such as packet parsing, flow table installation, forwarding, routing, reliable transmission, and congestion control as part of the application.

Second, users need to learn chip-specific languages like P4 [3] and NPL [28]. Even the high-level programming models like Lyra [7] and C3 [20] still focus on packet processing and require too much network knowledge (e.g., transmission windows and protocol fields) for software engineers.

Third, users need to understand low-level chip design details and limitations. Familiar data types and operations like floating points are missing, and users have to design approximations [13, 22, 31] manually. Even harder, users need to place their program on a pipeline of *stages* with isolated memory and deal with limitations, like once-only memory access per stage and a limited number of tables and rule entries.

Last but not least, users need to *statically* decide switch memory layout, table/register arrangement, etc., as the switch hardware can only modify them at boot time. Therefore, users need to reset the switch to start/remove an INC application, causing minute-level service interruption.

As a result, existing projects use INC only as a single application accelerator instead of a shared infrastructure. Even a simple application involves thousands of lines of code on both switches and hosts (Table 4 in Section 6). The development and operation difficulties prevent wide INC adoption.

In comparison, traditional software uses two abstraction layers to decouple application code from network details: 1) a *Socket layer* providing connection/session management, resource sharing, reliable communication, and byte stream abstraction; and 2) a *remote procedure call (RPC) layer* providing high-level data types and call interfaces. In the popular gRPC framework [10], users write a language-independent *interface definition language (IDL)* (e.g., `protobuf` [9]) specifying types of parameters and return values, and the gRPC compiler generates client and server *stubs* that users can integrate into application code. The stubs automatically marshal/un-

marshal arguments and handle underlying Socket connections. RPCs prove to be a powerful interface to build modern distributed systems. Unfortunately, neither layer exists in INC, leaving tedious network details to user applications.

We propose NetRPC to add both missing layers — an *INCLayer* and an *RPCLayer* — to bridge application programming and network packet processing, allowing users to leverage INC features to develop a diverse set of distributed applications using the familiar RPC interfaces.

The RPCLayer provides a high-level RPC interface. It is built on gRPC with two extensions: *INC-enabled data types (IEDTs)* and a NetFilter. IEDTs include basic types like integers and floating points, and collections like arrays and maps. Users define RPC services using the same protobuf language, just replacing vanilla gRPC types with IEDTs to allow NetRPC to recognize and process these data fields. In addition, users provide a NetFilter to specify the computation with INC, in terms of five *reliable INC primitives (RIPs)*. RIPs implement high-level operations on IEDTs such as arithmetics, reading/writing a map/array of arbitrary size, and synchronization primitives. RIPs also guarantee reliability, i.e., under various network conditions, RIPs eventually complete as long as the client/server processes survive.

RPCLayer also provides automatic data parallelism for calls with large arguments. NetRPC breaks up a call into subtasks, executes these subtasks concurrently, and sends out multiple *concurrent flows*. We offer it as a built-in feature to save programmers from handcrafting concurrent flows or co-flows to fully utilize the 100+ Gbps links in INC switches.

Analogous to the Socket, the INCLayer handles all flows from the RPCLayer. In addition to the basic guarantees of the Socket-like connection, reliable transport and congestion control, the INCLayer implements the RIPs using a set of protocols involving both the INC switches and the end-hosts.

We build NetRPC as a *general* INC-enabled RPC system. This is different from existing INC projects that only need to find one workaround for the switch hardware limitations as they target only a single application. The first design trade-off we need to make is between *generality (i.e., how programmable the network is)* and *simplicity (i.e., how easy it is to program it)*. Instead of building yet another general INC language, NetRPC chooses to provide only the necessary set of network-independent primitives and the simple NetFilter specification. Observing INC projects in the past ten years, we find only a handful successful types (Section 3.1). We design the primitives so that users can easily develop applications of all these types and enjoy the INC performance boosts.

New challenges for NetRPC include, from low level to high level: 1) efficiently managing the switch memory and pipeline stages to support the high-level array and map types; 2) hiding the switch hardware limitations from high-level programs; 3) supporting reliable transmission for different INC scenarios; 4) running multiple INC applications concurrently on a shared data plane; and 5) allowing users to define INC operations for their applications in the familiar gRPC abstraction.

We have many innovative designs to solve the above challenges. 1) Using a fallback mechanism, the end-host agents can take over all cases that the INC switches fail to handle; 2) Using an INC-compatible transport protocol, we can correctly handle packet retransmission and congestion control, maintaining both correctness and throughput; 3) Adapting a novel memory management scheme, we map from *keys* to unified 32-bit *logical addresses* that further map to switch *physical addresses*, allowing us to optimize the switch memory management much like normal caches; 4) By providing only a limited interface NetFilter, we abstract all obscure hardware limitations into a single high-level limitation (i.e., the primitives NetFilter supports).

We implement NetRPC using a testbed with two Barefoot Tofino [16] switches and eight machines. Using four non-trivial applications (Paxos, network monitoring, distributed training, and MapReduce) as examples, we show that 1) we reduce the *line of code* (LoC) on the end host to about 1/20, using less than two dozen network-related LoC per application; 2) NetRPC code is completely the same as vanilla gRPC code; and 3) we can offer the same or even better INC speedup.

In summary, our contributions include:

1) As a programming interface, NetRPC is the first framework to integrate INC acceleration into the RPC framework, reducing the bar of INC adoption in software.

2) As an INC system, NetRPC proposes a set of INC primitives applicable to different INC application types and innovative design elements to efficiently implement them, including reliable transport, memory management, and synchronization, as well as enabling a multi-application INC data plane.

3) Using four common INC application types on a real testbed, we demonstrate that we can offer the same INC performance boost with far fewer lines of code.

2 Related Work

Most existing INC applications make a network-software co-design. Even with the "network programming languages", users still have to handle many network engineering details.

Network-software co-design of INC. People have recently demonstrated many promising INC-accelerated applications, such as NetCache [19] and distCache [25] for caching, P4xos [6], NetChain [18] and NetLock [40] for agreement, SwitchML [31], SHARP [11], and ATP [22] for distributed ML training, and ElasticSketch [38], SilkRoad [26] and Sonata [12] for network monitoring. These solutions are similarly constructed as the network-software co-design — user interfaces, customized protocols, switch programs, rule installation, and endpoint agents — to achieve full-stack optimization and higher switch resource efficiency.

Chip-specific Programming Languages. People have proposed several chip-specific programming languages [3, 28,

32, 33] to support data plane customization. Existing programming languages are tightly coupled with corresponding ASICs. For example, Trident-4 [28] only supports NPL, while P4 programs can run on Tofino and Silicon One. P4 [3], arguably the most popular one for recent INC solutions, follows a *reconfigurable match table* (RMT) architecture. P4 programs first define packet headers and corresponding parsers and then process extracted header fields in a pipeline. Programmers must specify the actions on header fields, persistent switch registers at each pipeline stage, and drive actions by match-action tables. Also, users must define a *deparser* to reconstruct the packet for forwarding.

High-level network programming abstractions. There have been efforts to simplify the INC programming. E.g., Lyra [7] offers a one-big-pipeline abstraction that allows programmers to express their intent with simple statements; NCL [20] imports a window-based abstraction over packets as the basic processing units. μP4 [34] provides a lightweight logical architecture that abstracts away the structure of the underlying hardware pipelines for better program composition. NetVRM [42] allows developers to virtualize switch memory with a few modifications to existing P4 code. Chipmunk [8] adopts a domain-specific program synthesis technique to generate faster packet-processing code at the cost of longer compilation time. However, these high-level abstractions still revolve around networking details, such as (de)packetization, connection maintenance, and protocol stacks. The semantic gap between the software and network programming model is still a significant obstacle for ordinary software developers.

3 Design Overview

We design NetRPC to allow software developers to enjoy the performance benefits of INC without tedious network programming. We want NetRPC to be general enough to support typical INC application scenarios.

3.1 INC Application Types

INC accelerates applications primarily in two ways: optimizing bandwidth usage (reducing the *number of bytes* to servers) or reducing latency (removing the server from the round trip). People have proposed many INC applications. Table 3.1 summarizes the four types of applications.

The first two types handle large data sets with optimizing bandwidth as the main goal: (1) synchronous aggregation (SyncAtgr) for distributed machine learning (ML) training; (2) asynchronous aggregation (AsyncAtgr) for general MapReduce-type applications. The difference between these two types is that SyncAtgr aggregates only a fixed-sized array (e.g., the gradient updates) and works in iterations, i.e., we can proceed only *after* all clients send the updates. In contrast, AsyncAtgr aggregates over an arbitrary number of keys as they come in and allows accessing results at any time.

The other two types only use small data, with the main goal to optimize latency by avoiding sending packets to the server: (3) key-value cache (KeyValue) that require frequent queries and responses; and (4) Voting (Agreement) that involve counting votes from different clients until reaching a threshold. Unlike (1) and (2), each request is small, but the challenge is how to achieve a latency smaller than client-to-server RTT by not involving the server at all.

3.2 Challenges and Solution Overview

Providing a reliable data stream for general INC application types. Different from traditional networks, there are *side effects* when packets go through an INC switch, such as updating a map. Thus, when a packet goes through a switch twice in retransmission, the computation is no longer *idempotent*, violating the computation correctness. Prior solutions are application-specific, e.g., ATP [22] requires explicit server ACKs. It works in SyncAtgr, but not in the other three types because involving the server defeats latency optimization. We design an efficient and general retransmission mechanism that maintains the per-flow state on the switch using only a few bits in switch memory. We also design an effective flow and congestion control protocol (Section 5.1).

Making "normal path" efficient: Supporting memory-efficient arrays and maps on INC switches. Arrays and maps are core data structures in many applications, and INC significantly accelerates operations on them with parallel element processing. E.g., training applications use arrays to store the aggregated gradients, and monitoring applications keep the aggregates in a map, one key per metric. In both cases, the switch can add up all values in parallel. Existing systems either require pre-determined encoding of keys (e.g., knowing all the keys at compile time) or waste precious switch memory and packet header space to store the long keys. We leverage the host agents to generate a *two-level mapping* from keys of arbitrary lengths to a unified 32-bit *logical address space* and then map it to the switch physical memory. We also design a cache management algorithm running on the server agent to improve switch memory utilization efficiency (Section 5.2.2).

Making "corner cases" correct: Hiding switch hardware limitations from the upper-level program. We still need to handle switch hardware limitations. Our key idea is to use all *host agents* as a fallback mechanism. The host agents emulate all switch operations in software and thus can always provide correct INC results to the `RPCLayer` regardless of the switch's ability or resource. NetRPC supports two kinds of fallbacks: 1) arithmetic overflows that may happen in floating-point computation and accumulations (Section 5.2.1); and 2) insufficient memory on the switch (Section 5.2.2).

Supporting multi-application data plane. Prior arts support only a single application, and the life span of the switch program does not exceed that of the application. How-

Table 1: Four Common INC Application Scenarios and Primitives They Need

Type	Applications and Existing Systems	IEDT	Primitives
SyncAgtr	Distributed ML training (ATP [22], SHARP [11], SwitchML [31])	Array	`Map.get`, `Map.addTo`, `Map.clear`, `CntFwd`
AsyncAgtr	MapReduce (ASK [2], NetAccel [23], Cheetah [36])	Map	`Map.get`, `Map.addTo`, `Stream.modify`
KeyValue	Cache (NetCache [19], DistCache [25]), Monitoring (ElasticSketch [38])	Map	`Map.get`, `Map.addTo`
Agreement	Synchronization (P4xos [6], NetChain [18], NetLock [40])	Integer	`Map.get`, `Map.addTo`, `Map.clear`, `CntFwd`

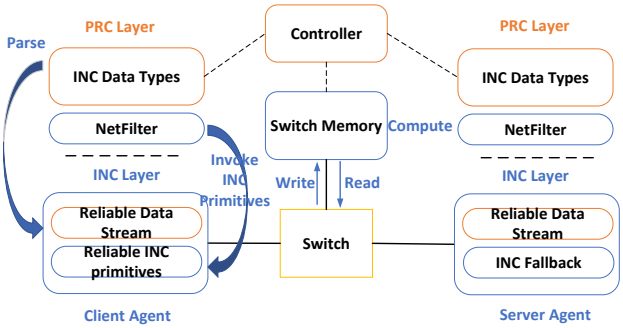


Figure 1: NetRPC system architecture.

```
1  import "netrpc.proto"
2  message NewGrad {
3    netrpc.FPArray tensor = 1;
4  }
5  message AgtrGrad {
6    netrpc.FPArray tensor = 1;
7  }
8  service Training {
9    rpc Update(NewGrad) returns (AgtrGrad)
         {} filter "agtr.nf"
10 }
```

Figure 2: Example `protobuf`: gradient updates

```
1  { //agtr.nf
2    "AppName": "DT-1",
3    "Precision": 8,
4    "get": "AgtrGrad.tensor",
5    "addTo": "NewGrad.tensor",
6    "clear": "copy",
7    "modify": "nop",
8    "CntFwd": {
9      "to": "ALL",
10     "threshold": 2,
11     "key": "ClientID",
12   },
13 }
```

Figure 3: Example `NetFilter`: gradient updates

ever, the RPC servers are long-running daemons, and server start/stop/restart events are common. It is prohibitively expensive to reset the switch on such events. We solve the problem with three designs: 1) letting all applications share the same set of RIPs; 2) sharing the same set of switch memory blocks among applications by partitioning the key spaces among them; 3) providing three choices of memory eviction behaviors to fit different applications (Section 5.2.2).

Interface `INCLayer` primitives with `RPCLayer` without breaking `protobuf` abstraction. Users need to tell NetRPC *what* to process in INC and *how* to process them. We need to add the INC specification to `protobuf` language, but we decide *not* to change the language to keep the learning curve low for users. Thus, we design the `NetFilter` as a configuration instead of a program. We only allow users to specify a fixed set of RIPs with at most one instance for each kind as a filter to process arguments and return values. The limitation simplifies NetRPC design yet still allows implementing all four common types of INC applications (Section 4).

The NetRPC contains a controller, host agents, and switch programs as in Figure 1. The system-wide *controller* is a dedicated process that handles registration and name lookups at initialization, while at runtime, it manages configurations on both switches and host agents. The *host agents* run on each client/server. Each host agent maintains a fixed number of connections (configurable) with the switch, even without running tasks. These connections are essential for the reliable communication (Section 5.1). A single *switch program* starts each INC switch at boot time and executes all primitives. The switch receives configurations from the controller to run applications without resetting the switch program (to avoid interrupting the network). If the switch fails to handle a primitive due to resource or functionality limitations, the primitive execution falls back to the server agents.

4 RPC Layer in NetRPC

In this section, we first introduce the NetRPC programming interface using gradient aggregation in the distributed training application as a concrete example. Then we briefly introduce interface implementation in the `RPCLayer`.

`protobuf` definition. Like in vanilla gRPC, users first provide a `protobuf` definition that compiles into the client and server stubs. Figure 2 shows an example `protobuf` file. The `messages` are user-defined types, and `service` is the RPC definition using `messages` as arguments and return values. The only modification to vanilla `protobuf` is the `filter` clause allowing users to provide the `NetFilter` file name (see below).

NetRPC data types. Users declare all variables that they want to process in INC using *INC-enabled data types (IEDTs)* defined by NetRPC. E.g., line 3 and 5 in Figure 2 defines variables (both `tensor`) as a `netrpc.FPArray` (floating point

```
shared_ptr<Channel> channel =
    CreateCustomChannel(server_ip,
    InsecureChannelCredentials());
unique_ptr<Stub> stub_(NewStub(channel));
void PushPull(double* data, int length) {
  NewGrad request;
  AgtrGrad reply;
  ClientContext context;
  request.mutable_tensor()->mutable_data()
        ->Add(data, data+length);
  Status status = stub_
        ->Update(&context, request, &reply);
  memcpy(data, reply.tensor().data(),
        length * sizeof(double))
  train(data);
}
```

Figure 4: Client program to use the RPC

Table 2: NetRPC Primitive Semantics

Primitive	Args	Semantics
`Map.addTo`	stream	map[stream.key] += stream.value
`Map.get`	stream	stream.value = map[stream.key]
`Map.clear`	empty	map[stream.key] = 0
`Stream.modify`	op,para	stream.value = op(stream.value, para)
`CntFwd`	key,th,tgt	cnt[key]++; if cnt[key] == th then forward(tgt) else drop

array) IEDT. Optionally, user can add normal gRPC data fields to the same messages, and NetRPC simply passes them to the server without processing.

Collections (`Array` and `Map`) are core data types in NetRPC. The item value can be integers or floating points, and keys can be integers or strings. NetRPC enables 1) automatically applying the user-defined `NetFilter` on *every* value in these collections and 2) accessing the *global INC map* using keys.

Life of a NetRPC call. In NetRPC, when a client initiates a call, the *client stub* marshals the arguments and sends them through one of two channels: messages with IEDT through the INC channel established by the per-host *client agent* and normal messages through the original gRPC Socket. In this paper, we only focus on the *data streams* in the INC channel. The underlying `INCLayer` processes the data stream and optionally interacts with the *INC map*. The INC map is a NetRPC abstraction of unlimited global memory addressable using keys or array indices. INC map is implemented on both switches and host agents (in Section 5.2.2). The return path is similar: the *server stub* marshals the return value and sends it through either the INC channel or the normal Socket.

The `NetFilter` and reliable INC primitives (RIPs). In addition, users need to specify their INC operations. Here, we have a choice in terms of what kind of operations NetRPC should provide. We want to find the sweet spot in the trade-off between generality and simplicity. We also want to provide a reconfigurable switch program to serve new applications. Therefore, we pick five primitives that we can compose together in a similar layout to implement existing types of INC operations (Section 3.1). Figure 5 displays this layout and its implementation on the switch. The users only need to provide configurations for these five primitives in their `NetFilter` file (Figure 3) to specify their INC operation of interest.

The `NetFilter` is a JSON configuration file. It contains a `AppName` that uniquely identifies an application, a `Precision` field that specifies the floating-point precision (number of digits after the decimal point). Lower precision allows INC to process more data without falling back to the host.

The more interesting part in `NetFilter` is the next five fields that allow users to provide arguments to RIPs, including three map-access primitives, `Map.addTo`, `Map.get`, and `Map.clear`, one data stream manipulation primitive, `Stream.modify`, and one synchronization primitive, `CntFwd`. Table 2 summarizes the parameters and semantics of these primitives.

`Map.addTo` *accumulates* data items from the stream to the map according to their keys/indices, and `Map.get` reads out the values of a specific key from the map. In Figure 3, we add the values of the `NewGrad.tensor` array to the INC map to aggregate the gradient, and on the return path, we read out the results from the INC map into the `AgtrGrad.tensor` array.

`Map.clear` defines how to clear a value from the INC map. In the example, `copy` means backing up the aggregates to the server before clearing it out to handle packet losses. We introduce other possible options in Section 5.2.2.

`Stream.modify` performs arithmetics on the stream. It only modifies the stream without accessing the INC map. In Figure 3, we set it to `nop`, as we do not modify streams. Table 8 in Appendix A lists all operations we support for `Stream.modify`.

The `CntFwd` is the most interesting primitive. It accumulates values on one or more keys (specified with `CntFwd.key`) in the INC map until the accumulator reaches the specified threshold (`CntFwd.threshold`). Then it forwards out the message to the destination(s) specified at `CntFwd.to`. The `CntFwd` primitive is essential to control both *how many* packets to forward to the clients/servers and *when* to forward them, and thus essential for SyncAgtr and Agreement applications. In this example, we set `key` to a single `ClientID`, meaning that we only need one counter for the number of unique clients who have sent gradient updates. In this case, only when exactly two unique clients have sent a stream, will the network aggregate the items and send back `AgtrGrad` to `ALL` clients.

There are other use cases for `CntFwd`. Setting the `CntFwd.threshold` to one makes the `CntFwd` behave as the `test&set` primitive in many instructions sets, useful to implement distributed mutual exclusion. Also, by providing a collection in the data stream, we can use a map of counters to track multiple votes in concurrent ballots, a widely-used functionality in distributed agreement protocols. `CntFwd` allows the switch to notify the clients only when enough votes arrive. Appendix D provides more examples of `CntFwd` primitive.

Table 1 summarizes the primitives used in each INC application type. Figure 5 illustrates a RIP pipeline running the example code in Figure 3. A SyncAgtr application pushes

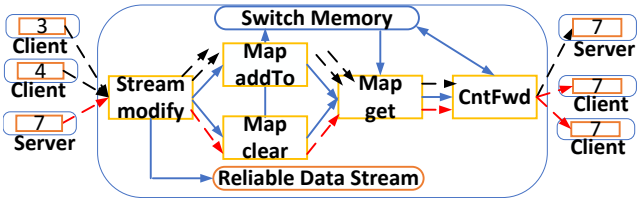


Figure 5: RIP pipeline in switches.

data into the network by its clients (black arrows) for on-switch aggregation and on-server backup. The server sends back the computation results (red arrows) to clients and clears the switch memory. The same switch program (and the RIP pipeline) completes all INC functions in the round trip without reconfiguring the switch.

Using RPC calls to build the application. With the `protobuf` and `NetFilter` definition, the remaining process is exactly the same as normal gRPC. The `protobuf` compiler generates client and server stubs, and users include stubs in their application. Figure 4 provides the client code using the RPC service defined in Figure 2. Note that the code is completely identical to vanilla gRPC, hiding INC details from the users.

Automatic data parallelism on RPCs with large arguments. There could be multiple concurrent NetRPC applications/channels and procedure calls in the runtime. The stub submits calls as tasks to the host agents. A task contains the application data (i.e., arguments or returned values, encoded as protobuf messages), the metadata (e.g., network program configurations), and the routing information.

The host agents maintain a thread pool of worker threads to process tasks. NetRPC automatically partitions the task into subtasks and dispatches them onto multiple worker threads for load balance. The worker threads serve the subtasks in their queue on a First-Come-First-Serve (FCFS) basis. The worker threads serialize the subtask's data into a sequence of packets (Appendix B.1) and send them over the user-level network stack we implement using DPDK [15].

Limitation of RIP abstraction. RIP abstraction targets simplifying programming in general INC scenarios where different INC applications regularly start and stop to share the infrastructure. It lacks logical semantics like looping and branching and thus can not implement complex algorithms (e.g., DHS [41]) or data structure (e.g., NetChain [18] queue). Adding more RIPs will extend the functionality but reduce the source available for each RIP. We leave further extensions and a customizable set of RIPs as future work.

5 INC Layer in NetRPC

`INCLayer` provides a reliable layer to support RIPs. There are two design objectives: 1) efficiently utilize INC switch resources to support a multi-application data plane with full INC performance boost; 2) provide an end-host software-based fallback mechanism to support reliable byte streams and INC primitives for the `RPCLayer`. As a result, `RPCLayer` can safely assume that the data stream is delivered reliably, and the `NetFilter` is fully executed in various network conditions.

In this section, we first introduce how we can build a general reliable data stream abstraction. Then we introduce the essential RIPs, including map access, arithmetics, and `CntFwd`. Finally, we briefly introduce the switch implementation.

5.1 Reliable Data Stream

Encoding IEDTs into sequences of packets. Client agents receive data streams containing multiple key-value pairs from the `RPCLayer`. Then the client agent encodes these pairs into separate packet headers using a user-level networking stack written in DPDK [15] and sends them out. Each packet contains a fixed number of key-value pairs (32 in the current setting), a sequence number, a *global application ID* (GAID), as well as other state information we will introduce in this section. Figure 14 in Appendix B.1 illustrates the packet header structure. The packet can optionally contain the normal payload with non-INC types for the application. NetRPC only processes the key-values encoded in the header.

Idempotent packet retransmission. In case of packet loss, traditional transport simply retransmits the lost packet. However, INC complicates the retransmission because the retransmitted packet can cause *side effects* on the switch, such as incrementing a map value again. In other words, naive retransmission is not *idempotent* and may lead to incorrect INC results. Switches need extra information to detect which packets are retransmitted. Traditional networking doctrine tells us that we shall avoid keeping states on switches. Thus, some INC designs choose to keep extra states on the sever [22] and let the server to ACK each packet. The ACK informs the switches about processed packets. This design requires the server to ACK every packet. It works on applications like gradient aggregation, where the INC is primarily used for reducing *server bandwidth* by only forwarding the results to the server to ACK while completing aggregation on the switch only. However, this design does not fit applications like KeyValue and Agreement, as server ACKs defeat the purpose of latency reduction via sub-RTT switch response.

To design a general protocol, we observe that 1) we send/receive all INC flows using host-agents under NetRPC control; 2) the INC switches have a relatively large memory, and the retransmission states are almost negligible compared to the INC map. Thus, we can safely keep the per-flow state on the switch as long as we can limit the number of agent flows.

We further design our protocol to minimize per-flow switch memory usage, allowing a switch to only keep a *bit array* of size w_{max} per flow, where w_{max} is the max sending window size. The switch initializes all bits to 1. Every packet contains a sequence number (`seq`), and a `flip` bit that is set to

$(seq/w_{max})\%2$. On receiving the packet, the switch checks the $(seq\%w_{max})$-th bit in the bit array. If the bit is the same as the `flip` in the packet header, the switch considers it as a retransmitted packet and thus skips updating the INC map. Otherwise, the packet is new, and the switch sets its corresponding bit to the `flip` and processes the packet normally.

We show that this simple protocol guarantees idempotent execution, i.e., 1) a packet's first appearance flips the bit, and 2) a packet's later appearance (retransmission) equals the bit. We prove it by *induction*. For sending window 0, all `flip` bits in packets are 0s, and the switch bitmap is all 1s. Since each packet only sets the bitmap to 0 once, at the end of window 0, all bits become 0. Then assuming the two properties hold for window $t-1$, we show that they still hold for window t. Recall that the client agent sends out the i-th packet in window t (denoted as P) only after the i-th packet in window $t-1$ (denoted as P') is ACKed. Therefore, when P first appears, P' should already set the bit as P''s `flip`. As P and P' are opposite in `flip`, P's first appearance flips the bit. P's later appearance would not flip the bit, and the window controls packets out of the window not to appear (and falsely flip the bit) between P's appearances. Thus, the two properties hold in window t. By induction, it is correct for all sending windows.

We can use $N \times w_{max}$ bits of switch memory to support N concurrent reliable flows on each agent. We experimentally set $w_{max} = 256$ and find it sufficient to achieve a per-flow bandwidth of 20+ Gpbs.

Flow control and congestion control. Note that w_{max} is a fixed value. We still need to deal with the flow control and congestion control due to resource contentions on either the end hosts or switches. We use the same mechanism to handle both flow and congestion control by automatically setting a congestion window $cw \leq w_{max}$.

Traditional congestion control, like the one in TCP, relies on round-trip-time (RTT) and duplicate ACKs to adjust cw. However, in INC primitives like `CntFwd`, these signals may not reflect the real network congestion as the receiver needs to wait for the slowest sender before ACKing.

Thus, NetRPC adopts an ECN (i.e., explicit congestion notification)-based congestion control mechanism. The switches set the ECN when the ingress port length exceeds the threshold. Meanwhile, it writes the ECN information to the INC map under a special key. Thus, all retransmission packets carry ECN until cleared like other map values. This prevents ECN signal loss due to packet loss. Otherwise, the client agents adjust the cw using the same *additive increase multiplicative decrease (AIMD)* policy as priot arts [22]. Experiments show that this design allows multiple flows to achieve both high goodput and fairness in bandwidth sharing.

Other transport protocols. Several recent transport protocols have affected the design of `INCLayer`. MTP [35] proposes a message-based protocol to customize congestion control, load balancing and resource isolation for INC. However, it requires maintaining per-pathlet states on both packet headers and end hosts, importing extra overhead to hurt the system's performance. DCTCP [1] imports a more fine-grained congestion window adjustment based on the ECN proportion. This approach is inapplicable to INC scenarios because we have to count the maximum number of ECNs in a single (i.e., the most congested) path instead of the total ECN proportion due to incast. This consumes more resources on the switch and will reduce the stream goodput, so we utilize AIMD to simplify implementation and will extend the protocol in future work (Section 7).

5.2 Reliable INC Primitive Designs

5.2.1 Computation and arithmetic overflows

Floating point arithmetic by quantization. INC switches only have limited 32-bit arithmetic functionality, yet INC applications like training require floating-point (FP) arithmetic. The standard practice uses quantization to fit the FP numbers into 32-bit integers (aka, fixed-point numbers). NetRPC quantizes an FP value in the client agent by multiplying it with a scaling factor (the `precision` field in `NetFilter`) and maps the value back to FP before handing it to the `RPCLayer`.

Handling overflows. While people have shown that the precision loss might not cause problems in many applications, 32-bit fixed-point numbers do not offer enough *representable range* in many cases, and thus overflow is unavoidable. Even without FP numbers, just using the `Map.addTo` to accumulate values may also lead to overflow. Thus, we need a way to handle occasional overflows.

When the switch detects an overflow during computation, it sets the overflowed value to `MAX_INT` or `MIN_INT` and forwards the packet normally. When a host agent receives a packet with `MAX_INT`/`MIN_INT` value, it suspects there is an overflow [1] and gives up the result. Then the client agents mark and resend these overflow packets, causing the switch to skip the processing and directly forward them to the server agent. The server agent computes the correct result using 64-bit integers or FP numbers in software.

Fallback on network fabrics without INC support. A similar fallback mechanism works when there are no programmable switches or data-plane resources reach capacity. If the controller fails to assign the INC application to any switch, the server agent will execute RIPs in software using the same switch failure handling mechanism. Therefore, the application is guaranteed to derive the correct results with the transparent fallback, only losing performance benefits from INC.

[1] Strictly speaking, there is one possibility of a false positive where the result is exactly `MAX_INT`/`MIN_INT`. The false positive only slightly affects performance leading to an extra retry, but not correctness.

5.2.2 Memory: INC map-access primitives

Memory address spaces in `INCLayer`. The `RPCLayer` supports maps with arbitrary keys, while the `INCLayer` only provides a 32-bit *logical* address space per application. The client agent hashes keys with different types and lengths into the 32-bit address space. We handle all collisions by putting the colliding keys into the payload to bypass the switch INC and let the server agent to process them. We choose not to use a larger logical space as we find it sufficient to support multiple applications with acceptable collisions. A short address saves bits in packets, increasing the effective bandwidth.

`INCLayer` maps the 32-bit address space onto the *physical* address space on switches. Each physical address corresponds to a *register* on a switch. Switches may have different numbers of registers. E.g., the switch we use has about 160K registers available per pipeline stage, and we use eight stages to support map-related primitives.

It is not trivial to map the logical address to a physical switch address. The above hashing approach does not work here because switch registers are a valuable resource we want to make full use of, but when the utilization is high, the collision rate increases fast, causing many fallbacks to servers. In fact, we need to pack the physical memory tightly. Also, we want to avoid keeping the logical-physical address mapping on switches; otherwise, it wastes switch memory.

In some applications, such as distributed training (as in ATP [22]) in SyncAgtr, it is simple as every client has the same set of keys. Each of them only needs to sort the keys and give each key a sequence number. However, it does not work in general cases, such as AsyncAtgr, where each client might have a different set of keys.

Multiple clients of a single application. We solve the problem by letting the server agent, shared by multiple clients, decide and maintain the mapping for all its clients. The first time a client uses a new logical address, it sends packets to the server without INC. If there is switch memory available, the server agent will piggyback a mapping for this address on the returning ACK. Then the client can send subsequent packets with the physical address set in the packet for the switch. If the switch memory is full, the server will not return the mapping, and thus clients keep sending subsequent packets to the server without INC. With the method, we ensure all clients calling the same server use a consistent mapping.

Handling multiple applications. According to the applications' requests, the controller reserves switch memory at application registration time. When an application gets no switch memory, they fall back to using server agents. We use a simple FCFS policy for the static allocation among different applications and leave advanced memory scheduling as future work. Note that although the controller reserves memory at registration time, the actual allocation only happens when the clients plan to send out data streams. Thus we can avoid holding memory unnecessarily.

Cache replacement policies. The switch memory serves as a cache for certain keys, and we need a replacement policy at the server agent. We take an approximation to the *least-recently-used* (LRU) policy. Each client agent counts the uses of each logical address within a *cache update window*, and at the end of the window, they send the counter to the server, allowing the server to compute the most-used keys in the last window. Then in the next period, the server evicts less used values. We also evaluate other popular cache replacement policies in Section 6, and we show that this periodic counting-based LRU policy works well.

Optimization for synchronous aggregation. In addition to the general logical-physical mapping, we realize that the SyncAgtr (i.e., distributed training) applications like SwitchML [31] only require access to large continuous arrays. It is more memory-efficient to be able to allocate such arrays in a few *circular buffers* instead of many individual addresses. NetRPC supports such buffers of a fixed size of 256 keys.

Preventing switch memory leaks on host failures. Unlike existing INC designs that serve only a single application, NetRPC is a shared infrastructure supporting many applications. Thus, we need to take care of potential switch memory leaks resulting from the crashing of user programs or host machines before they can explicitly release the memory. We address this issue with a *two-level timeout* mechanism.

NetRPC processes a packet with an *admission rule* that checks the GAID. We keep a timestamp of the last time the rule runs for each GAID. The controller periodically polls the switch for these timestamps. If it finds a stale timestamp, it triggers the *first-level timeout* by notifying the server agent to retrieve the application's INC map. After a longer period, the server agent triggers the *second-level timeout*, sending the saved data items to the user-defined stub or deleting them if the stub no longer exists. As switch memory is small and precious, we want to reclaim it quickly with a small *first-level timeout*. However, the small timeout unavoidably introduces false positives, hurting the correctness of programs with low communication frequency, such as monitoring infrequent events. In fact, these applications will benefit little from INC anyways, and the timeout mechanism allows them to run just like normal applications. Servers have much larger memory and thus can keep user maps longer, providing the correctness of such programs similar to software.

The `Map.clear` primitive. The switch memory only supports `Map.addTo` instead of directly overwriting the value. Thus, to start a new accumulation (e.g., a new iteration of training, restarting a vote, etc.), the user program needs to execute three steps: 1) `Map.get` the accumulator value to the hosts, and 2) `Map.clear` the memory and 3) start to `Map.addTo` new values. However, there is a risk that the packets get dropped *en-routing* to the host. In this case, the memory is already cleared, so the value is permanently lost.

NetRPC provides different methods to prevent this loss, as

there is a latency-throughput tradeoff. We decide to allow users to choose from three clear policies in `NetFilter`.

1) [Copy]: The client-call stream first carries the map's value to the server, and then the return stream from the server will `Map.get` and `Map.clear` the values. Thus we guarantee the server has a backup in case the return packet is lost. This policy requires no extra switch memory at the cost of forwarding more data to the server and thus higher latency.

2) [Shadow]: The switches double memory allocation. The data stream uses two memory segments alternatively: `Map.get` from one and `Map.clear` the other. This approach reduces latency at the cost of doubling memory usage and thus is only suitable for latency-sensitive applications with few data items.

3) [Lazy]: The `Map.clear` primitive only lets the host agents to save the current value and let the switch to keep accumulating without clearing. The host agent subtracts the saved value to compute the accumulated value since the last clear. When the accumulator eventually overflows, we fall back to the server agent using the same overflow logic and clear the switch memory. If the application (e.g., voting) has a slow-increasing counter, lazy policy involves little overhead.

The multiple `clear` policies allow users to better customize their INC applications according to their SLA requirements and workload features. We compare the performance of the three policies in Section 6.

Implementation on the switch. We allow 32 key-value pairs per packet. We use four register groups per stage and 8 out of the 12 stages on the switch to implement the INC map access. This design fits the switch hardware limitation: a packet can only access each group of registers in the switch once per trip. For the same reason, we arrange `Map.get`/`Map.addTo` and `Map.clear` to execute in the opposite direction of a packet round trip. These primitives are organized in a flow chart on the switch pipeline (Figure 15 in Appendix C). Appendix D displays a number of example settings of `NetFilter` in different application types.

5.2.3 Forwarding: the CntFwd primitive

The `CntFwd` primitive requires two extra pieces of logic in the switch. First, the switch needs to recognize the packet is a `CntFwd` packet, and then the packet goes through the normal map-access pipeline to increase and read the values in the accumulator. We implement different computation logic for the accumulator (`test&set` or accumulate) by applying different match-action tables according to the `CntFwd.threshold`. Finally the packet enters the last stage on the switch that decides whether to `drop`, `send`, or `multicast` the packet.

6 Evaluation

In this section, we show that NetRPC achieves the following desirable properties: 1) NetRPC supports four kinds of INC applications; 2) NetRPC significantly reduces the amount of application code; 3) NetRPC achieves the same performance as handcrafted INC applications; 4) NetRPC handles situations like packet loss, congestion, etc. In addition, we evaluate the effects of policy settings (clear and caching).

Table 3: Workload and Baseline in Experiments

App Type	App	INC Baselines	Dataset
SyncAgtr	Distributed Training	ATP [22] SwitchML [31]	ImageNet [14]
AsyncAgtr	WordCount	ASK [2]	Yelp [39]
KeyValue	Network Monitoring	ElasticSketch [38]	CAIDA Anonymized Internet Trace [4]
Agreement	Paxos	P4xos [6]	Synthatic workload

6.1 Experiment Settings

NetRPC implementation. We implement NetRPC switch logic on a 12-stage programmable switch. The NetRPC switch pipeline contains 32 read-write memory segments corresponding to the 32 key-value pairs in the NetRPC packet. Each memory segment contains 40k 32-bit units to restore INC states or the INC map. Depending on the service configuration, we vary packet lengths from 192 to 320 bytes.

NetRPC includes four modules: $\sim 4K$ lines of P4 code for the switch logic, $\sim 2K$ lines of Python code for the remote controller, $\sim 2K$ lines of C++ code as the plugin of gRPC++ [10], and $\sim 3K$ lines of C++ code for the NetRPC end-host agents using DPDK. We also implement four types of INC applications with only 200 $\sim$ 500 lines of code each.

Testbed. We run NetRPC on a testbed of 8 GPU machines and two programmable switches. The devices form a dumbbell topology: two connected switches, each with four machines. In the experiment, we use "X-to-Y" to denote a topology with X clients and Y servers. The switch contains a Barefoot Tofino chip and provides 32 $\times$ 100 Gbps ports. Each machine has a Mellanox ConnectX-5 dual-port 100 Gbps NIC. Each machine is equipped with two NVIDIA GeForce RTX 2080Ti GPUs, 56 CPU cores at 2.20GHz, and 192GB RAM. The machines install NVIDIA driver 430.34, CUDA 10.0, Mellanox driver OFED 4.7-1.0.0.1, and Ubuntu 18.04.

Workloads and baselines. Table 3 shows the workloads and baselines we use. We run various typical models (VGG, ResNet, AlexNet) for SyncAgtr. We also implement each application's pure software version as baselines using DPDK.

6.2 Reducing User Code Complexity

We compare the user-written lines of code (LoC) of NetRPC applications with existing INC arts. Table 4 shows that NetRPC reduces the overall human-written code by over 97% in all four application types. To enable INC in an RPC, the application developers only need to configure the `NetFilter` to enable/disable RIPs on the switch without writing any switch

Table 4: LoC Comparisons: NetRPC vs. Prior INC Arts

	NetRPC		Prior INC Arts	
	Endhost	Switch	Endhost	Switch
SyncAggr	173	13	3394	5329
AsyncAggr	166	26	3278	4258
KeyValue	162	26	898	2360
Agreement	1453	26	5441	931

Table 5: Microbenchmark on Basic INC Functions

Metrics	NetRPC	Prior Arts	DPDK
SyncAgtr Goodput(Gbps)	50.55	46.44 (ATP)	40.11
AsyncAgtr Goodput(Gbps)	72.31	73.96 (ASK)	45.88
Voting Delay(μs)	20	22 (P4xos)	92
Monitor Delay(ms)	3.52	3.26 (ElasticSketch)	4.05
Packet Processing Capacity(Mpps)	>1000	>1000	83.47

code. `NetFilter` results in a huge LoC reduction (12-21 LoCs in NetRPC v.s. 931-5329 in prior arts). On the host, NetRPC also reduces the LoC of host programs by 95%, 95%, 73%, and 82% for the four applications compared with existing INC applications, as NetRPC users only write code to process data-stream as call arguments, avoiding the tedious network functions like (de)packetization, reliability, etc.

6.3 End-to-end Application Performance

Distributed ML training. We set up eight worker machines for this evaluation. We use two existing INC frameworks, SwitchML [31] and ATP [22], and a pure software solution, BytePS, as baselines. We implement the NetRPC version on BytePS with only 500 LoC modifications. All INC versions use a single parameter server (PS), while the software version uses eight to provide enough throughput.

Figure 6 shows the average training speed per worker. We have the following observations: 1) INC solutions outperform non-INC ones for most models because they avoid incast to the PS. NetRPC, ATP, and SwitchML are 42%, 42%, and 11% faster than BytePS in VGG16; 2) For all models, NetRPC performs similar to ATP (97% to 100% of ATP), and at most 28% faster than SwitchML; 3) the training speeds on ResNet are similar because they are computation-intensive, and communication does not affect the overall performance much.

We believe the performance gain in NetRPC over existing systems is from the automatic parallel streams. As a side benefit, NetRPC uses only a single port (or one pipeline) instead of recirculation like ATP or SwitchML. Using fewer ports is essential for the multi-application data plane. SwitchML-RDMA [30] uses even more pipelines by chaining four pipelines together to achieve a performance gain over ATP. We do not adopt the design because resource efficiency is one of our key considerations.

Paxos. We use NetRPC to implement a Paxos [21] consensus system, offloading the leader and vote counting functions to switches. The implementation only contains about 700 LoC changes. We use an INC baseline, P4xos [6], and two software ones, `libpaxos` [24] and DPDK Paxos [6]. We run two proposers, two acceptors, and three learners in all cases.

Figure 7 summarizes the results on both throughput and 99th-percentile latency to achieve one consensus. Key findings include: 1) NetRPC achieves a maximum throughput of 503K messages/second, 12% higher than P4xos, and 7.86× and 4.93× higher than the two software solutions. INC solutions are much faster because they offload packet processing to the switch to alleviate the CPU bottleneck on servers. NetRPC has higher throughput than P4xos because it only sends the final results to the learners, reducing the workload on servers and saving the traffic on learner links. 2) The 99th-percentile latency of NetRPC is 311 ms and 96 ms shorter than software but 42 ms higher than P4xos. This is because we choose not to run the acceptors on switches like P4xos and thus need an extra round trip to the software acceptor. We believe the location and replication flexibility of the acceptor is a worthwhile tradeoff for the extra latency, given that it is still much faster than pure software.

6.4 Micro-benchmarks

To better understand NetRPC performance impact, we conduct a series of micro-benchmarks, focusing on INC-related functions only. We also use both prior INC arts and pure software DPDK implementation as comparison baselines.

Throughput. We perform SyncAtgr and AsyncAtgr on a 2-to-1 testbed and measure the *sender goodput*, using ATP and ASK as INC baselines.

The first row in Table 5 shows the result. NetRPC offers 9% higher throughput than ATP. The reason is that NetRPC does not apply recirculation (we use `copy` policy in this experiment) as ATP and SwitchML, which costs extra ports or pipelines on the switch . Instead, it relies on the parallel message sending (Section 4) to increase the goodput. Not surprisingly, both INC solutions outperform software solutions, e.g., NetRPC offers 26% higher goodput than pure DPDK. In fact, the end-to-end training results (42% faster, see Section 6.3) show an even larger improvement than the micro-benchmark, as in SyncAtgr, the shorter latency also improves GPU utilization as we spend less time waiting for the aggregation results.

The second row shows the goodput in AsyncAtgr. NetRPC achieves a similarly high throughput as ASK (about 73 Gbps). Unlike SyncAtgr, the keys count as part of a valid payload in this case, and thus the goodput is higher. Both INC solutions have 37% higher throughput than the pure DPDK.

Latency. We measure the average latency for the two latency-sensitive applications: Agreement and KeyValue, using P4xos voting and ElasticSketch [38] (monitoring) as baselines. The third row in Table 5 shows the average voting latency. Both NetRPC and P4xos outperform DPDK with a 76% latency

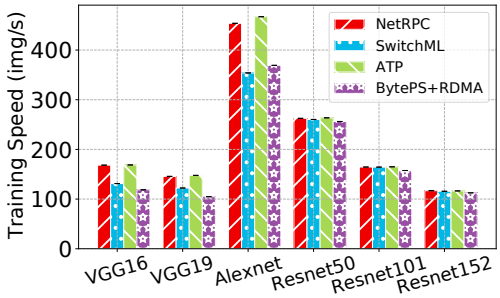
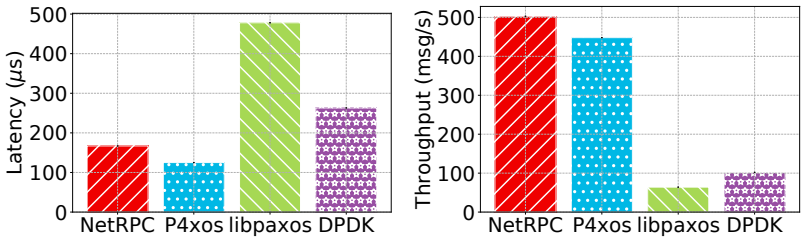


Figure 6: Deep Learning Training Speed



(a) 99th-Percentile Latency (b) Throughput

Figure 7: End-to-end Performance of Paxos Systems.

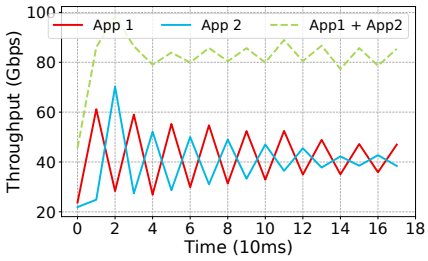


Figure 8: Congestion Control: Fairness

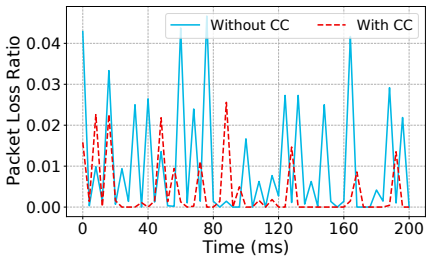


Figure 9: Congestion Control: Packet Loss

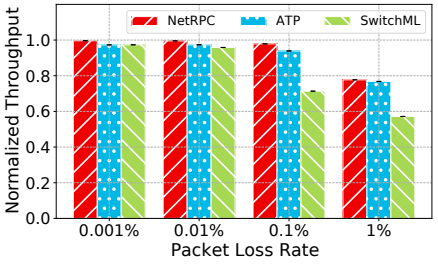


Figure 10: Packet Loss Rates vs. Throughput

reduction. NetRPC and P4xos offer similar latency, showing that NetRPC abstraction layers do not add extra latency.

The last two rows of Table 5 compares performance for KeyValue types, specifically in *flow counting*. Both NetRPC and ElasticSketch have lower latency than DPDK, by 13% and 20%, respectively. Notably, the last row of Table 5 also shows a 10× packet processing capacity increase from DPDK. NetRPC is about 0.26 ms (9%) slower than ElasticSketch because we do not have the same application-specific optimization that avoids modifying packets. We believe a less-than-10% latency increase is a reasonable price to pay for the general programming model by omitting the optimization.

Congestion control performance. To evaluate the effects of congestion control in NetRPC, we concurrently run two applications: a SyncAggr and an AsyncAggr on the same data plane (i.e., the same switch, host, and links), each having two clients and one server. Figure 8 shows the throughput over a short time period. We observe that the throughput quickly converges within 200 ms, and the combined bandwidth reaches 77% to 89% of the 100Gbps link. Also, the two application fairly shares the available bandwidth. Figure 9 shows the packet loss ratio over a short time period with/without congestion control. We can see that our ECN-based congestion and flow control reduces packet loss by about 63%, as it automatically adjusts the sending window to avoid overwhelming both the link and the server agent (Section 5.1).

Reliability mechanisms. To evaluate how NetRPC handles packet losses, we inject packet losses at different rates to emulate unreliable network. We run three INC applications NetRPC, ATP, and SwitchML and verified that all three correctly handles packet loss. Figure 10 shows the normalized throughput. NetRPC performs retransmission correctly under packet loss, using on-switch states only. At a high loss rate, NetRPC has a more graceful performance degradation. Compared with the no-loss case, NetRPC, ATP, and SwitchML's throughput decrease by 22%, 23%, and 43%, respectively. With 1% loss, NetRPC shows significantly less performance degradation than SwitchML because it adopts out-of-order ACKs and thus learns and reacts to packet loss faster.

Handling overflows. We run SyncAggr under synthetic workload varying overflow ratios from 0.001% to 1%. Figure 11 plots the throughput vs. overflow ratios. In all experiments, we check the computation results to ensure that NetRPC detects and corrects the overflow as we expect. When the overflow ratio exceeds 0.1%, we notice throughput degradation due to the software fallback. NetRPC still achieves about 65 Gbps throughput at 1% overflows. Note that the overflow ratio in real workload is far less than 1% with a reasonable quantization scaling factor for floating-point numbers. In contrast, the pure software solution only achieves a max of 40 Gbps.

Performance of `clear` policies. NetRPC offers three ways to handle `Map.clear` in `NetFilter` (Section 5.2.2). We measure the performance of a 2-to-1 SyncAggr using three `Map.clear` policies, and Table 6 summarizes the results. `Lazy` policy performance depends on the ratio of arithmetic overflow, and we use three ratios of 0%, 1%, and 10%. `Copy` policy achieves the highest throughput without extra memory cost but also has the highest latency because it relies on servers to backup

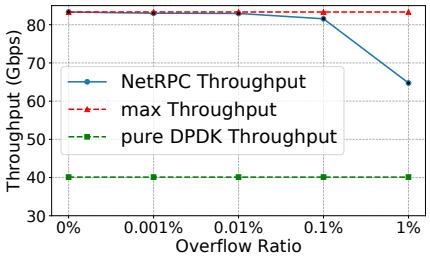


Figure 11: Overflow Ratio vs. Throughput

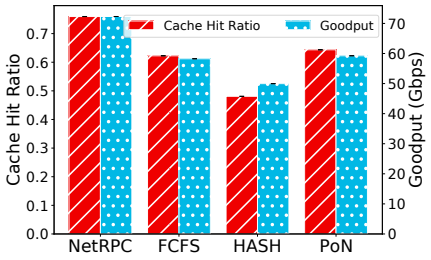


Figure 12: Caching Policy Comparison

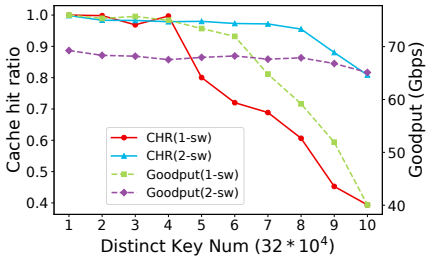


Figure 13: NetRPC on Two Switches

Table 6: Clear Policy Impact on Performance

	Latency	Memory	Throughput
copy	74μs	1x	83.11Gbps
shadow	24μs	2x	50.41Gbps
lazy (0%)	22μs	1x	83.31Gbps
lazy (1%)	23μs	1x	64.75Gbps
lazy (10%)	30μs	1x	34.82Gbps

Table 7: Concurrent Application Throughput and Latency

Metrics	1APP	4APP	4APP×5
Sync Goodput (Gbps)	50.55	24.88	24.84
Async Goodput (Gbps)	72.31	36.01	36.60
Goodput Sum (Gbps)	N/A	60.89	61.44
KeyValue Delay (ms)	3.52	3.56	3.85
Agreement Delay (μs)	20	21	24

the cleared states for reliability. `Shadow` policy offers a good latency of 24μs but doubles memory usage and has the lowest throughput because it needs to recirculate the packet and keep an extra copy. `Lazy` policy achieves both the highest throughput and lowest latency with no overflows. But as the overflow ratio increases, both metrics degrade. The actual accumulator overflow ratio depends on the data. Thus, we leave them as a user configuration in the `NetFilter`.

Cache policy. As we discuss in Section 5.2.2, a good cache policy alleviates traffic incast at the server and improves performance. We evaluate multiple cache policies. The experiment uses $32 \times 4K$ switch memory with 2-to-1 traffic. Comparison baselines are FCFS, hash-based caching (HASH), and Power of N (PoN). HASH policy uses the hash key as the index to address the switch memory (like ASK [2] and ATP [22]) and falls back to the server agent on hash collisions. PoN is a classic policy in sketches [38]: it only caches the hot keys whose hit number exceeds a threshold N and gives up caching when the switch memory is full. We tune the hyper-parameter N to maximize the performance experimentally.

Figure 12 shows the result. First, the CHR is positively correlated with the goodput, indicating the need for cache policy optimization. NetRPC's periodic cache update outperforms other cache policies by 18% $\sim$ 57% on cache hit ratio (CHR) and 22% $\sim$ 44% on goodput. HASH performs the worst because it ignores the locality of keys in the same packet: if some keys are cached, but their adjacent keys in the same stream are not due to hash collision, the entire packet will never hit the cache. PoN and FCFS behave similarly as they stop caching new hot keys if the cache has been fully filled. Compared with these baselines, NetRPC catches up the locality better and adapts to high-skewed key distribution better because it always caches the recent hot keys and periodically updates the switch cache to make up space for newer ones.

6.5 Multiple Concurrent Applications

An important goal of NetRPC is to support a multi-application data plane without switch rebooting. To evaluate the performance, we run multiple instances of all four application types in a 2-to-1 topology. We evaluate using three concurrency settings: 1) running a single application instance ("1APP "); 2) running one instance per type ("4APP"); and 3) running five instances per type ("4APP ×5") . Table 7 shows the total goodput and average latency. In all cases, we measure and report the throughput of SyncAgtr and AsyncAgtr and the latency of KeyValue and Agreement. In the 4APP ×5 case, we take the average of all instances of the measured type.

When concurrent applications increase from 4 to 20, we observe that the total bandwidth of SyncAgtr and AsyncAgtr stays roughly the same. Although KeyValue and Agreement do not use much bandwidth, they do contend for switch PPS (packets per second) and queue up in sending threads. The experiments show that small applications have little impact on bandwidth-heavy ones. We observe only a 20% latency increase compared to the 1APP case. These results demonstrate the successful resource sharing ability of NetRPC.

6.6 Running on Multiple Switches

Limited by available hardware, we only validate NetRPC's cross-switch capability with two-switches. We chain the two switches into a longer pipeline, and thus a packet can carry more key-value pairs. The NetRPC server agent decides which

key to put on which switch. We compare the performance of running 2-to-1 MapReduce on the testbed with one / two switches. We loop through the distinct keys multiple times, and thus a cache smaller than the number of distinct keys will suffer cache misses. Then we measure the CHR and the goodput varying with the number of distinct keys as an indicator of how well NetRPC is using memory on both switches.

Figure 13 shows the result. Each switch stores $M = 32 \times 40K$ values with distinct keys. We confirm that the goodput starts to drop at M using one switch, but $2M$ with two. The peak goodput decreases slightly with more switches (from 75 Gbps to 69 Gbps), mainly because of the increased host workload to encode more keys into the packet. Beyond the switch memory capacity, the goodput first decreases slightly (5.3% of peak throughput with $1.5M$ keys for one switch) and then dramatically (22% with $2M$ keys). This is because offering a 75 Gbps workload, there is little hope that the server CPU can handle many cache misses. Nevertheless, the two-switch setting shows a $1.63\times$ improvement over the one-switch case when handling $2.5M$ distinct keys, showing that NetRPC can efficiently utilize memory on multiple switches.

7 Conclusion and Future Work

In-network computation (INC) comes from software-defined networking (SDN), but INC is fundamentally different from SDN because it mainly provides *computation* service instead of *communication*. Thus, we need a new programming model for INC to *better describe computation*. We need high-level data structures, collections, memory, and procedure calls that center around end-hosts instead of packets, headers, tables, and pipelines that center around switches. On the other hand, we recognize that the INC data plane is still a shared network infrastructure, not an application-specific accelerator. Thus, both generality and multi-application support are essential.

NetRPC, to our knowledge, is the first framework that integrates INC into the familiar RPC programming model. NetRPC allows users to implement different types of INC applications using the familiar gRPC framework and run them on a single shared INC data plane. NetRPC achieves 97% of LoC deduction for INC applications and offers similar or better performance boosts than handcrafted systems.

Current NetRPC mainly focuses on *mechanisms* of INC + RPC integration. In future work, we will focus on *policies*, such as scheduling among different applications, efficient sharing between INC workload and other SDN or traditional network traffic, efficient end-host CPU, GPU, and INC co-scheduling. We will also explore NetRPC on more complex topologies, especially those with oversubscribed links. We will extend NetRPC congestion control with more fine-grained window adjustment. We will open source NetRPC on the publication of this paper to benefit the INC community.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[2] Anonymous. ASK: In-network aggregation service for key-value streams, 2022. https://anonymous.4open.science/r/ASK-80BF.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] Anonymized Internet traces, 2008. https://www.caida.org/data/passive/passive_dataset.xml.

[5] Cisco. One silicon, one experience, multiple roles, 2019. https://blogs.cisco.com/sp/one-silicon-one-experience-multiple-roles.

[6] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.

[7] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 435–450, 2020.

[8] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 150–160, 2019.

[9] Google. Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data, 2008. https://developers.google.com/protocol-buffers.

[10] Google. gRPC: A high performance, open source universal rpc framework, 2020. https://grpc.io/.

[11] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol SHArP: a hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10. IEEE, 2016.

[12] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[13] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21(79):1–31, 2004.

[14] ImageNet. Imagenet is an image database organized according to the wordnet hierarchy, 2022. https://www.image-net.org/.

[15] Intel. DPDK is the data plane development kit that consists of libraries to accelerate packet processing workloads running on a wide variety of cpu architectures., 2013. https://www.dpdk.org/.

[16] Intel. Barefoot tofino, 2020. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[17] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[20] George Karlos, Henri Bal, and Lin Wang. Don't you worry'bout a packet: Unified programming for in-network computing. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 99–107, 2021.

[21] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[22] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[23] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. The case for network accelerated query processing. In *CIDR*, 2019.

[24] General purpose Paxos library, 2013. https://bitbucket.org/sciascid/libpaxos.

[25] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[26] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[27] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[28] NPL. Open, high-level language for developing feature-rich solutions for programmable networking platforms, 2021. https://nplang.org/.

[29] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

[30] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[31] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[32] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[33] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132, 2013.

[34] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μP4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 329–343, 2020.

[35] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. Tcp is harmful to in-network computing: Designing a message transport protocol (mtp). In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 61–68, 2021.

[36] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2407–2422, 2020.

[37] Raajay Viswanathan and Aditya Akella. Network-accelerated distributed machine learning using mlfabric. *arXiv preprint arXiv:1907.00434*, 2019.

[38] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[39] Yelp. An all-purpose dataset for learning, 2022. https://www.yelp.com/dataset.

[40] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 126–138, 2020.

[41] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2285–2293, 2021.

[42] Hang Zhu, Tao Wang, Yi Hong, Dan RK Ports, Anirudh Sivaraman, and Xin Jin. NetVRM: Virtual register memory for programmable networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 155–170, 2022.

Table 8: Arithmetic Operations in `Stream.modify`

OP	Semantics
MAX	stream.value = max(stream.value, para)
MIN	stream.value = min(stream.value, para)
ADD	stream.value += para
ASSIGN	stream.value = para
SHIFTL	stream.value <<= para
SHIFTR	stream.value >>= para
BAND	stream.value &= para
BOR	stream.value |= para
BNOT	stream.value = ∼stream.value
BXOR	stream.value ^= para

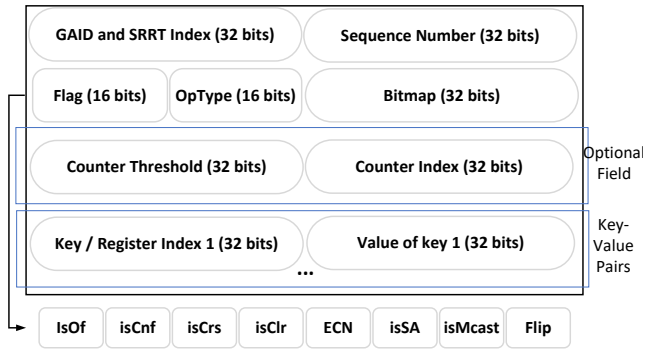


Figure 14: NetRPC Packet Format

A Arithmetic in NetRPC

We list the arithmetic operators of `Stream.modify` and their semantics supported by NetRPC in Table 8.

B NetRPC Protocol

B.1 Packet Format

The packet contains three kinds of fields. The key-value pairs encode the application data with the format of an array of <key/index, value> tuples; computation control fields encode the `NetFilter` configurations and guide the switch program for the computation; transport control fields maintain the channel connection.

Key-value pairs. Each NetRPC packet carries 32 key-value pairs. These pairs are either processed on the switch or the server agent by the selected primitives. The computation results are also carried back in the same position.

Computation Control Fields. The control flag bits contain the basic information about primitives selection. Current bits in use can indicate the following choices: whether any overflow happens (`isOf`); whether to use `CntFwd` (`isCnf`); whether to clear the target memory (`isClr`).

`OpType` indicates the type of arithmetic operation on key-value pairs. NetRPC supports various line-rate on-packet computation as we discuss in Appendix A. In `bitmap` field, the *i*-th bit in the bitmap indicates whether the switch should process the *i*-th key-value pair. The `CntFwd` fields only come into effect when the `isCnf` flag is set. `counter index` tells the switch which counter (register) to increase; when the register value equals to the `counter threshold`, the switch should forward the packet instead of dropping it.

Transmission Control Fields. Concurrent NetRPC connections (de)multiplex the network, and NetRPC distinguishes the flows by the `GAID`. On hosts, received packets are classified to the applications; on the switch, the `GAID` is also used for admission control. In NetRPC, each sending thread maintains a short-term connection to serve applications' calls/tasks and thus assigns a sequence number (starting from zero) for each packet. In addition, the reliability control requires sending threads to maintain a long-term connection (cross the tasks) with the switch. The field State Register of Reliable Transmission `SRRT` is the switch memory address to store the state, and the `flip bit` is the reliable state to store. Some bits in the `Control Flag` also controls the routing: whether the packet should cross the switch to the server agent (`isCross`); `ECN` indicates whether the switch is experiencing congestion (queue buildup); whether the packet comes from the server agent (`isSA`); whether to multicast the packet (`isMcast`).

Optimization. Some optional fields will be removed if unnecessary in the computation to improve the network bandwidth efficiency and the goodput. (1) If we address the key-value or value stream linearly to the switch memory, we can eliminate the key fields and indicate the starting index of the memory segment by the `counter index` field. (2) If the computation does not need `CntFwd`, we can eliminate the `CntFwd` fields.

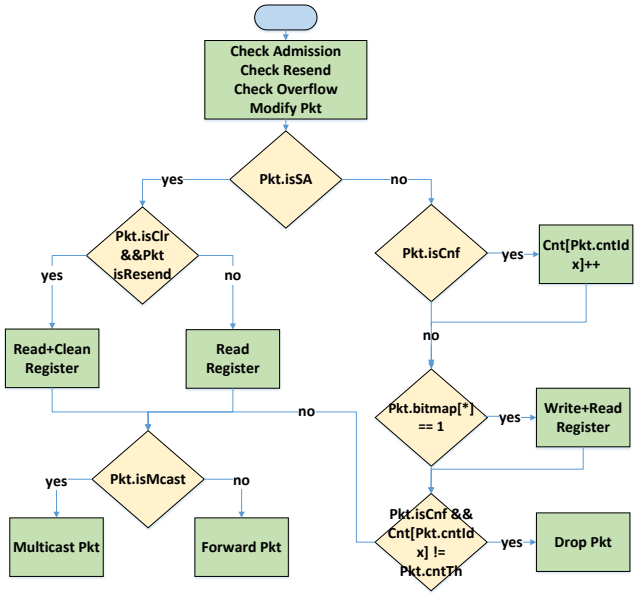


Figure 15: NetRPC Switch Logic

C Switch pipeline details

There is a 12-stage pipeline in our switch, and we use 8 to implement the map access primitives. The remaining four stages handle the reliable transmission, flow and congestion control, as well as `Stream.modify` and the `CntFwd` primitive. Figure 15 illustrate the flowchart for switch logic.

When the switch receives a NetRPC packet, it will first check whether the corresponding application (GAID) has registered. Unregistered packets will be forwarded as normal ones. Moreover, the switch checks whether it receives the packet for the first time. Otherwise, it avoids `Map.addTo`/`Map.clear` primitives on the switch memory but still `Map.get` values from registers into the packets. An overflow packet will be forwarded directly for fallback without on-switch processing.

For packets to the server, the switch first executes `Stream.modify` and `CntFwd` if required, then processes key-value pairs in the packet: `Map.addTo` the switch registers and `Map.get` the computation results back to replace the value. The switch drops those packets that enable `CntFwd` but do not reach the threshold and forwarded/multicast the rest packets.

For packets from the server, the switch first `Map.get` register values into the packet and then decides whether to clear the corresponding registers. The switch will forward/multicast the packets according to control flags and routing rules.

D NetRPC Implementation Examples

We enumerate some NetRPC implementation of classic INC applications: MapReduce, lock server, and network monitoring in Figure 16 to 24.

```
import "netrpc.proto"
message ReduceRequest {
  netrpc.STRINTMap kvs = 1;
}
message ReduceReply {
  string msg = 1;
}
message QueryRequest {
  string msg = 1;
}
message QueryReply {
  netrpc.STRINTMap kvs = 1;
}
service MapReduce {
  rpc ReduceByKey (ReduceRequest) returns
      (ReduceReply) {} filter "reduce.nf"
  rpc Query (QueryRequest) returns (
      QueryReply) {} filter "query.nf"
}
```

Figure 16: RPC Service Definition of Distributed MapReduce

```
{ //reduce.conf
  "AppName": "MR-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "ReduceRequest.kvs",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "NULL",
  },
}
{ //query.conf
  "AppName": "MR-1",
  "Precision": 0,
  "get": "QueryReply.kvs",
  "addTo": "nop",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "NULL",
  },
}
```

Figure 17: `NetFilter` of Distributed MapReduce

```
shared_ptr<Channel> channel =
    CreateCustomChannel(server_ip,
    InsecureChannelCredentials());
unique_ptr<Stub> stub_(NewStub(channel));
pair<string,int>* MapReduce(pair<string,
    int>* data, int length) {
    ReduceRequest request1;
    ReduceReply reply1;
    ClientContext context1;
    for(int i = 0; i<length; i++){
        (*request1.mutable_kvs()->
            mutable_map())[data[i].first]
            = data[i].second;
    }
    Status status = stub_->ReduceByKey(&
        context1, request1, &reply1);
    QueryRequest request2;
    QueryReply reply2;
    ClientContext context2;
    stub_->Query(&context2, request2, &
        reply2);
    int sz = reply2.mutable_kvs()->
        mutable_map()->size(), idx = 0;
    pair<string,int>* output = new pair<
        string,int>[sz];
    for(auto it: (*reply2.mutable_kvs()->
        mutable_map())){
        output[idx].first = it.first;
        output[idx++].second = it.second;
    }
    return output;
}
```

Figure 18: Client Stub for Distributed MapReduce

```
import "netrpc.proto"
message LockRequest {
  netrpc.STRINTMap map = 1;
}
message LockReply {
  string msg = 1;
}
message ReleaseRequest {
  netrpc.STRINTMap map = 1;
}
message ReleaseReply {
  string msg = 1;
}
service Lock {
  rpc GetLock (LockRequest) returns (
      LockReply) {} filter "lock.nf"
  rpc Release (ReleaseRequest) returns (
      ReleaseReply) {} filter "release.nf"
}
```

Figure 19: RPC Service Definition of Distributed Lock Server

```
{ //lock.conf
  "AppName": "LS-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "nop",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 1,
    "key": "LockRequest.kvs",
  },
}
{ //release.conf
  "AppName": "LS-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "nop",
  "clear": "copy",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "ReleaseRequest.kvs",
  },
}
```

Figure 20: `NetFilter` of Distributed Lock Server

```
shared_ptr<Channel> channel =
    CreateCustomChannel(server_ip,
    InsecureChannelCredentials());
unique_ptr<Stub> stub_(NewStub(channel));
void BlockingLock(string* lockTarget, int
    length) {
    LockRequest request1;
    LockReply reply1;
    ClientContext context1;
    for(int i = 0; i<length; i++){
        (*request1.mutable_kvs()->
            mutable_map())[lockTarget[i]]
            = 1;
    }
    Status status = stub_->LockSend(&
        context1, request1, &reply1);
    /* critical section */
    ReleaseRequest request2;
    ReleaseReply reply2;
    ClientContext context2;
    for(int i = 0; i<length; i++){
        (*request2.mutable_kvs()->
            mutable_map())[lockTarget[i]]
            = 0;
    }
    stub_->Release(&context2, request2, &
        reply2);
}
```

Figure 21: Client Stub for Blocking Lock Acquire and Release

```
import "netrpc.proto"
message MonitorRequest {
  netrpc.STRINTMap kvs = 1;
  string payload = 1;
}
message MonitorReply {
  string payload = 1;
}
message QueryRequest {
  string message = 1;
}
message QueryReply {
  netrpc.STRINTMap kvs = 1;
}
service Monitor {
  rpc MonitorCall (MonitorRequest) returns
       (MonitorReply) {} filter "monitor.
      nf"
  rpc Query (QueryRequest) returns (
      QueryReply) {} filter "query.nf"
}
```

Figure 22: RPC Service Definition of Network Monitoring

```
{ //monitor.conf
  "AppName": "MON-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "MonitorRequest.kvs",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SERVER",
    "threshold": 0,
    "key": "NULL",
  },
}
{ //query.conf
  "AppName": "MON-1",
  "Precision": 0,
  "get": "QueryReply.kvs",
  "addTo": "nop",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "NULL",
  },
}
```

Figure 23: `NetFilter` of Network Monitoring

```
1  shared_ptr<Channel> channel =
       CreateCustomChannel(server_ip,
       InsecureChannelCredentials());
2  unique_ptr<Stub>stub_(NewStub(channel));
3  pair<string,int>* MonitorRPC(string*
       metrics, int length) {
4      MonitorRequest request1;
5      MonitorReply reply1;
6      ClientContext context1;
7      for(int i = 0; i<length; i++){
8          (*request1.mutable_kvs()->
               mutable_map())[metrics[i].
               first] = 1;
9      }
10     request1.payload = "Hello";
11     Status status = stub_->MonitorCall(&
           context1, request1, &reply1);
12     if (status.ok()) {
13       cout << reply1.payload << endl;
14     }
15     QueryRequest request2;
16     QueryReply reply2;
17     ClientContext context2;
18     stub_->Query(&context2, request2, &
           reply2);
19     int sz = reply2.mutable_kvs()->
           mutable_map()->size(), idx = 0;
20     pair<string,int>* output = new pair<
           string,int>[sz];
21     for(auto it: (*reply2.mutable_kvs()->
           mutable_map())){
22         output[idx].first = it.first;
23         output[idx++].second = it.second;
24     }
25     return output;
26 }
```

Figure 24: Client Stub for RPC with Monitoring

Bolt: Sub-RTT Congestion Control for Ultra-Low Latency

Serhat Arslan* *Stanford University*
Yuliang Li *Google LLC*
Gautam Kumar *Google LLC*
Nandita Dukkipati *Google LLC*

Abstract

Data center networks are inclined towards increasing line rates to 200Gbps and beyond to satisfy the performance requirements of applications such as NVMe and distributed ML. With larger Bandwidth Delay Products (BDPs), an increasing number of transfers fit within a few BDPs. These transfers are not only more performance-sensitive to congestion, but also bring more challenges to congestion control (CC) as they leave little time for CC to make the right decisions. Therefore, CC is under more pressure than ever before to achieve minimal queuing and high link utilization, leaving no room for imperfect control decisions.

We identify that for CC to make quick and accurate decisions, the use of precise congestion signals and minimization of the control loop delay are vital. We address these issues by designing Bolt, an attempt to push congestion control to its theoretical limits by harnessing the power of programmable data planes. Bolt is founded on three core ideas, (i) Sub-RTT Control (SRC) reacts to congestion *faster* than RTT control loop delay, (ii) Proactive Ramp-up (PRU) *foresees* flow completions in the future to promptly occupy released bandwidth, and (iii) Supply matching (SM) explicitly matches bandwidth demand with supply to maximize utilization. Our experiments in testbed and simulations demonstrate that Bolt reduces 99th-p latency by 80% and improves 99th-p flow completion time by up to 3× compared to Swift and HPCC while maintaining near line-rate utilization even at 400Gbps.

1 Introduction

Data center workloads are evolving towards highly parallel, lightweight applications that perform well when the network can provide low tail latency with high bandwidth [5]. Accordingly, the Service Level Objectives (SLOs) of applications are becoming more stringent, putting increasing responsibility on network performance. To support this trend, the industry is inclined towards increasing line rates. 100Gbps links are already abundant, 200Gbps is gaining adoption, and industry standardization of 400Gbps ethernet is underway [24].

*Work done as a student researcher at Google

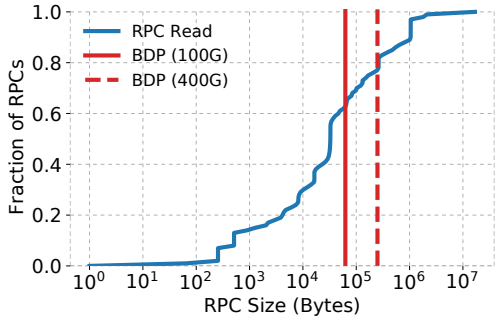


Figure 1: RPC size distribution for READ operations

With the increasing line rates, CC needs to make decisions with *higher quality and timeliness* over *a burstier workload*. We illustrate this based on a recent analysis of RPC sizes in our data centers with respect to BDP sizes at 100Gbps and 400Gbps (calculated using a typical base delay/RTT in data centers). Our findings are presented in Figure 1.

The fraction of RPCs that fit within 1 and 4 BDP increases from 62% and 80% at 100Gbps to 80% and 89% at 400Gbps. These RPCs are performance-sensitive to queuing and under-utilization. Ultimately, even a single incorrect or slow CC decision may end up creating tens of microseconds of tail queuing [12], or cause under-utilization [53] which prolongs the flow completion time by a few RTTs. Therefore, an increasing fraction of such RPCs raises the bar for the quality and timeliness of CC.

Concomitantly, at higher bandwidth, the workload becomes burstier and thus harder to control. Figure 1 also reveals that a 400Gbps link with just 40% load sees an RPC arrival or completion roughly every RTT! Hence, it becomes more difficult to control queuing and under-utilization as they arrive and finish quickly at RTT timescales. We expect these numbers to be even more challenging for upcoming workloads such as disaggregated memory and ML.

We identify two key aspects of CC that are important to address the challenges of achieving higher CC quality and timeliness on burstier workloads:

First, **granular feedback** about the location and severity of

congestion allows avoiding over/under-reaction [3]. A precise CC algorithm would receive the exact state of the bottleneck to correctly ramp down during congestion and ramp up during under-utilization. This congestion information would intuitively involve telemetry such as the current queue occupancy and a measure of link utilization [35]. Then, end-hosts would be able to calculate the exact number of packets they can inject into the network without creating congestion.

Second, the **control loop delay** is a determinant of how sensitive a control algorithm can be. It is defined as the delay between a congestion event and the reaction from the senders arriving at the bottleneck. Smaller the control loop delay, the more accurate and simpler decisions a control system can make [41]. The state-of-the-art CC algorithms in production are reported to work well to the extent their control loop delay allows [30, 35, 60]. However, even a delay of one RTT will be too long to tolerate for future networks because of the increasing BDPs [58]. We conjecture that the inevitable next step is to reduce the control loop delay to sub-RTT levels.

Fortunately, the flexibility and precision provided by programmable switches [7, 11, 22] allow designing new mechanisms to reduce the control loop delay and increase the granularity of control algorithms. These state-of-the-art switches can generate custom control signals to report fine-grained telemetry so that flows don't need to rely on end-to-end measurements for detecting congestion at the bottleneck link.

In this work, we present Bolt, our effort of harnessing the power of programmable data planes to design an extremely precise CC for ultra-low latency at very high line rates. Bolt collects congestion feedback with absolute minimum (sub-RTT) delay and ramps up flows proactively to occupy available bandwidth promptly. To achieve this, it applies the "packet conservation" principle [25] onto the traffic with accurate per-packet decisions in P4 [9]. Small per-packet `cwnd` changes, combined with the fine-grained in-network telemetry, help limit the effects of noise in the instantaneous congestion signal. With Bolt, end-hosts do not make implicit estimations about the severity and exact location of the congestion or the number of competing flows, freeing them from manually tuned hard coded parameters and inaccurate reactions.

The main contributions of Bolt are:

1. A discussion for the fundamental limits of an optimal CC algorithm with minimal control loop delay.
2. Description of 3 mechanisms that collectively form the design of Bolt – an extremely precise CC algorithm with the shortest control loop possible.
3. Implementation and evaluation of Bolt on P4 switches in our lab which achieves 86% and 81% lower RTTs compared to Swift [30] for median and tail respectively.
4. NS-3 [48] implementation for large scale scenarios where Bolt achieves up to 3× better 99th-p flow completion times compared to Swift and HPCC [35].

The remainder of the paper describes the rationale behind the design of Bolt in §2, design details in §3, and implementation insights in §4. Further evaluations and benchmarks are provided in §5 followed by practical considerations in §6. Finally, a survey of related work is presented in §7.

2 Towards Minimal Control Loop Delay

Timely feedback and reaction to congestion are well understood to be valuable for CC [42]. With Bolt, we aim to push the limits on minimizing the control loop delay that is composed of two elements: (1) *Feedback Delay* (§2.1) is the time to receive any feedback for a packet sent, and (2) *Observation Period* (§2.2) is the time interval over which feedback is collected before `cwnd` is adjusted. Most CC algorithms send a window of packets, observe the feedback reflected by the receiver over another window, and finally adjust the `cwnd`, having a total control loop delay that is even longer than an RTT [1, 10, 19, 30, 35, 60]. In this section, we describe both *Feedback Delay* and *Observation Period* in detail and discuss how these elements can be reduced to their absolute minimum motivating Bolt's design in §3.

2.1 Feedback Delay

There are two main types of feedback to collect for congestion control purposes: (i) *Congestion Notification* and (ii) *Underutilization Feedback*.

2.1.1 Congestion Notification

The earliest time a CC algorithm can react to drain a queue is when it first receives the notification about it. Traditionally, congestion notifications are reflected by the receivers with acknowledgments [1, 8, 30, 35, 42, 47, 60]. We call this the RTT-based feedback loop since the delay is exactly one RTT.

To demonstrate how notification delay affects performance, we run an experiment where the congestion notification is delivered to the sender after a constant configured delay (and not via acknowledgments). Setting this delay to queuing delay plus the propagation time in the experiment is equivalent to RTT-based control loops described above. The experiment runs two flows with Swift CC [30] on a dumbbell topology[1] where the second flow joins while the first one is at a steady state. The congestion signal is the RTT the packet will observe with current congestion. Figure 2 (left) shows the time to drain the congested queue for different notification delays. Clearly, smaller notification delay helps mitigate congestion faster as senders react sooner to it.

More importantly, in addition to traveling unnecessary links, traditional RTT-based feedback loops suffer from the congestion itself because the notification waits in the congested queue before it is emitted. Adding the queuing delay

[1] RTT is 8 μs and all the links are 100Gbps.

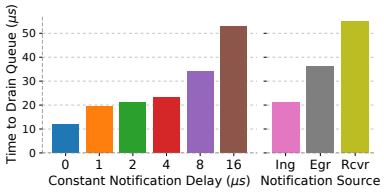
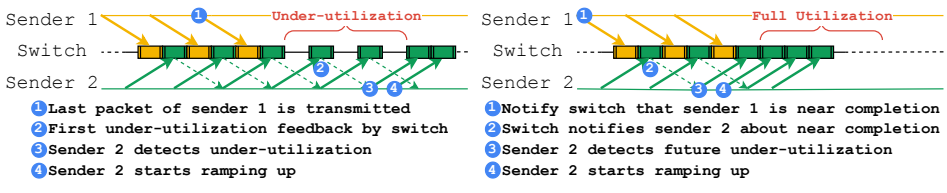


Figure 2: Effect of notification delay on queue draining time



(a) Reactive Ramp-Up **(b)** Proactive Ramp-Up

Figure 3: Under-utilization feedback

to the notification delay hinders tackling congestion even more. During severe congestion events, this extra delay can add multiples of the base RTT to the feedback delay [30].

To understand this more, we also measure the congestion mitigation time of scenarios where the notification is generated at different locations in the network in Figure 2 (right). "Rcvr" represents the RTT-based feedback loop where the congestion notification is piggybacked by the receiver. "Egr" is when the switch sends a notification directly to the sender from the egress pipeline, after the packet waits in the congested queue. "Ing" is when the notification is generated at the ingress pipeline, as soon as a packet arrives at the switch. As expected, generating the congestion notification as soon as possible improves performance by more than 2×.

Correspondingly, we stress that in order to reduce the notification delay to its absolute minimum, the congestion notification should travel directly from the bottleneck back to the sender without waiting in the congested queue.

2.1.2 Under-utilization Feedback

While flow arrival events add to congestion in the network, flow completion events open up capacity to be used by other flows. When a flow completes on a fully utilized link with zero queuing, the packets of the completing flow leave the network and the link will suddenly become under-utilized until the remaining flows ramp up (Figure 3a). As traffic gets more dynamic, such under-utilization events become more frequent, reducing the total network utilization. Therefore, in addition to detecting congestion, a good control algorithm should also be able to detect any under-utilization in order to capture the available bandwidth quickly and efficiently [44].

In practice, CC schemes deliberately maintain a standing queue under a steady state, so that when a flow completes, the packets in the queue can occupy the bandwidth released by the finished flow until the remaining flows ramp up [34, 40]. For example, while HPCC was designed to keep near-zero standing queue, the authors followed up that in practice, HPCC target utilization should be set to 150% to improve network utilization [36], which implies half a BDP worth of standing queue. Other CC schemes used in practice also maintain standing queues by filling up the buffers to a certain level before generating any congestion signal [1, 30, 60].

Figure 4 demonstrates how Swift behaves upon a flow completion when a long enough standing queue is not maintained. There are two flows in the network[2] and one of them completes at $t = 200\mu$s. The remaining flow's `cwnd` takes about 25 RTTs to occupy the released bandwidth as per the additive increase mechanism in Swift. During this time interval, under-utilization happens despite the non-zero queuing at a steady state. This under-utilization can also be observed when there are a larger number of flows if the standing queue size is not adjusted appropriately [53].

Ideally, any remaining flow should immediately capture the `cwnd` of the completing flow without under-utilizing the link. Therefore we conclude that an optimal congestion control algorithm would detect flow completions early enough, **proactively**, to ramp up as soon as the spare capacity becomes available (Figure 3b).

2.2 Observation Period

In addition to the feedback delay, the total control loop delay is usually one RTT longer for window-based data center CC schemes. Namely, once the sender adjusts its `cwnd`, the next adjustment happens only after an RTT to prevent reacting to the same congestion event multiple times. We call this extra delay the *observation period* and illustrate it in Figure 5.

Once-per-window semantics is very common among CC schemes where the per-packet feedback is aggregated into per-window observation. For example, DCTCP [1] counts the number of ECN markings over a window and adjusts `cwnd` based on this statistics once every RTT. Swift compares RTT against the target every time it receives an ACK but decreases `cwnd` only if it has not done so in the last RTT. Finally, HPCC picks the link utilization observed by the first packet of a window to calculate the reference `cwnd` which is updated once per window. As a consequence, flows stick to their `cwnd` decision for an RTT even if the feedback for a higher degree of congestion arrives immediately after the decision.

Updating `cwnd` only once per window removes information about how dynamic the instantaneous load was at any time within the window. This effect, naturally, results in late and/or incorrect congestion control decisions, causing oscillations between under and over-utilized (or congested) links when flows arrive and depart. Consider the scenario[2] in Figure 6

[2]The dumbbell topology from Figure 2 (RTT: 8 μs, 100Gbps links).

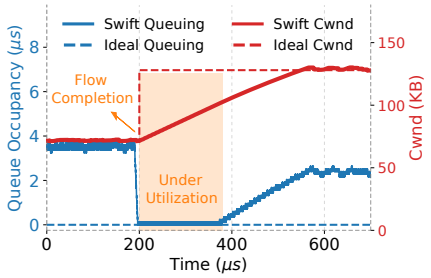


Figure 4: `cwnd` of the remaining Swift flow and queue occupancy after a flow completion.

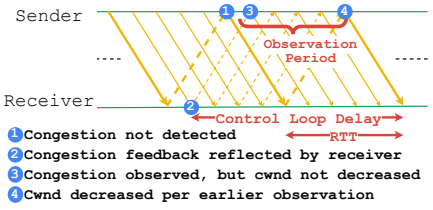


Figure 5: Observation period adds up to an RTT to the control loop delay.

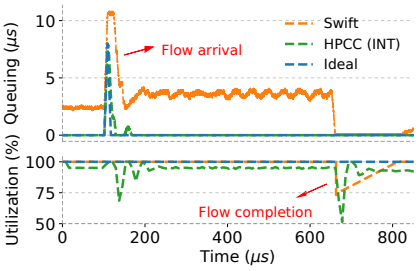


Figure 6: HPCC and Swift's reaction to flow arrival and completion.

where a new flow joins the network at $t = 100\mu$s while another flow is at its steady state. HPCC drains the initial queue built up in a couple of RTTs, but immediately oscillates between under-utilization and queuing for a few iterations. Moreover, the completion of a flow at $t = 650\mu$s again causes oscillations. Under highly dynamic traffic, such oscillations may increase tail latency and reduce network utilization.

An alternative way to avoid oscillations would be to react conservatively similar to Swift. It also reduces `cwnd` only once in an RTT during congestion but uses manually tuned parameters (i.e. *ai* and β) to make sure reactions are not impulsive. Although oscillations are prevented this way, Figure 6 shows that Swift takes a relatively long time to stabilize.

We conclude that once per RTT decisions can lead to either non-robust oscillations or relatively slow convergence. This is especially problematic in high-speed networks where flow arrivals and completions are extremely frequent. Ideally, the shortest observation period would be a packet's serialization time because it is the most granular decision unit for packet-switched networks. Yet, the per-packet CC decisions should only be incremental to deal with the noise from observations over such a short time interval.

3 Design

Bolt is designed for ultra-low-latency even at very high line rates by striving to achieve the ideal behavior shown in Figures 4 and 6. The design aims to reduce the control loop delay to its absolute minimum as described in §2. *First*, the congestion notification delay is minimized by generating notifications at the switches and reflecting them directly to the senders (§3.1). *Second*, the flow completion events are signaled by the senders in advance to hide the latency of ramp-up and avoid under-utilization (§3.2). *Third*, `cwnd` is updated after each feedback for quick stabilization where the update is at most one per packet to be resilient to noise. Together, these three ideas allow for a precise CC that operates on a per-packet basis minimizing incorrect CC decisions.

Prior works have separately proposed sub-RTT feedback [17, 50, 57], flow completion signaling [18], and per-packet cwnd adjustments [16, 27] which are discussed in §7. Bolt's main innovation is weaving these pieces into a harmonious and precise sub-RTT congestion control that is feasible for modern high-performance data centers. The key is to address congestion based on the *packet conservation principle* [25] visualized in Figure 7 where a network path is modeled as a pipe with a certain capacity of packets in-flight at a time. When the total `cwnd` is larger than the capacity by 1, there is an excess packet in the pipe which is queued. If the total `cwnd` is smaller than the capacity by 1, the bottleneck link will be under-utilized by 1 packet per RTT. Therefore, as soon as a packet worth queuing or under-utilization is observed, one of the senders should *immediately* decrement or increment the `cwnd`, without a long observation period.

Bolt's fundamental way of minimizing feedback delay and the observation period while generating precise feedback for per-packet decisions is materialized with 3 main mechanisms:

1. **SRC (Sub-RTT Control)** reduces congestion notification delay to its absolute minimum. (§3.1)
2. **PRU (Proactive Ramp Up)** hides any feedback delay for foreseen under-utilization events. (§3.2)
3. **SM (Supply Matching)** quickly recovers from unavoidable under-utilization events. (§3.3)

To realize these 3 mechanisms, Bolt uses 9 bytes of transport-layer header detailed in listing 1. We explain the purpose of each field as we describe the design of Bolt whose switching logic is summarized in Algorithm 1.

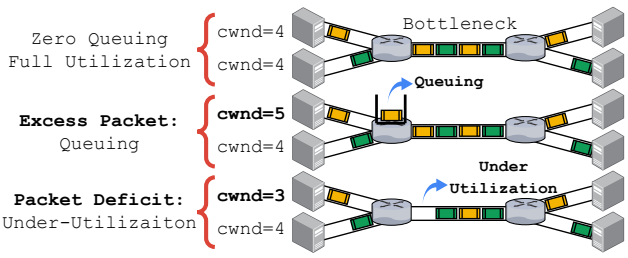


Figure 7: Pipe model of Packet Conservation Principle

Algorithm 1: BOLT LOGIC AT THE SWITCH

```
1  BoltIngress (pkt):
2      if !pkt.data then ForwardAndReturn(pkt)
3      CalculateSupplyToken(pkt)                    ▷ see Algorithm 3
4      if cur_q_size ≥ CC_THRESH then               ▷ Congested
5          if !pkt.dec then
6              pkt_src.queue_size ← switch.q_size
7              pkt_src.link_rate ← switch.link_rate
8              pkt_src.t_data_tx ← pkt.tx_time
9              SendSRC(pkt_src)
10         pkt.dec, pkt.inc ← 1,0
11     else if pkt.last then                        ▷ Near flow completion
12         if !pkt.first then pru_token++
13     else if pkt.inc then                         ▷ Pkt demands a token
14         if pru_token > 0 then
15             pru_token ← pru_token − 1
16         else if sm_token ≥ MTU then
17             sm_token ← sm_token − MTU
18         else
19             pkt.inc ← 0                          ▷ No token for cwnd inc.
20     ForwardAndReturn(pkt);
```

Listing 1: Bolt header structure

```
header bolt_h:
  bit<24> q_size;     // Occupancy at the switch
  bit<8>  link_rate;  // Rate of congested link
  bit<1>  data;       // Flags data packets
  bit<1>  ack;        // Flags acknowledgements
  bit<1>  src;        // Flags switch feedback
  bit<1>  last;       // Flags last wnd of flow
  bit<1>  first;      // Flags first wnd of flow
  bit<1>  inc;        // Signals cwnd increment
  bit<1>  dec;        // Signals cwnd decrement
  bit<1>  reserved;   // Reserved
  bit<32> t_data_tx; // TX timestamp for data pkt
```

3.1 SRC - Sub-RTT Control

As discussed in §2.1.1, a smaller feedback delay improves the performance of CC. Therefore, Bolt minimizes the delay of the feedback by generating control packets at the *ingress* pipeline of the switches and sending them directly *back to the sender*, a mechanism available in programmable switches such as Intel-Tofino2 [32]. While in spirit, this is similar to ICMP Source Quench messages [45] that have been deprecated due to feasibility issues in the Internet [33], Bolt's SRC mechanism exploits precise telemetry in a highly controlled data center environment.

Figure 8 depicts the difference in the paths traversed by the traditional ACK-based feedback versus the SRC-based feedback mechanism. As SRC packets are generated at ingress, they establish the absolute minimum feedback loop possible by traveling through the shortest path between a congested switch and the sender. Moreover, to further minimize the feedback delay, Bolt prioritizes ACK and SRC packets over data packets at the switches.

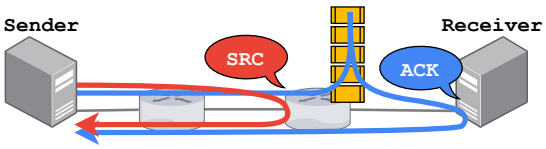


Figure 8: Path of ACK-based vs. SRC-based feedback

Bolt generates SRC packets for every data packet that arrives when the queue occupancy is greater than or equal to the CC_{THRESH} which is trivially set to a single MTU for minimal queuing. Yet, if there are multiple congested switches along the path of a flow, generating an SRC at each one of them for the same data would flood the network with an excessive amount of control packets. To prevent flooding switches mark the *DEC* flag of the original data packet upon generation of an SRC packet, such that no further SRC packets at other hops can be generated due to this packet (lines 5 and 10 in Algorithm 1). This implies that the number of SRC packets is bounded by the number of data packets in the network at any given time. In practice, however, we find that the actual load of SRC packets is extremely lower (§5.2.1) and present an approximation for the additional load of SRC packets in Appendix A.

When there are multiple congested hops, and the flow receives SRC packets only from the first one, the `cwnd` decrement still helps mitigate congestion at all of them. Consequently, even if congestion at the first hop is not as severe as the others, Bolt would drain the queue at the first hop and quickly start working towards the subsequent hops.

Bolt stamps two vital pieces of information on the SRC packets – the current queue occupancy and the capacity of the link. In addition, it reflects the TX timestamp of the original data packet (lines 6-8 in Algorithm 1). As the sender receives this packet, it runs the decision logic shown in Algorithm 2. First, rtt_{src} is calculated as the time between transmitting the corresponding data packet and receiving an SRC packet for it. This is the congestion notification delay for Bolt which is always shorter than RTT and enables sub-RTT control. The reflection of the TX timestamp enables this computation without any state at the sender. Next, $reaction_factor$ is calculated as a measure of this flow's contribution to congestion. Multiplying this value with the reported queue occupancy gives the amount of queuing this flow should aim to drain. All the flows aiming to drain only what they are responsible for organically help for a fair allocation.

Finally, $\frac{rtt_{src}}{target_q}$ gives the shortest time interval between two consecutive `cwnd` decrements. This interval prevents overreaction because switches keep sending congestion notifications until the effect of the sender's `cwnd` change propagates to them. For example, if the target queue has a single packet, the sender decrements its `cwnd` only if rtt_{src} has elapsed since the last decrement. However, if the queue is larger, Bolt allows

Algorithm 2: BOLT LOGIC AT THE SENDER HOST

```
1  HandleSrc (pkt_src):
2      rtt_src ← now − pkt.t_tx_data
3      reaction_factor ← flow.rate / pkt_src.link_rate
4      target_q ←                          ▷ in number of packets
5          pkt_src.queue_size × reaction_factor
6      if rtt_src / target_q ≤ now − last_dec_time then
7          cwnd ← cwnd − 1
8          last_dec_time ← now
9  HandleAck (pkt_ack):
10     if pkt_ack.inc then                 ▷ Capacity available
11         cwnd ← cwnd + 1
12     if pkt_ack.seq_no ≥ seq_no_at_last_ai then
13         cwnd ← cwnd + 1                 ▷ per-RTT add. inc.
14         seq_no_at_last_ai ← snd_next
```

more frequent decrements to equalize the total `cwnd` change to the target queue size in exactly one rtt_{src}. As the required `cwnd` adjustments are scattered over rtt_{src}, Bolt becomes more resilient to noise from any single congestion notification.

Events such as losses and timeouts do not happen in Bolt as it starts reacting to congestion way in advance. However, due to the possibility of such events occurring, say due to misconfiguration or packet corruption, handling retransmission timeouts, selective acknowledgments, and loss recovery are kept the same as in Swift [30] for completeness.

3.2 PRU - Proactive Ramp Up

Bolt explicitly tracks flow completions to facilitate Proactive Ramp Up (PRU). When a flow is nearing completion, it marks outgoing packets to notify switches, which plan ahead on distributing the bandwidth freed up by the flow to the remaining ones competing on the link. This helps remaining Bolt flows to *proactively ramp up* and eliminate the under-utilization period after a flow completion (see Figure 3b).

When flows larger than one BDP are sending their last `cwnd` worth of data, they set the *LAST* flag on packets to mark that they will not have packets in the next RTT. Note that this does not require knowing the application-level flow size. In a typical transport like TCP, the application injects a known amount of data to the connection at each `send` API call, denoted by the `len` argument [29]. Therefore, the amount of data waiting to be sent is calculable. *LAST* is marked only when the remaining amount of data in the connection is within `cwnd` size. Our detailed implementation is described in §4.2.

A switch receiving the *LAST* flag, if it is not congested, increments the *PRU token* value for the associated egress port. This value represents the amount of bandwidth that will be freed in the next RTT. The switch distributes these tokens to packets without the *LAST* flag, i.e. flows that have packets to send in the next RTT, so that senders can ramp up proactively.

However, only flows that are not bottlenecked at other hops should ramp up. To identify such flows, Bolt uses a greedy approach. When transmitting a packet, senders mark the *INC* flag on the packet. If a switch has PRU tokens (line 14 in Algorithm 1) or has free bandwidth (line 16 in Algorithm 1, explained in §3.3), it keeps the flag on the packet and consumes a token (line 15 and 17, respectively). Else, the switch resets the *INC* flag (line 19), preventing future switches on the path to consume a token for this packet. Then, if no switch resets the *INC* flag along the path, it is guaranteed that all the links on the flow's path have enough bandwidth to accommodate an extra packet. The receiver reflects this flag in the ACK so that the sender simply increments the `cwnd` upon receiving it (lines 10-11 in Algorithm 2). There are cases where the greedy approach can result in wasted tokens and we discuss the fallback mechanisms in §3.3.

Flows shorter than one BDP are not accounted for in PRU calculations. When a new flow starts, its first `cwnd` worth of packets are not expected by the network and contribute to the extra load. Therefore, the switch shouldn't replace these with packets from other flows once they leave the network. Bolt prevents this by setting the *FIRST* flag on packets that are in the first `cwnd` of the flow. Switches check against the *FIRST* flag on packets before they increment the *PRU token* value (line 12 of Algorithm 1).

Note that PRU doesn't need reduced feedback delay via SRC packets, because it accounts for a flow completion in the *next* RTT by design. A sender shouldn't start ramping up earlier as it can cause extra congestion before the flow completes. Therefore, the traditional RTT-based feedback loop is the right choice for correct PRU accounting.

3.3 SM - Supply Matching

Events like link and device failures or route changes can result in under-utilized links without proactive signaling. In addition, *PRU tokens* may be wasted if assigned to a flow that can not ramp up due to being already at line rate, or bottlenecked by downstream switches. For such events, conventional CC approaches rely on gradual *additive* increase to slowly probe for the available bandwidth which can take several tens of RTTs [1, 30, 42, 60]. Instead, Bolt is able to probe *multiplicatively* by explicitly matching utilization demand to supply through *Supply Matching* (SM) described below.

Bolt leverages stateful operations in programmable switches to measure the instantaneous utilization of a link. Each switch keeps track of the mismatch between the supply and demand for the link capacity for each port, where the number of bytes the switch can serialize in unit time is the supply amount for the link; and the number of bytes that arrive in the same time interval is the demand for the link. Naturally, the link is under-utilized when the supply is larger than the demand, otherwise, the link is congested. Note the similarity to HPCC [35] that also calculates link utilization, albeit from

Algorithm 3: Supply Token calculation at the ingress pipeline for each egress port of the switch

```
1  CalculateSupplyToken (pkt) :
2      inter_arrival_time ← now − last_sm_time
3      last_sm_time ← now
4      supply ← BW × inter_arrival_time
5      demand ← pkt.size
6      sm_token ← sm_token + supply − demand
7      sm_token ← min(sm_token, MTU)
```

an end-to-end point of view which restricts it to make once per RTT calculations. Bolt offloads this calculation to the switch data plane so that it can capture the precise instantaneous utilization instead of a coarse-grained measurement.

When a data packet arrives, the switch runs the logic in Algorithm 3 to calculate the *supply token* value (sm_token in the algorithms) associated with the egress port. The token accumulates the mismatch between the supply and demand in bytes on every packet arrival for a port. A negative value of the token indicates queuing whereas a positive value means under-utilization. When the token value exceeds one MTU, Bolt keeps the *INC* flag on the packet and permits the sender to inject an additional packet into the network (lines 16-17 in Algorithm 1). The *supply token* value is then decremented by an MTU to account for the inflicted future demand.

If a switch port doesn't receive a packet for a long time, the *supply token* value can get arbitrarily large, which prohibits capturing the instantaneous utilization if a burst of packets arrive after an idle period. To account for this, Bolt caps the *supply token* value at a maximum of one MTU. Details on how this feature is implemented in P4 are provided in §4.

As noted earlier, there are cases where there can be wasted tokens, i.e. a switch consumes a token (either PRU or SM) to keep *INC* bit, but is reset by downstream switches. In such cases, SM will find the available bandwidth in the next RTT. In the worst case, this happens for consecutive RTTs and Bolt falls back to additive increase similar to Swift [30] (lines 12-14 in Algorithm 2). Namely, `cwnd` is incremented once every RTT to allow flows to probe for more bandwidth and achieve fairness even if they do not receive any precise feedback as a fail-safe mechanism.

4 Implementation

We implemented Bolt through Host (transport layer and NIC) and Switch modifications in our lab. We used Snap [38] as our user-space transport layer and added Bolt in 1340 LOC in addition to the existing Swift implementation. Plus, the switch-side implementation consists of a P4 program – `bolt.p4` – in 1120 LOC. Figure 9 shows the overview of our lab prototype as a whole and we provide details below.

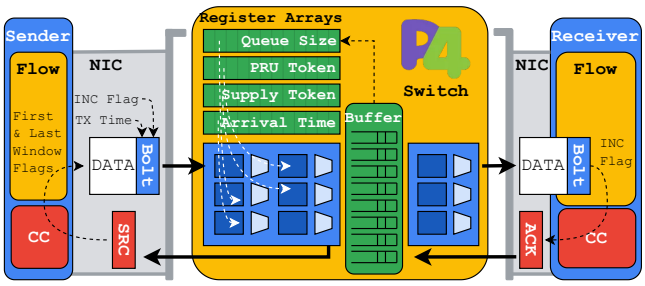


Figure 9: Bolt system overview

4.1 Switch Prototype

We based our implementation on the programmable data plane of Intel Tofino2 [11] switches in our lab as they can provide the queue occupancy of the egress ports in the ingress pipelines and generate SRC packets [32]. This is crucial for Bolt to minimize the feedback delay incurred by SRC packets as they are not subject to queuing delay at congested hops.

When congestion is detected in the ingress pipeline, the switch mirrors this packet to the input port while forwarding the original one along its path. The mirroring configuration is determined with a lookup table that matches the ingress port of the packet and selects the associated mirroring session.

The mirrored packet is then trimmed to remove the payload and the flow identifiers (i.e. source/destination addresses and ports) are swapped. Finally, *SRC* flag is set on this packet to complete its conversion into an SRC packet.

The entire `bolt.p4` consists mainly of register array declarations and simple if-else logic as shown in Algorithm 1. There are 4 register arrays for storing queue occupancy, token values, and the last packet arrival time. All of the register arrays are as large as the number of queues on the switch because the state is maintained per queue. In total, only 3.6% and 0.6% of available SRAM and TCAM, respectively, are used for the register arrays, tables, and counters.

The switch keeps the last packet arrival time for every egress port to calculate the supply for the link. On each data packet arrival, the difference between the current timestamp and the last packet arrival time is calculated as the inter-arrival time. This value should ideally be multiplied with the link capacity (line 4 of Algorithm 3) to find the supply amount. However, since floating point arithmetic is not available in PISA pipelines, we use a lookup table indexed on inter-arrival times to determine the supply amount. We set the size of this lookup table as 65536 where each entry is for a different inter-arrival time with a granularity of a nanosecond. Consequently, if the inter-arrival time is larger than 65 microseconds, the *supply token* value is directly set to its maximum value of 1 MTU which triggers *INC* flag to be set. We find that, at a reasonably high load, 65 microseconds of inter-arrival time is rare enough for links greater than 100Gbps such that any longer value can be safely interpreted as under-utilization.

Our prototype is based on a single HW pipeline. Therefore,

we implemented Bolt entirely at the ingress pipeline to make it easier to understand and debug its logic. However, since PRU and SM maintain state per egress port, they could also be implemented at the egress pipeline with minor modifications. This way, the state for packets from multiple ingress pipelines would naturally be aggregated.

4.2 Host Prototype

Our transport layer uses the NIC hardware timestamps to calculate rtt_{src} as described in Algorithm 2. When a sender is emitting data, the TX timestamp is stamped onto the packet. The switch reflects this value back to the sender, so that rtt_{src} is the difference between the NIC time when the SRC packet is received (RX timestamp) and the reflected TX timestamp. This precisely measures the network delay to the bottleneck without any non-deterministic software processing delays.

The transport layer also multiplexes RPCs meant for the same server onto the same network connection. Then, the first `cwnd` bytes of a new RPC isn't necessarily detected as the *first* window of the connection. To mitigate this issue, our prototype keeps track of idle periods of connections and resets the *bytes-sent counter* when a new RPC is sent after such a period. Therefore the *FIRST* flag is set on a packet when the counter value is smaller than `cwnd`.

Finally, the *last* window marking for PRU requires determining the size of the remaining data for each connection. In our prototype, the connection increments *pending bytes counter* by the size of data in each `send` API call from the application. Every time the connection transmits a packet into the network, the counter value is decremented by the size of the packet. Therefore the *LAST* flag is set on a packet when this counter value is smaller than `cwnd`.

4.3 Security and Authentication

Getting Bolt to work for encrypted and authenticated connections was a key challenge in our lab. Our prototype uses a custom version of IPsec ESP [23, 28] for encryption atop the IP Layer. However, switches need to read and modify CC information at the transport header without breaking end-to-end security. The *crypt_offset* of the protocol allows packets to be encrypted only beyond this offset. We set it such that the transport header is not encrypted, but is still authenticated.

In addition, switches cannot generate encrypted packets due to the lack of encryption and decryption capabilities. To remedy this, we generate SRC packets on switches as unreliable datagrams per RoCEv2 standard by adding IB BTH and DETH headers while removing the encryption header.

The RoCEv2 packets have the invariant CRC calculated over the packet and appended as a trailer. Fortunately, Tofino2 provides a CRC extern that is capable of this calculation over small, constant-size packets [31]. As a result, NICs are able to forward the SRC packets correctly to the upper layers based on the queue pair numbers (QPN) on the datagrams.

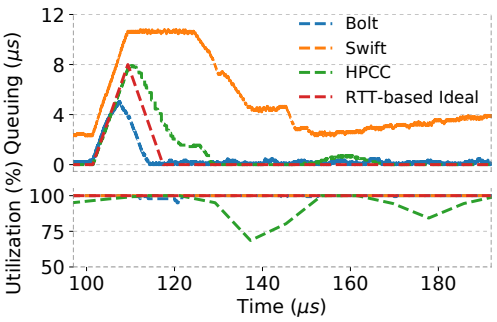


Figure 10: Bolt's reaction to flow arrival versus the ideal behavior.

5 Evaluation

We evaluate Bolt on NS3 [48] micro-benchmarks to demonstrate its fundamental capabilities in §5.1 followed by sensitivity and fairness analysis in §5.2 and §5.3. Then, in §5.4, we run large-scale experiments to measure the end-to-end performance of the algorithm, i.e. flow completion time slowdowns. Finally, we evaluate our lab prototype in §5.5.

5.1 Micro-Benchmarks

5.1.1 Significance of SRC

The only way for Bolt to decrease `cwnd` is through SRC whose effectiveness is best observed during congestion. Therefore, we repeat the same flow arrival scenario described in Figure 6 with Bolt.[3] Typically, with conventional RTT-based congestion control algorithms, a new flow starting at line rate emits BDP worth of packets until it receives the first congestion feedback after an RTT. If the network is already fully utilized before this flow, all emitted packets end up creating a BDP worth of queuing even for an RTT-based ideal scheme. Then, the ideal scheme would stop sending any new packets to allow draining the queue quickly which would take another RTT. This behavior is depicted as red in Figure 10 where a new flow joins at 100μs.

HPCC's behavior in Figure 10 is close to the ideal given that it is an RTT-based scheme with high precision congestion signal. As the new flow arrives, the queue occupancy rises to 1 BDP. However, the queue is drained at a rate slower than the link capacity because flows continue to occasionally send new packets while the queue is not completely drained.

Bolt, on the other hand, detects congestion earlier than an RTT. Therefore it starts decrementing `cwnd` before the queue occupancy reaches BDP and completely drains it in less than 2 RTTs, even shorter than the RTT-based ideal scheme.

In addition, HPCC's link utilization drops to as low as 75% after draining the queue and oscillates for some time, which is due to the RTT-long observation period (§2.2). Bolt's per-packet decision avoids this under-utilization.

[3] The dumbbell topology with two flows (8 μs RTT at 100Gbps).

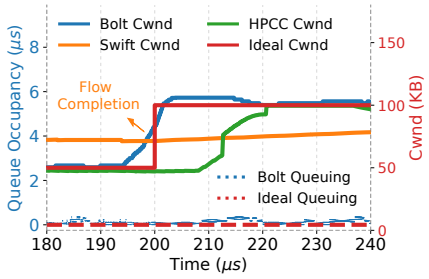
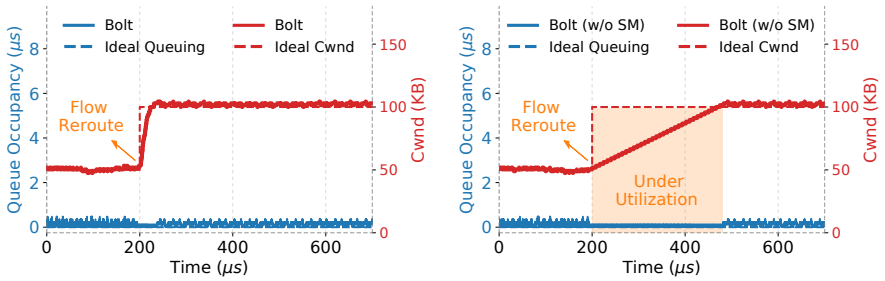


Figure 11: `cwnd` of the remaining flow and queue occupancy after a flow completion.



(a) Bolt **(b)** Bolt (without SM)

Figure 12: `cwnd` of the remaining flow and queue occupancy after a flow is rerouted.

Utilization (%)		PRU OFF	PRU ON
SM	**OFF**	90.46	97.38
	ON	92.41	98.54

Table 1: Effectiveness of PRU and SM on the bottleneck utilization.

5.1.2 Significance of PRU

Flow completions cause under-utilization without proactive ramp-up or standing queues because conventional congestion control algorithms take at least an RTT to react to them (§2.1.2). Moreover, as shown in Figure 4 for Swift, a standing queue might not be enough to keep the link busy if the `cwnd` of the completing flow is larger than the queue size

We repeat the same scenario with Bolt to test how effective proactive ramp-up can be upon flow completions against Swift and HPCC. Figure 11 shows the `cwnd` of the remaining flow and the queue occupancy at the bottleneck link. When a Bolt flow completes at t=200μs, the remaining one is able to capture the available bandwidth in 1μs because it starts increasing `cwnd` (by collecting PRU tokens) one RTT earlier than the flow completion. Moreover, neither queuing nor under-utilization is observed. HPCC, on the other hand, takes 20μs ($>$ 2$\times$RTT) to ramp up for full utilization because it needs one RTT to detect under-utilization and another RTT of observation period before ramping up. Finally, Swift takes more than 370μs to reach the stable value due to the slow additive increase approach which doesn't fit into Figure 11. The complete ramp-up of Swift is shown in Figure 4.

Although PRU and SM seem to overlap in the way they quickly capture available bandwidth, PRU is a faster mechanism compared to SM because it detects under-utilization proactively. To demonstrate that, we create a star topology with 100Gbps links and a base RTT of 5μs, where 5 senders send 500KB to the same receiver. Flows start 15μs apart from each other to complete at different times so that PRU and SM can kick in. We repeat while disabling PRU or SM and measure the bottleneck utilization to observe how each mechanism is effective at achieving high throughput.

Table 1 shows the link utilization between the first flow completion and the last one. When only PRU is disabled, the utilization drops by 6% despite having SM. On the other hand, disabling SM alone causes only a 1% decrease. This indicates that PRU is a more powerful mechanism compared to SM when under-utilization is mainly due to flow completions in the network. Together, they increase utilization by 8%.

5.1.3 Significance of SM

Unlike flow completions, events such as link failure or rerouting are not hinted in advance. Then, PRU doesn't kick in, making Bolt completely reliant on SM for high utilization. To demonstrate how SM quickly captures available bandwidth, we use the same setup from Figures 4 and 11, but reroute the second flow instead of letting it complete.

Figure 12 shows the `cwnd` of the remaining flow after the other one leaves the bottleneck. Thanks to SM, `cwnd` quickly ramps up to utilize the link in 23μs (12a). When SM is disabled, the only way for Bolt to ramp up is through traditional additive increase which increases `cwnd` by 1 every RTT (12b). Therefore it takes more than 33 RTTs to fully utilize the link.

5.2 Sensitivity Analysis

5.2.1 Overhead of SRC

To mitigate congestion, Bolt generates SRC packets in an already loaded network. In order to understand the extra load created by SRC, we measure the bandwidth occupied by SRC packets at different burstiness levels. For this purpose, we use the same star topology from §5.1.2. The number of senders changes between 1 and 63 to emulate different levels of burstiness towards a single receiver at 80% load. The traffic is based on the READ RPC workload from Figure 1.

Figure 13 shows the bandwidth occupied by the SRC packets (top) and the 99th-p queue occupancy at the bottleneck (bottom) with a different number of senders. When there are multiple senders, the SRC bandwidth is stable at 0.33Gbps (0.33% of the capacity). Similarly, the tail queuing is also bounded below 6.4μs for all the experiments. Therefore, we conclude that Bolt is able to bound congestion with a negligible amount of extra load in the network. In §5.5, we show that the overhead is negligible for the lab prototype as well.

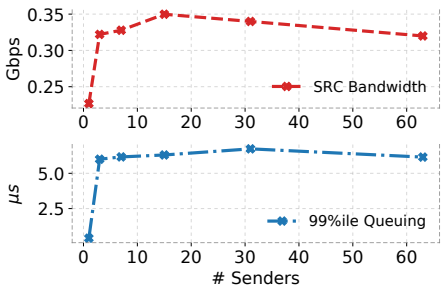


Figure 13: SRC overhead and sensitivity for different levels of burstiness

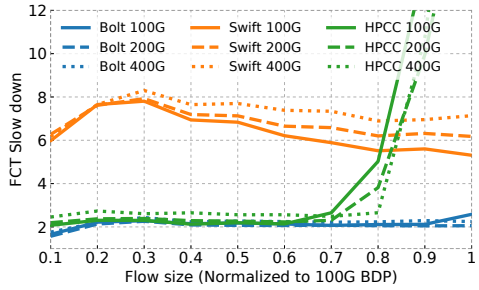


Figure 14: 99th-p Slowdown for messages smaller than BDP

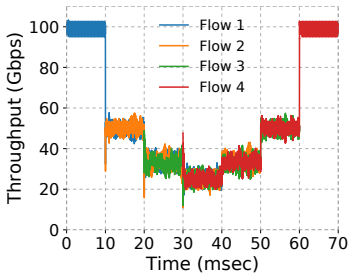


Figure 15: Fair allocation by Bolt

Metric	Swift	HPCC	Bolt
99th-p Queuing (msec)	23.543	23.066	13.720
99th-p FCT Slowdown	7017	5037	5000

Table 2: Tail queuing, and FCT slowdown for 5000-to-1 incast.

5.2.2 Robustness Against Higher Line Rates

One of the goals of Bolt is to be robust against ever-increasing line rates in data centers. To evaluate the performance at different line rates, we repeat the simulations from §5.2.1 with 63 senders where we increase the link capacity from 100Gbps to 200Gbps and 400Gbps. This way, the burstiness of the senders increases, making it difficult to maintain small queuing at the switches. Therefore, flow completion time (FCT) slowdown [14][4] of small flows are affected the most, whereas throughput oriented large flows would trivially be better off with higher line rates.

Accordingly, we plot the 99th-p FCT slowdown for flows that are smaller than BDP (at 100 Gbps) in Figure 14. Swift's performance monotonically decays with higher link rates due to the increasing burstiness. Similarly, HPCC at 400Gbps achieves 25% worse performance compared to the 100Gbps scenario for flow sizes up to 0.7 BDP. For the rest of the workload, HPCC makes a leap such that it performs worse than other algorithms irrespective of the line rates. Bolt on the other hand is able to maintain small and steady tail slowdowns for all the small flows despite the increasing line rates.

5.3 Fairness Analysis

To test the fairness of Bolt, we run an experiment on a dumbbell topology with 100Gbps links. We add or remove a new flow every 10 milliseconds and measure the throughput of each flow which is shown in Figure 15. Our results indicate that Bolt flows converge to the new fair share quickly when the state of the network changes.

[4]FCT slowdown is flow's actual FCT normalized by its ideal FCT when the flow sends at line-rate (e.g., when it was the only flow in the network).

5.4 Large Scale Simulations

One of the most challenging cases for CC is a large-scale incast. To evaluate Bolt's performance in such a scenario, we set up a 5000-to-1 incast on the star topology described earlier where each one of 50 senders starts 100 same size flows at the same time. Table 2 presents the 99th-p queue occupancy and FCT slowdown for the incast. Since Bolt detects congestion as early as possible, it bounds tail queuing to a 41% lower level compared to Swift and HPCC. In addition, the tail FCT slowdown for Bolt is 5000, indicating full link utilization. Moreover, the bandwidth occupied by the SRC packets is as low as 0.77Gbps throughout the incast. This is only twice the overhead for 80% load in §5.2.1, despite the extreme bursty arrival pattern of the incast.

We also evaluate the performance of Bolt on a cluster-scale network where 64 servers are connected with 100Gbps links to a fully subscribed fat-tree topology with 8 ToR switches. All the other links are 400Gbps and the maximum unloaded RTT is 5μs. We run traffic between servers based on two workloads at 80% load: (i) the READ RPC workload described in Figure 1 represents traffic from our data center, (ii) the Facebook Hadoop workload [49]. Figure 16 and 17 show the median and 99th-p FCT slowdown for the workloads. Note that the Hadoop workload is relatively more bursty where 82% of the flows/RPCs fit within a BDP in the given topology. Hence a large fraction of the curves in Figure 17 is flat where all the RPCs in this region are extremely small (i.e. single packet).

For both of the workloads, Bolt performs well across all flow sizes. Specifically, Bolt and HPCC achieve very low FCT for short flows (<7KB) because of a few design choices: First, they maintain zero standing queues. Plus, Bolt's SRC reduces the height of queue spikes after flow arrivals. HPCC, on the other hand, tends to under-utilize the network upon flow completions (§2.1.2), statistically reducing queue sizes.

FCT of median-size flows (a few BDPs) starts to degrade for HPCC due to under-utilization described in §2.1.2 and §2.2. Bolt performs up to 3× better in this regime by avoiding under-utilization thanks to PRU and SM. Swift's standing queues prevent under-utilization, but FCTs are high because

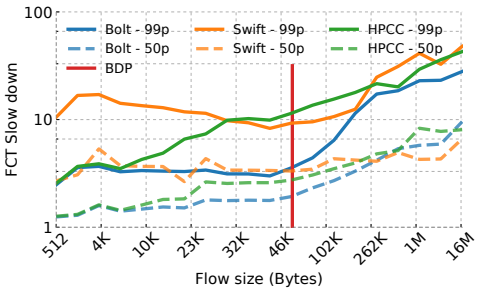


Figure 16: FCT slowdown for READ RPC Workload from Figure 1

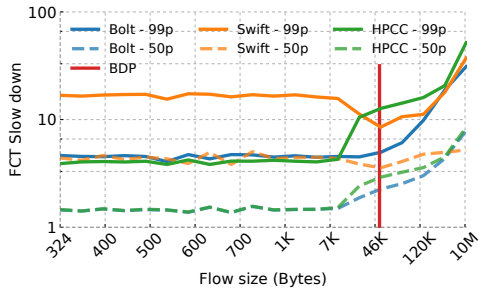


Figure 17: FCT slowdown for Facebook Hadoop Workload

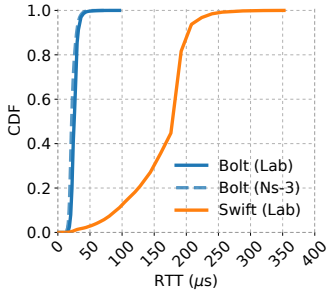


Figure 18: Bolt's lab prototype matches its simulator

median-size flows are also affected by the queuing delay.

The impact of queuing diminishes and utilization becomes the dominant factor for long flows. Therefore Bolt and Swift perform better than HPCC. In addition, Bolt is slightly better at the tail compared to Swift, while Swift is slightly better at the median, suggesting that Bolt is fairer.

5.5 Bolt in the Lab

Our lab testbed consists of 2 servers and 2 Intel Tofino2 [11] switches. Each server runs 4 packet processing engines running Snap [38] that provide the transport layer with the Bolt algorithm. Each engine is scheduled on a CPU core that independently processes packets so that we are able to create a large number of connections between the servers. Links from the servers to the switches are 100Gbps and the switches are connected to each other with a 25Gbps link to guarantee that congestion takes place within the network. The base-RTT in this network is 14μs and we generate flows between the servers based on the READ RPC workload.

We evaluate Bolt on two scenarios. First, we run 100% (of 25Gbps) load to see if our prototype can saturate the bottleneck. Then, we run 80% load to compare the congestion mitigation performance of Bolt against Swift in a more realistic scenario. Finally, we verify that our results from the lab and the simulations match to verify our implementations.

The median and the 99th-p RTT at 100% load for Swift are 189μs and 208μs respectively. These numbers are high because Swift maintains a standing queue based on the configured base delay to fully utilize the link even after flow completions. Bolt on the other hand, attains 27μs and 40μs of median and tail RTT, 86% and 81% shorter than Swift. In the meantime, it achieves 24.7Gbps which is only 0.8% lower compared to Swift despite the lack of a standing queue.

We repeat the same experiment with 80% load and observe that both Swift and Bolt can sustain 80% (20Gbps) average link utilization. Figure 18 shows the CDF of measured RTTs throughout the experiment. Similar to the 100% load case, the median and tail RTTs for Bolt are 25μs and 40μs, 86% and 83% lower compared to Swift respectively[5].

Moreover, we measure that the bandwidth occupied by the SRC packets in our lab is 0.13Gbps, 0.536% of the bottleneck capacity. This is consistent with our observation in §5.2 despite the larger SRC packets with custom encapsulations.

Finally, we simulate the 80% load experiment in NS3 [48] with the same settings to verify that our simulator matches our observations in the lab. Figure 18 also shows the CDF of RTTs measured throughout the simulation. The median and tail RTTs from our simulations are 21μs and 39μs, within 15% and 0.025% of the lab results respectively.

6 Practical Considerations

Typically, new products are deployed incrementally in data centers due to availability, security, or financial concerns. As a consequence, the new product (i.e. the CC algorithm) lives together with the old one for some time called brownfield deployment. We identify three potential issues that Bolt could face during this phase and address them below.

First, some switches in the network may not be capable of generating SRC packets while new programmable switches are being deployed. Unfortunately, the vanilla Bolt design can not control the congestion at these switches. This can be addressed by running an end-to-end algorithm on top of Bolt. For example, imagine the Swift algorithm calculates a fabric `cwnd` as usual in parallel with Bolt's calculation of `cwnd` using SRC packets. Then, the minimum of the two is selected as the effective `cwnd` for the flow. When an older generation switch is congested, SRC packets are not generated, but Swift adjusts the `cwnd`. Consequently, flows benefit from ultra-low queuing at the compatible switches while falling back to Swift when a non-programmable switch becomes the bottleneck.

Second, hosts would also be migrated to Bolt incrementally. Therefore, Bolt would need to coexist with the prior algorithm. Studying the friendliness of algorithms with Bolt through frameworks such as [26] and [56] remains a future work. For example, TCP CUBIC would not coexist well with Bolt as it tries to fill the queues until a packet is dropped while Bolt would continuously decrement its `cwnd` due to

[5]For Swift we set 50μs base target delay as specified in the paper [30] and 200μs as flow scaling range. Swift's average RTT in Figure 18 is higher than Swift paper's value (~50μs), because of two reasons. First, this workload is burstier than the ones in Swift paper. Second, the 25Gbps bottleneck implies a higher level of flow scaling than with 100Gbps links.

queuing. Instead, we propose the use of QoS (Quality of Service) queues to isolate Bolt traffic from the rest. Appendix B describes a baseline approach for such deployment.

Finally, scenarios where packet transmissions are batched (say by the NIC) even when the `cwnd` is smaller than BDP can still trigger SRC generation, inhibiting flows to increase `cwnd` to the right value. We find that transport offloading on modern smart NICs uses batching to sustain high line rates. Bolt alleviates such bursts with a higher CC_{THRESH} that tolerates batch size worth of queuing at the switches.

7 Related Work

In addition to HPCC [35] and Swift [30] that serve as our primary comparison points, several other schemes have similar ideas or goals.

FastTune [59] uses programmable switches for precise congestion signal. Similar to HPCC, it calculates link utilization over an RTT to multiplicatively increase or decrease `cwnd`. For shorter feedback delay, it pads the INT header onto ACK packets in the reverse direction instead of data packets. ExpressPass [10] utilizes the control packets in the reverse direction as well. Nonetheless, forward and reverse paths for a flow are not always symmetrical due to ECMP-like load balancing or flow-reroutes. Therefore, Bolt chooses to explicitly generate SRC packets with little overhead (§5.2).

FastLane [57] is one of the early proposals to send notifications from switches directly to the senders. However, notifications are generated only for buffer overflows which is late for low latency CC in data centers. Annulus [50], on the other hand, uses standard QCN [21] packets from switches with queue occupancy information. Yet these packets are not L3 routable, so Annulus limits its scope only to detecting bottlenecks one hop away from senders. Bolt brings the best of both worlds and controls congestion at every hop while knowing the precise state of congestion.

XCP [27] and RCP [13] also propose congestion feedback generated by the switch. Switches wait for an average RTT before calculating CC responses that are piggybacked on the data packet and reflected on the ACK. As discussed in §2.2, this implies a control loop delay of two RTTs in total.

FCP [18] uses budgets and prices to balance the load and the traffic demand. FCP switches calculate the price of the link based on the demand while senders signal flow arrivals or completions similar to SM and PRU in Bolt. However, the required time series averaging and floating-point arithmetic make the calculation infeasible for programmable switches while consuming bytes on the header. In contrast, Bolt is based on the packet conservation principle with a simple, yet precise logic implementable in P4 and requires only 3 bits on the header (*FIRST*, *LAST*, and *INC*) for SM and PRU.

Switch feedback has also been studied for wireless settings. For instance, ABC [16] marks packets for `cwnd` increments or decrements with an RTT-based control loop for congestion mitigation. On the other hand, Zhuge [39] modifies the wireless AP to help senders detect congestion quicker. However, since it is challenging to modify schemes in WAN, Zhuge relies on the capabilities of existing schemes for the precision of the congestion signals, i.e. delayed ACKs for TCP.

Receiver-driven approaches such as NDP [19], pHost [15], and Homa [43] require receivers to allocate/schedule credits based on the demand from senders. They work well for congestion at the last hop because receivers have good visibility into this link. For example, when an RPC is fully granted, the Homa receiver starts sending grants for the next one without the current RPC being finished to proactively utilize the link. This is similar to PRU in Bolt despite being limited to the last hop. Unfortunately, the last hop is not always the bottleneck for a flow especially when the fabric is over-subscribed [52].

Schemes that use priority queues [2, 4, 20, 43] are proposed to improve the scheduling performance of the network to approximate SRPT [51] like behavior. We find deploying such schemes to be rather difficult because, typically, QoS queues in data centers are reserved to separate different services.

On-Ramp [37] is an extension for CC which proposes to pause flows at the senders when the one-way delay is high. Bolt can also benefit from its flow control mechanism. We leave evaluating Bolt with this extension as future work.

There are also per-hop flow control mechanisms such as BFC [17] and PFFC [55] that pause queues at the upstream switches via early notifications from the bottleneck. The deadlock-like issues of PFC [54] are resolved by keeping the per-flow state on switches, which we find challenging in our data centers as switches have to implement other memory or queue-intensive protocols, e.g., routing tables or QoS. Therefore, we scope Bolt to be an end-to-end algorithm with a fixed state similar to other algorithms in production [30, 35, 60].

8 Conclusion

Increasing line rates in data centers is inevitable due to the stringent SLOs of applications. Yet, higher line rates increase burstiness, putting more pressure on CC to minimize queuing delays for short flows along with high link utilization for long flows. We find that two key aspects of CC need to be pushed to their boundaries to work well in such highly dynamic regimes based on experience with our data centers.

Bolt addresses these aspects thanks to the flexibility and precision provided by programmable switches. First, it uses the most granular congestion signal, i.e. precise queue occupancy, for a per-packet decision logic. Second, it minimizes the control loop delay to its absolute minimum by generating feedback at the congested switches and sending them directly back to the senders. Third, it hides the control loop delay by making proactive decisions about foreseeable flow completions. As a result, accurate `cwnd` is calculated as quickly as possible, achieving more than 80% reduction in tail latency and 3× improvement in tail FCT.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.*, 40(4):63–74, August 2010. doi:10.1145/1851275.1851192.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. PFabric: Minimal near-Optimal Datacenter Transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, August 2013. doi:10.1145/2534169.2486031.

[3] Serhat Arslan and Nick McKeown. Switches Know the Exact Amount of Congestion. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3375235.3375245.

[4] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, 2015.

[5] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Commun. ACM*, 60(4):48–54, March 2017. doi:10.1145/3015146.

[6] Luiz Andrè Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, San Rafael, CA, USA, 3rd edition, October 2018. URL: https://doi-org.stanford.idm.oclc.org/10.2200/S00874ED3V01Y201809CAC046.

[7] Tanya Bhatia. UADP - The Powerhouse of Catalyst 9000 Family. Cisco Systems Inc., December 2018. URL: https://community.cisco.com/t5/networking-blogs/uadp-the-powerhouse-of-catalyst-9000-family/ba-p/3764605.

[8] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. RFC 5681, September 2009. URL: https://rfc-editor.org/rfc/rfc5681.txt, doi:10.17487/RFC5681.

[9] Mihai Budiu and Chris Dodd. The P4-16 Programming Language. *SIGOPS Oper. Syst. Rev.*, 51(1):5–14, September 2017. URL: https://doi-org.stanford.idm.oclc.org/10.1145/3139645.3139648, doi:10.1145/3139645.3139648.

[10] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 239–252, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3098822.3098840.

[11] Intel Corporation. Tofino 2: Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise, May 2021. URL: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[12] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013. URL: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[13] Nandita Dukkipati. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3292347. URL: https://dl-acm-org.stanford.idm.oclc.org/doi/10.5555/1368746.

[14] Nandita Dukkipati and Nick McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, January 2006. doi:10.1145/1111322.1111336.

[15] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2716281.2836086.

[16] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A Simple Explicit Congestion Controller for Wireless Networks . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/goyal.

[17] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, Renton, WA, April 2022. USENIX Associa-

tion. URL: https://www.usenix.org/conference/nsdi22/presentation/goyal.

[18] Dongsu Han, Robert Grandl, Aditya Akella, and Srinivasan Seshan. FCP: A Flexible Transport Framework for Accommodating Diversity. *SIGCOMM Comput. Commun. Rev.*, 43(4):135–146, August 2013. doi:10.1145/2534169.2486004.

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3098822.3098825.

[20] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 422–434, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405878.

[21] IEEE Standard for Local and Metropolitan Area Networks– Virtual Bridged Local Area Networks Amendment 13: Congestion Notification. *IEEE Std 802.1Qau-2010 (Amendment to IEEE Std 802.1Q-2005)*, pages 1–135, 2010. doi:10.1109/IEEESTD.2010.5454063.

[22] Broadcom Inc. High-Capacity StrataXGS Trident4 Ethernet Switch Series, May 2021. URL: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[23] Google Inc. PSP, March 2022. URL: https://github.com/google/psp.

[24] Versa Technology Inc. 400G Ethernet: It's Here, and It's Huge, December 2021. URL: www.versatek.com/400g-ethernet-its-here-and-its-huge/.

[25] Van Jacobson. Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 314–329, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/52324.52356.

[26] Raj Jain, Dah-Ming Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR*, cs.NI/9809099, January 1998. URL: https://arxiv.org/abs/cs/9809099.

[27] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 89–102, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/633025.633035.

[28] Stephen Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005. URL: https://www.rfc-editor.org/info/rfc4303, doi:10.17487/RFC4303.

[29] Michael Kerrisk. send(2) — Linux manual page, March 2021. URL: https://man7.org/linux/man-pages/man2/send.2.html.

[30] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3406591.

[31] Shiv Kumar, Pravein Govindan Kannan, Ran Ben Basat, Rachel Everman, Amedeo Sapio, Tom Barbette, and Joeri de Ruiter. Open Tofino, July 2021. URL: https://github.com/barefootnetworks/Open-Tofino.

[32] Jeongkeun Lee, Jeremias Blendin, Yanfang Le, Grzegorz Jereczek, Ashutosh Agrawal, and Rong Pan. Source Priority Flow Control (SPFC) towards Source Flow Control (SFC), November 2021. URL: https://datatracker.ietf.org/meeting/112/materials/slides-112-iccrg-source-priority-flow-control-in-data-centers-00.

[33] Konstantin Lepikhov. Source Quench. Atlassian Corporation Pty Ltd., April 2018. URL: https://wiki.geant.org/display/public/EK/Source+Quench.

[34] Yuliang Li. *Hardware-Software Codesign for High-Performance Cloud Networks*. PhD thesis, Harvard University, 2020. URL: https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37368976.

[35] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342085.

[36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control, December 2021 [Online]. URL: https://hpcc-group.github.io/results.html.

[37] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 47–63, Berkeley, CA, USA, April 2021. USENIX Association. URL: https://www.usenix.org/conference/nsdi21/presentation/liu.

[38] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359657.

[39] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving Consistent Low Latency for Wireless Real-Time Communications with the Shortest Control Loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 193–206, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544225.

[40] Rui Miao, Li Bo, Hongqiang Harry Liu, and Ming Zhang. Buffer sizing with HPCC. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, pages 1–2, New York, NY, USA, 2019. Association for Computing Machinery. URL: http://buffer-workshop.stanford.edu/papers/paper5.pdf.

[41] Leonid Mirkin and Zalman J. Palmor. Control Issues in Systems with Loop Delays. In Dimitrios Hristu-Varsakelis and William S. Levine, editors, *Handbook of Networked and Embedded Control Systems*, pages 627–648. Birkhäuser Boston, Boston, MA, 2005. doi:10.1007/0-8176-4404-0_27.

[42] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787510.

[43] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230543.3230564.

[44] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting TCP for Reconfigurable Datacenter Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/mukerjee.

[45] John Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984. URL: https://www.rfc-editor.org/info/rfc896, doi:10.17487/RFC0896.

[46] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993. doi:10.1109/90.234856.

[47] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, February 2018. URL: https://rfc-editor.org/rfc/rfc8312.txt, doi:10.17487/RFC8312.

[48] George F. Riley and Thomas R. Henderson. The ns-3 Network Simulator. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-12331-3_2.

[49] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 123–137, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787472.

[50] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 735–749, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405899.

[51] Linus E. Schrage and Louis W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research*, 14(4):670–684, 1966. doi:10.1287/opre.14.4.670.

[52] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787508.

[53] Bruce Spang, Serhat Arslan, and Nick McKeown. Updating the theory of buffer sizing. *Performance Evaluation*, 151:102232, 2021. doi:https://doi.org/10.1016/j.peva.2021.102232.

[54] IEEE Standard. Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011*, (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011):1–40, 2011. doi:10.1109/IEEESTD.2011.6032693.

[55] Shie-Yuan Wang, Yo-Ru Chen, Hsien-Chueh Hsieh, Ruei-Syun Lai, and Yi-Bing Lin. A Flow Control Scheme Based on Per Hop and Per Flow in Commodity Switches for Lossless Networks. *IEEE Access*, 9:156013–156029, 2021. doi:10.1109/ACCESS.2021.3129595.

[56] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 17–24, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3365609.3365855.

[57] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. FastLane: Making Short Flows Shorter with Agile Drop Notification. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 84–96, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2806777.2806852.

[58] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 78–85, New York, NY, USA, 2017. Association for Computing Machinery. URL: https://doi-org.stanford.idm.oclc.org/10.1145/3131365.3131375, doi:10.1145/3131365.3131375.

[59] Renjie Zhou, Dezun Dong, Shan Huang, and Yang Bai. FastTune: Timely and Precise Congestion Control in Data Center Network. In *2021 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 238–245, New York City, NY, USA, 2021. IEEE. doi:10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00043.

[60] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787484.

Appendix

A Approximating SRC Overhead

Bolt switches generate SRC packets for every data packet they receive as long as there is queuing, given that the data packet is not marked with the *DEC* flag. Then the number of

SRC packets depend on how long queuing persisted and how many packet are received in this time interval.

At steady state where no new RPCs join the network, we can estimate the fraction of time queuing persist on a bottleneck. Congestion at this regime happens only due to the once-per-RTT additive increase of 1 by each flow.

As described in §3.1, senders pace `cwnd` decrements such that the total number of decrements equals the queue occupancy after 1 rtt_{src}. This implies that any queuing will persist for 1 rtt_{src}, but will be completely drained after. Since a new congestion is not inflicted until the next RTT, we conjecture that the fraction of time that the switch has non-zero queuing is governed by the following golden ratio:

$$\text{fraction of time switch is congested} = \frac{rtt_{src}}{rtt} \tag{1}$$

which is always less then 1.

Note that equation 1 is an approximation for congestion interval since it doesn't incorporate traffic load, new RPC arrivals or multi bottleneck scenarios. Nonetheless, we can calculate the number of SRC packets generated at a bottleneck with it.

$$\text{\# of SRC pkts} = \text{\# of DATA pkts} \times \frac{rtt_{src}}{rtt} \tag{2}$$

Finally, we map equation 2 to the bandwidth occupied by the SRC packets by incorporating the link capacity and the packet sizes:

$$\text{SRC Bandwidth} = C \times \frac{p_{src}}{p_{data}} \times \frac{rtt_{src}}{rtt} \tag{3}$$

Where C is the rate at which the traffic is flowing through the bottleneck link, p_{src} is the size of SRC packets and p_{data} is the size of data packets, i.e. MTU.

When we calculate the bandwidth of SRC packets according to equation 3 for the simulation in §5.2.1, we find 0.37Gbps which is within 12% of the simulation result of 0.33Gbps. Moreover, equation 3 gives 0.10Gbps for our lab setup in §5.5 which is within 23% of the measured value of 0.13Gbps.

B Bolt with QoS

The relationship between congestion control algorithms and QoS has always been contradictory. An ideal congestion control algorithm aims to mitigate any queuing at the switch, whereas a QoS mechanism always needs enough queuing to be able to differentiate packet priorities and serve one before the other. Put another way, QoS only takes effect when the arrival rate at a link is greater than the capacity such that it causes queue build-up. Yet, QoS is vital for commercial networks in order to be able to differentiate applications or tenants for business related reasons [6]. This is particularly true for unavoidable transient congestion events, i.e. incast.

Algorithm 4: Supply Token calculated for QoS queue i at the switch with n QoS levels serving the same egress port

```
1  Function CalculateSupplyToken(pkt):
2      inter_arrival_time ← now − last_sm_time
3      last_sm_time ← now
4      w_effective ← 0
5      for j ← 0 to n do
6          if i = j || q_size_j ≠ 0 then
7              w_effective ← w_effective + w_j
8      supply ← BW × inter_arrival_time × (w_i / w_effective)
9      demand ← pkt.size
10     sm_token ← sm_token + supply − demand
11     sm_token ← min(sm_token, MTU)
```

Fortunately, the way Bolt reports queue occupancy is QoS-agnostic such that it can generate SRC packets with the occupancy of the queue assigned by the QoS mechanism. Consequently, it would try to minimize queuing at that particular queue. Similarly, the way *PRU token* are calculated would be queue specific instead of being egress port specific. For example, if there are P ports on a switch and n QoS levels per port, the size of the register array that maintains the token values would be of $P \times n$ and flows would only be able to proactively ramp-up if another flow with the same QoS level is about to finish.

On the other hand, accounting for the *supply token* requires the service rate for the associated queue (§3.3) which would be a dynamic value depending on the current demand for different QoS levels. We identify two approaches for maintaining *supply tokens* correctly and implementing a QoS aware version of Bolt on programmable switches.

B.1 Ideal Approach

Imagine a scenario where weighted fair queuing [46] is applied for QoS purposes. Then, Bolt would need to be able to increment the *supply token* value based on the weight associated to the QoS level (w_i) and the link capacity (C) as well as the demand for each QoS level. For example, when all QoS levels have at least 1 packet in their queue, a packet arriving at QoS level i should increment the token value by $C \times w_i \times t_{inter-arr}$.

If a QoS queue is empty, its weight is distributed to other QoS levels in proportion to each level's own weight. Therefore, Bolt should adjust the *supply token* value of QoS level i based on the logic presented in Algorithm 4.

Note that in order to be able to determine the service rate of each queue, queue occupancy of other queues would be required. This requirement creates a challenge for P4 switches since only one queue's occupancy can be read at a time. A workaround to this would be to create shadow register arrays

for each priority queue where they get updated whenever a value is not being read from them. Moreover the calculation at the line 8 of Algorithm 4 requires floating point arithmetic which could be address via lookup tables.

B.2 Heuristic Approach

A simpler mechanism to enable QoS on Bolt switches would be to introduce probabilistic SRC generation where higher priority traffic has lower probability to generate a SRC packet. This would naturally keep the rates of high priority flows high while throttling others. Yet, an extensive empirical study would be required to determine the probabilities such that the queuing for all the QoS levels are bounded to some extent.

Understanding the impact of host networking elements on traffic bursts

Erfan Sharafzadeh[1], Sepehr Abdous[1], Soudeh Ghorbani[1,2]

[1]*Johns Hopkins University* , [2]*Meta*

Abstract

Conventional host networking features various traffic shaping layers (e.g., buffers, schedulers, and pacers) with complex interactions and wide implications for performance metrics. These interactions can lead to large bursts at various time scales. Understanding the nature of traffic bursts is important for optimal resource provisioning, congestion control, buffer sizing, and traffic prediction but is challenging due to the complexity and feature velocity in host networking.

We develop Valinor, a traffic measurement framework that consists of eBPF hooks and measurement modules in a programmable network. Valinor offers visibility into traffic burstiness over a wide span of timescales (nanosecond- to second-scale) at multiple vantage points. We deploy Valinor to analyze the burstiness of various classes of congestion control algorithms, qdiscs, Linux process scheduling, NIC packet scheduling, and hardware offloading. Our analysis counters the assumption that burstiness is primarily a function of the application layer and preserved by protocol stacks, and highlights the pronounced role of lower layers in the formation and suppression of bursts. We also show the limitations of canonical burst countermeasures (e.g., TCP pacing and qdisc scheduling) due to the intervening nature of segmentation offloading and fixed-function NIC scheduling. Finally, we demonstrate that, far from a universal invariant, burstiness varies significantly across host stacks. Our findings underscore the need for a measurement framework such as Valinor for regular burst analysis.

1 Introduction

Measurement studies show that traffic is bursty across a wide range of timescales in diverse contexts such as Ethernet LANs [38], WANs [56], data centers [25], and WWW traffic [21]. In particular, microsecond-scale congestion events, sometimes called *microbursts*, have been the focus of numerous measurement and control papers recently [13, 18, 19, 25, 33, 37, 72]. However, the modulating effect of host networking on traffic burstiness at various timescales is relatively less investigated. This paper addresses this gap. We ask *what causes the traffic to emerge from hosts in bursts?* Is burstiness an *scale-invariant* property of traffic, i.e., does the traffic retain its burstiness across a wide range of timescales, or do the microbursts become smooth at coarse timescales? Are canonical burst countermeasures such as TCP pacing and packet scheduling effective in curtailing bursts?

These questions have far-reaching implications for network performance and design. Controlling bursts at different timescales requires deploying mechanisms that operate at the corresponding pace. Microbursts, for instance, require real-time techniques with sub-RTT control loops, whereas bursts at longer timescales can be more effectively managed by resource provisioning techniques such as topology engineering and routing that take seconds to minutes to complete [71].

Unfortunately, studying the impact of host networking on bursts is complex. Take the Linux network stack as an example: the egress traffic that originates from the Linux kernel stack passes through many layers and optimizations before arriving at the wire. Transport protocol internals like initial window size, cumulative acknowledgments, queueing disciplines (qdiscs), driver rings, segmentation offloading, and hardware packet scheduler at the NIC all handle the traffic. All these elements and their complex interactions can play a role in forming or suppressing bursts at various timescales. These challenges are further compounded by the heterogeneity, scale, and the velocity of evolution in today's networks that constantly change in response to increasing demand and the rollout of new services [26, 46, 73].

To address this challenge, we build Valinor, a high-resolution traffic measurement framework that enables network operators to systematically and periodically dissect the elements of host networking, their impact on traffic burstiness in isolation, and importantly, their interactions with the emergent traffic patterns, all at different timescales. To ensure visibility into the impact of the software stack and the shape of the traffic on the wire through time, Valinor is composed of two main components: 1) An in-host timestamping frame-

work (Valinor-H) based on eBPF that collects egress packet metadata nearly at the last stage of software stack processing. 2) An in-network packet timestamping framework (Valinor-N) that captures packet arrival timestamps in the programmable switch data plane immediately after the NIC, and sends the timestamp data to offline servers for collection, storage, and burstiness analysis.

Our analysis of the impact of host networking on the shape of traffic using Valinor reveals some surprising results. As an example, classical work paints a unifying and consistent picture of scale-invariant burstiness, i.e, they show the same degree of variability across a wide range of timescales in a variety of different network types [21, 38, 56]. It has also been established that this scale-invariant burstiness is primarily caused by *application layer* characteristics such as long-tailed flow size distributions and is "robust": it holds for a variety of transport protocols (e.g., TCP Reno, Vegas, and flow controlled UDP) and various network configurations [23, 53].

In contrast, our investigations paint a more nuanced and complex picture. We show that burstiness at various timescales varies significantly across host configurations (hardware configurations, transport protocols, scheduling, etc.). We also show the pronounced modulating effect of below application layer elements on bursts. This implies that, for the same heavy-tailed flow size distribution, the ultimate shape of traffic on the wire depends heavily on the host configuration such as the NIC scheduler. Plus, Valinor's analysis of newer reliable transport protocols (e.g., Homa [47], DCTCP [9], and BBR [17]) reveals the high degree of variability of burstiness for these protocols. As an example, BBR is less bursty not just at fine timescales (a result that is consistent with the literature [47, 52]) but also at coarse timescales. The latter finding (new to the best of our knowledge) implies that techniques such as topology engineering [71] and multi-timescale congestion control [66]—premised on the long-range burstiness of traffic—may yield limited performance improvements under these new protocols.

Finally, given the impact of some variants of transport protocols on bursts, we quantify the effectiveness of TCP pacing and active queue management paradigms such as CoDel [49] in qdiscs (software packet schedulers) in mitigating bursts. Our results show the pronounced impact of lower-layer functions (residing in the driver and NIC) on forming the ultimate shape of traffic on the wire relative to the higher-layer software operations of the TCP/IP stack and qdiscs. As an example, active queue management techniques such as CoDel and RED in the Linux kernel try to prevent the formation of large and lasting bursts. However, our results show that their impact is effectively erased by offloading (TSO, serialization, etc.) and the NIC scheduler. For example, while in isolation, the frequency of large 300 KB bursts under CoDel is 500 times lower than FIFO, this difference is barely visible on the wire after packets pass through the multi-queue NIC with segmentation offloading. Moreover, TCP pacing enforced in the qdiscs generates between 1.8×-19× larger bursts when NIC scheduler and offloading are in action compared to when in isolation.[1] This result indicates that the countermeasures for controlling bursts should be moved further down the packet processing pipeline at the end hosts.

Our results on the variability of burstiness (based on hardware configurations, transports, etc.)—combined with the ever-evolving workloads and features in today's networks—highlight the need for periodic traffic measurement and analysis. To facilitate this, we have released Valinor's sources and artifacts as open-source software.[2] We next introduce the mathematical notions developed for capturing bursts across time, present their practical implications in networks (§2), provide some background on host networking and the design space of burst measurement frameworks (§3), and present the design of Valinor (§4) before delving into our findings (§5).

2 Background: scale-invariant burstiness

Measurements of the Internet traffic show periods of sustained greater-than-average or lower-than-average traffic rates across a wide range of timescales [21, 24, 38, 53, 56]. This behavior, sometimes called *scaling* or *self-similarity*, has broad implications for performance. In this section, we first formalize the notion of self-similarity and re-introduce the Hurst exponent, a mathematical representation of self-similarity, before discussing the implications of self-similarity and characterizing bursts at fine timescales such as microbursts.

Self-similarity. Self-similarity is a notion pioneered by Benoit Mandelbrot [45] which refers to a phenomenon where a certain property of an object (such as an image or a timeseries) is preserved with respect to scaling in space and/or time. If an object is self-similar, its parts, when magnified, resemble the shape of the whole [55].

More formally, let $(X_t)_{t\in\mathbb{Z}_+}$ be a timeseries, e.g., this timeseries can represent a traffic trace measured at some fixed time granularity. The aggregated series $X_i^{(m)}$ is defined as

$$X_i^{(m)} = 1/m(X_{im-m+1} + \ldots + X_{im})$$

In other words, X_t is partitioned into blocks of size m, their values are averaged, and i denotes the index of these blocks.

Autocorrelation is a mathematical representation of the degree of similarity between a timeseries X_t and a time-shifted version of X_t over successive time intervals. It measures the relationship between the current value of a timeseries and its

[1] Despite making the traffic bursty and hard to manage, these low-level functions are essential for reducing the processing overhead and meeting the increasingly high link rates. For example, disabling TCP segmentation offload results in a 3× increase in CPU utilization, 71% lower throughput, and a 46% increase in median packet RTTs for a multi-flow *Iperf* test. Relatedly, disabling MQ results in a 4% decline in the throughput of the same workload.

[2] https://hopnets.github.io/valinor

future values. A strong positive autocorrelation for a traffic volume timeseries, for example, suggests that if the volume is high (i.e., higher than average) now, then it is likely to be also high in the next time slot, whereas a strong negative autocorrelation implies that a high-volume slot is likely to be followed by a low volume one.

Let $r(k)$ and $r^{(m)}(k)$ denote, respectively, the autocorrelation functions (ACFs) of X_t and $X_i^{(m)}$ where k is the time shift from the original timeseries. We say that X_t is *self-similar*, or more accurately *asymptotically second-order self-similar*, if these conditions hold:

$$r(k) \sim c \times k^{-\beta} \quad (1)$$

$$r^{(m)}(k) \sim r(k) \quad (2)$$

for large k and m, where $0 < \beta < 1$ and c is a constant, and $f(x) \sim g(x)$ as $x \to a$ means that $\lim_{x \to a} f(x)/g(x) = 1$ [66]. X_t is self-similar in the sense that its ACF $r(k)$ behaves hyperbolically with $\sum_{k=0}^{\infty} r(k) = \infty$ (Eq. 1). This property is also referred to as *long-range dependence*. Equation 2 implies that for self-similar timeseries, the autocorrelation structure is preserved with respect to time aggregation.

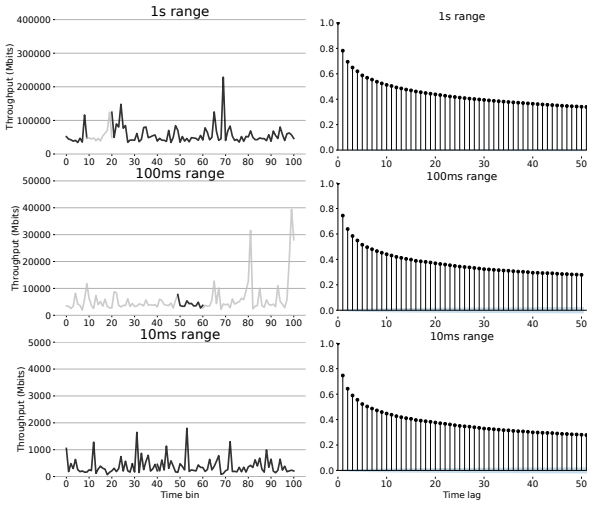
In networks, the traffic is called self-similar if the aggregated traffic over varying timescales remains bursty, regardless of the granularity of the timescale.

The Hurst exponent. Let $H = 1 - \beta/2$. H is called the *Hurst exponent*. The Hurst exponent, a number in the $(0,1)$ range that is sometimes referred to as the *index of long-range dependence*, is a measure of the long-term memory of a timeseries. It characterizes the self-similarity and long-range dependence of the timeseries:

- $0.5 < H < 1$ indicates a self-similar timeseries with long-term positive autocorrelations, i.e, a high value in the series (e.g., higher than average traffic volume) is likely to be followed by another high value. Plus, the values a long time into the future also tends to be high. It follows from Eq. (1) above that the closer H is to 1, the more long-range dependent X_t is. Conversely, H values closer to 0.5 show weaker long-range dependence.
- $H = 0.5$ indicates a completely uncorrelated series.
- $0 < H < 0.5$ indicates a *mean-reverting* timeseries, i.e., one with long-term switching between high and low values in adjacent pairs of time slots. That is, a single high value in the timeseries is likely to be followed by a low value.[3]

Various techniques (e.g., rescaled-range analysis and Periodogram [67]) exist for estimating H for an empirical dataset. Similar to the seminal work on Bellcore Ethernet traffic self-similarity [38], we use the rescaled-range, *R/S*, for the results presented in this paper. The details of this method are presented in Appendix §A.

[3] Note that the $0 < \beta < 1$ condition in the equations above is a requirement for self-similar, and not mean-reverting, series.



(a) Time-series **(b) Auto-correlation**

Figure 1: A self-similar timeseries with H=0.88.

Example: Figure 1a (the first row) shows a simulated scenario where 32 TCP connections generate a synthetic workload using Pareto flow size distribution with a mean of 4200 KB and $\alpha = 1.05$ and exponential arrivals that create 6 Gbps offered load. We plot the traffic rate (in Mbps) against time where time granularity is $1s$. A data point is the aggregated traffic volume over a $10ms$ interval. The second row of the same figure depicts the same traffic series where a randomly selected second interval in the first timeseries (the highlighted segment in the first row) is magnified by a factor of ten, resulting in a granularity of $100ms$ in the truncated timeseries. The last row similarly rescales a randomly selected slot by $10\times$. The figures show that this trace is self-similar: when traffic is aggregated over varying timescales, the aggregate traffic pattern remains bursty, regardless of the granularity of the timescale. This visual scaling is confirmed by the Hurst coefficient, $H = 0.88$, and the autocorrelation functions of the trace (Figure 1b) that show positive, slow (almost polynomial) decaying, and consistently shaped correlations across various timescales. Slow-decaying ACFs signify long-range dependence in a timeseries.

Practical implications of self-similarity. Self-similarity has broad implications on network design and performance, e.g., it is shown to lead to increased delay and loss [5, 6, 22, 42, 50, 53, 66]. We next discuss some of the key implications of self-similarity:

- **Queueing performance and buffer sizing.** Self-similarity greatly influences queueing performance. From a queueing theory standpoint, the defining characteristic of self-similarity is that the queue length distribution decays much more slowly than short-range-dependent traffic (polynomially vs. exponentially under short-range dependent traffic, e.g., Poisson processes) [66]. For strongly self-similar traffic, the mean queue

length increases with the buffer size [54]. This implies that networks with strongly self-similar traffic should deploy small buffers to control the queueing delay.

- **Throughput and latency trade-off.** Prior work [53, 54] shows that jointly provisioning low delay and high throughput is adversely affected by self-similarity.
- **Traffic prediction and burst countermeasures.** The correlation structures present in self-similar traffic can be detected and exploited to predict future traffic over timescales larger than an RTT [66].[4] Traffic prediction at long timescales, in turn, is invaluable for designing the appropriate burst countermeasures. For instance, resource provisioning techniques with control loops larger than an RTT (e.g., multi-scale congestion control [66], re-routing, and topology rewiring [71]) enhance the performance of self-similar traffic.

Microbursts. Given their ubiquity and impact, in particular in data centers, microsecond-scale traffic surges, known as *microbursts* [13, 41, 72], have been the focus of many recent proposals [4, 19, 27, 37, 41, 70].

The intensity of a microburst has often been measured implicitly based on buffer utilization, or in more extreme cases, packet loss. Related work also quantifies microbursts as the number of packets from one flow that occupy a buffer at a time snapshot [33], the evolution of switch queue length over time [63], an uninterrupted sequence of packets with gaps of smaller than a threshold [35], and/or sequence size of larger than a threshold [68]. Using metrics that are independent of network queues allows us to perform universal measurements in the entire network, i.e., both at the hosts and the switches. Yet, measurement systems intending to quantify microbursts can leverage all the above definitions to provide a holistic view of burstiness behavior.

From the technical perspective, we define a burst as the cumulative sum of packet bytes whose inter-arrivals are smaller than a threshold τ. Setting the minimum value for τ initially depends on link speeds and MTUs. For example, in a fully utilized 40 Gbps link with MTU = 1500 bytes, packets arrive 300 ns apart. Therefore, an initial τ of 2-10$\times$ of this value is small enough to detect microbursts and large enough not to miss consecutive packets from flows. To ensure that τ is not affected by the network configuration and the internal characteristics of the workloads, we repeat our measurement with a wide range of values for τ.

3 Approaches to measuring traffic bursts

In this section, we provide a brief background on host networking and present the design space of burst measurement frameworks before discussing Valinor in §4.

[4] The prediction methods span diverse domains such as regression theory, neural networks, and estimation theory [66].

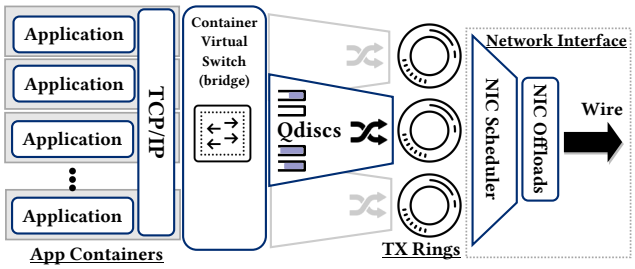


Figure 2: Conventional network processing stack architecture in a containerized Linux deployment.

3.1 Conventional host networking

Conventional network stacks consist of various processing layers glued together via several optimization techniques. In Linux, application data is passed to socket interfaces (buffering in the userspace), and then to the transport protocol processing (transport buffers, short queues [20]). Transport protocols populate *sk_buffs*,[5] a collection of data pointers and header information. After performing routing, *sk_buffs* eventually make their way towards interface *qdiscs*, the hierarchical packet schedulers in Linux. *qdiscs* operate in parallel on all CPU cores and forward the scheduled *sk_buffs* towards driver rings where another layer of buffering is performed before notifying the NIC [65]. Finally, with the conventional offloading features enabled, the NIC performs scatter/gather [58], segmentation, checksum, and sends the packets on the wire [16]. Figure 2 depicts an overview of the packet's path through the network processing pipeline in a Linux host.

3.2 Capturing timestamps

High-resolution timestamping is essential for burst analysis. Various techniques exist for capturing packet arrivals:

NIC timestamps. Hardware timestamping is available in all commodity NICs. This feature is supported by the Linux kernel via ancillary socket data. When a user requests timestamping through a socket option, the transmission timestamps are generated in the hardware before sending the packet on the wire and are eventually sent to the source socket. Therefore, the application is responsible for polling the error queue and reading the timestamps. Hardware timestamping supports most TCP and UDP connections, however, it suffers from two main shortcomings. First, if the operating system fails to poll the timestamp registers of the NIC in time, e.g., in higher packet rates, the timestamp will be overwritten by that of the next packet. Plus, modifying the network application to receive timestamps may impact the application's workload pattern, and thus must be performed with extra care.

[5] *sk_buff* stands for "Socket buffer" and is used to represent the socket data that eventually is shaped into the packet. *sk_buffs*, therefore, may contain a single or multiple packets.

Modifying networking stack software. To study realistic network traffic with higher arrival rates, hardware timestamping is not ideal due to the need to change the application internals and high overheads. An alternative solution is to directly capture the timestamps closer to the packet processing, e.g., the NIC driver, and either add the timestamps to the packet payload or re-route them to the userspace. Alas, accessing and modifying the packet data requires offloading features such as scatter-gather IO and Segmentation Offloading to be turned off. Additionally, timestamps that are at *sk_buff* granularity may not imitate the inter-packet gaps on the wire due to the intervention of lower layers.

eBPF hooks. eBPF offers a series of hooks inside the Linux kernel and the NIC driver that allows fast execution of arbitrary data plane logic. An eBPF program consists of a data plane and a control-plane code targeting a specific hook on the RX or TX path (XDP hook in the receive path of NIC driver and traffic control (*tc*) hook on the TX path of the qdisc subsystem are two examples). eBPF *tc* programs are registered to the kernel using the *tc* command and are executed inside a lightweight RISC virtual machine. eBPF also provides fast data structures that enable shared state between the kernel and the userspace. This allows us to perform burst measurements offline with any workload configuration without modifying the kernel source or packet payloads.

While eBPF relieves us from directly modifying the packet processing code in the kernel, it presents two shortcomings. First, eBPF, similar to the previous solution, works at *sk_buff* granularity since packet segmentation is almost always offloaded to the NIC. Therefore, the eBPF framework can only measure the gaps between larger chunks of data, not packets. Additionally, our measurements show that each eBPF invocation incurs up to 1 μs of delay, mostly due to memory accesses. While this overhead may be acceptable at the *sk_buff* granularity, the framework will lose its visibility into nanosecond-scale events. Ultimately, eBPF provides a convenient solution to plug into the network data path with minor interference. Making it a viable burstiness probing point on the egress path. We present the design and implementation of the Valinor eBPF framework, Valinor-H, in §4.1.

Timestamping in the switch data-plane. A holistic method to capture the behavior of all host networking components (including the NIC) is to perform measurements immediately after transmitting the packets on the wire, i.e., at the first network hop. Fortunately, the rise of programmable switch architectures with high-resolution timestamping enables capturing packet arrival timestamps and sending this data off the critical communication path for offline processing. This further ensures zero interference with the ongoing communication and the ability to track the entire egress host networking components. We describe the design of our in-network measurement system, Valinor-N, in §4.2.

Programmable NICs share many of the strengths of in-network measurements (e.g., timestamping close to the wire, low overhead, and no interference) but do not provide visibility into in-network queue occupancies. Plus, our experience with commodity DPUs [15] shows inconsistencies in the capabilities of existing devices. General-purpose SoC NICs [15] are either bound to their slow ARM CPUs or do not offer per-packet timestamping capabilities on their fast path. Due to these practical issues as well as the greater visibility that in-network measurements offer, alongside its host module, Valinor currently leverages programmable networks for capturing bursts on the wire.

4 Valinor measurement framework

For designing Valinor, we have three goals in mind:

1. Offering visibility into the host networking traffic, as well as the shape of the traffic on the wire.
2. Offering high-resolution timestamping of packet arrivals in line with the increasing link bandwidths and faster packet processing pipelines.
3. Providing insights on traffic shape and burstiness at different scales and time ranges.

We design and implement Valinor, a measurement framework that consists of two main timestamping prongs to study packet arrivals from the host and network vantage points. First, we design Valinor-H to study the host's view of its egress traffic by choosing *tc* eBPF hooks. For capturing the external picture of traffic burstiness, we design Valinor-N, a timestamping module for programmable fabric.

4.1 Valinor-H: burst measurement in hosts

Valinor-H offers visibility into the impact of the software stack on traffic, immediately before the traffic is passed to the NIC. The insight into the characteristics of the traffic entering the hardware can help the design of the functions offloaded to the NIC. This becomes increasingly important as more and more functions migrate to the NIC, driven by the dire need to reduce software overhead.[6]

Figure 3 presents the design of our eBPF framework. Our framework consists of two separate programs. The data plane program follows a strict set of C-like instructions that are executed at the *tc* qdisc, every time a *sk_buff* arrives. We design

[6] As network speeds increase at a faster pace than CPU speeds, software overhead is increasingly the performance bottleneck [52]. This has motivated the offloading of various functions such as segmentation, serialization, scheduling, and even transport protocol processing to the NIC [11, 58, 64].

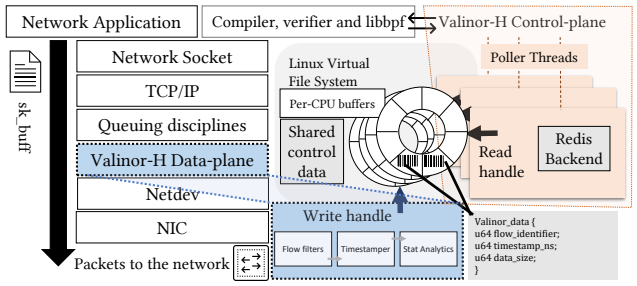


Figure 3: The eBPF measurement framework's architecture. Valinor-h consists of a data plane and a control plane, communicating via lock-free ring buffers.

circular buffers capable of storing up to 2^{16} arbitrary data entries shared with the control plane. Then, the *write handle* determines the correct location for adding new timestamp entries and updates the data structure. In the control plane, we initialize the data plane and the circular buffer and start polling the buffer for new data. The data entries carry the *sk_buff* lengths as well as the flow hash and protocol header information. The *read handle*, retrieves the timestamp entries one by one and hands them to the Redis workers for persistent storage.

One challenge that arises when using a shared data structure is synchronization between the data plane and the control plane. This scenario generally needs locking mechanisms to prevent a race condition, however, the nature of the timestamping data, being strictly increasing, lifts this heavy burden. Therefore, in the control plane, Valinor-H only reads and increments its write handle if the timestamp value is larger than the previous value read. Another synchronization issue arises when multiple CPUs attempt to store packet metadata in the shared memory. Luckily, eBPF offers per-CPU structures to prevent race conditions in the data plane. The Valinor-H control plane uses separate threads to read from per-CPU buffers simultaneously.

With the in-host measurement framework, network operators can verify the operation of higher-level network processing layers on the transmission path of the sender hosts. Valinor-H, at this stage, can capture the ingress traffic into the NIC which includes the traffic egress from qdiscs, the transport layer, and the applications. To capture the traffic behavior in the core of the network, and on a per-packet granularity, we introduce Valinor-N in the following section.

4.2 Valinor-N: in-network burst measurement

Software-based measurements in the host stack are bound to the coarse-grained *sk_buff* arrivals and are implemented before NIC functions (i.e., ring schedulers and segmentation offloads). Hence, the captured traffic behavior might not match that of the wire. To fill this gap, we introduce the in-network variant of Valinor based on programmable switch data planes. Valinor-N consists of three pieces: 1) the switch component, 2) the collector data plane, and 3) the analysis component. Valinor-N is able to I) capture per-packet arrival timestamps with zero overhead outside the critical path, II) collect and store timestamp entries arriving at line rate, and III) perform various analyses on timestamp data to provide an in-depth image of the traffic burstiness at different scales.

Valinor Switch. The switch data plane program uses *mirroring* and *timestamping* functionalities available in the PISA architecture. For every packet that matches user-defined flow filters, Valinor-N appends the arrival timestamp, queuing delay, and the size of the original packet along with its layer l-4 header information to a special IP packet with a pre-defined Valinor header. The packet is then sent to a collector server. The server machine, deployed outside the critical path of the communication between traffic endpoints, aggregates the timestamp information and performs the offline analysis.

Timestamp collection. The collector machine features a userspace packet processing framework based on DPDK that parses the arrived packets and stores the timestamp information along with flow metadata into an in-memory Redis [3] instance. Analysis of the timestamp data is then performed by querying the data store. Receiving timestamp packets at line rate and storing them in persistent storage poses several scalability challenges to the design of the collector component. To ensure that software can drain NIC buffers at line rate, we designate multiple worker threads to read and process the incoming packets. After parsing timestamp headers, the worker threads extract the timestamp data and send them to additional worker threads that are responsible for communicating with Redis. The stored metadata is then retrieved by the analysis framework to perform burst analysis using timestamps.

Valinor-N's Redis workers issue batched commands during idle periods to minimize interference with packet processing workers. We use Redis sorted sets to store timestamp entries sorted by arrival times since the packets that arrive at the collector may have a different order from the packets that arrive at the Valinor-N switch data plane. We use 1G *hugepages* and large memory pools to ensure that timestamp packets are not dropped at higher rates (Up to 40 Gbps in our testbed).

Offline timestamp processing. The last piece of Valinor's design is the offline timestamp analysis framework that queries the Redis data structures and performs analysis on timestamp data. Our framework is able to report various statistics on traffic burstiness by measuring the packet inter-arrivals. For example, in the next section, we report our findings on the scaling behavior, caused by various packet processing components in the sender machine. We

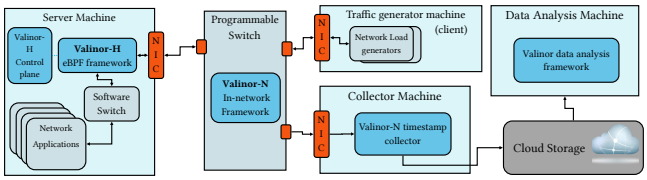


Figure 4: Deployment overview of Valinor framework.

report actual burst sizes in bytes, inter-arrival distributions, queueing delays, and various burstiness time series analyses. We implement the offline processing framework in Python.

5 Findings

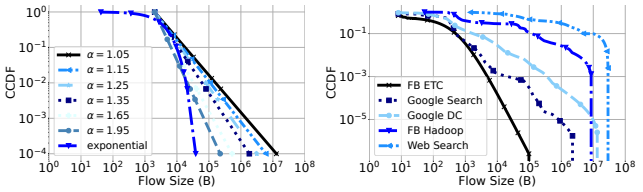
We deploy Valinor to analyze the burstiness of various workloads and configurations. Our results show:

- Host networking, largely overlooked in prior self-similarity studies, plays a major role in forming and suppressing bursts.
- Lower layers of the network processing stack (such as segmentation offloading and NIC scheduling) compromise the effectiveness of software-based traffic shaping and active queue management solutions.
- Software pacing has major limitations. For workloads with a mixture of short and large flows, lower layers of the network processing stack mask the impact of software-based traffic pacing. For workloads with very short flows, software pacing can blunt bursts but leads to a major increase in RTT and significant throughput reduction.
- NIC driver buffer sizing and process scheduling can reshape bursts.

Experiment setup. Figure 4 demonstrates how Valinor framework components come together in a basic deployment. For evaluating Valinor, we use a wide range of workload distributions. We deploy Iperf instances alongside Homa's open-source load generator [47] inside Linux containers and configure the workload generators to simulate different trace-driven workload patterns including Facebook's ETC, Google search, aggregated Google data center, DCTCP's web search, and Facebook's intra-cluster and intra-rack Hadoop traces [9,12,47,60]. Unless stated otherwise, all application containers are connected via an *OVS* [2] virtual bridge to the external interface. Our testbed consists of servers featuring Intel Xeon E5-2620 v4 processors, 64 GB of memory, and Intel XL710 40G NICs. We connect the servers via a Wedge-100 Tofino switch running Valinor-N timestamping framework. We deploy Valinor-H on Linux kernel 5.17 with the latest version of *libbpf* and *iproute2* installed. The collector machine features

Setting	Default Value	Parameter Range
Transport	TCP cubic	cubic, reno, BBR, DCTCP, Homa
Qdisc	fq	fq, fq_codel, pfifo_fast, HHF, SFQ
Byte Queue Limit	Dynamic	[100B-10MB]
MTU	1500	1500, 9000
Process scheduler	CFS	CFS, FIFO, Microquanta

Table 1: Default system configuration and tested parameter ranges.



(a) Synthetic workload **(b) Trace-driven workload**

Figure 5: Two sets of workloads used throughout the experiments. The figures show the complementary cumulative distribution functions (CCDFs) of flow sizes.

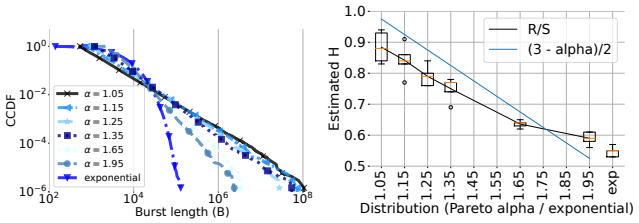
Valinor's userspace data plane based on *DPDK* v20. We disable idle states on all servers and set the frequency governor to *performance* to minimize the interference of power-saving features on networking performance. The default settings for the evaluated components are summarized in Table 1.

Finally, to calculate microburst lengths, since we use 40Gbps links, we set the burst inter-arrival threshold to 500ns for the presented results (see §2). Valinor also computes microburst lengths for other threshold settings (ranging from 5ns to 10μs). While the threshold setting impacts the size and quantity of observed bursts, we did not notice any difference in relative burstiness when comparing multiple cases.

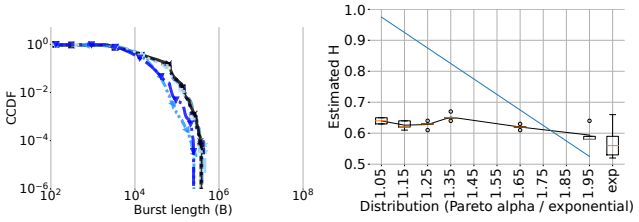
5.1 Revisiting structural causality

Where does traffic burstiness come from? Prior work [23,53,54] shows that the heavy-tailed property of the flow size distribution directly determines link-level traffic self-similarity, a phenomenon that is sometimes referred to as *structural causality*. Heavy-tailed flow size distributions are shown to be the sufficient condition for generating scale-invariant burstiness and the network stack is shown to play a negligible role in self-similarity [23,53]. For instance, for traffic generated by TCP Reno for a heavy-tailed Pareto file size distribution with the shape parameter α, there exists an almost linear relation between H and α: the estimated H is close to $(3-\alpha)/2$.[7] Heavier tailed distributions (i.e., α close to 1) are more strongly self-similar (H closer to 1). The self-similarity of traffic with heavy-tailed flow sizes is in contrast to the lack

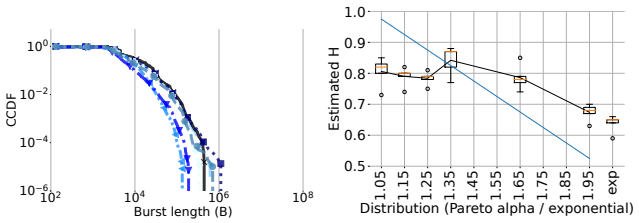
[7]The $H=(3-\alpha)/2$ relation shows the values of H predicted by the a theoretical ON/OFF model in the idealized case corresponding to a fractional Gaussian noise process with independent traffic sources with constant ON/OFF amplitude [54]. This captures an ideal self-similar process.



(a) Simulation microburst sizes (b) Simulation Hurst exponents



(c) In-host microburst sizes (d) In-host Hurst exponents



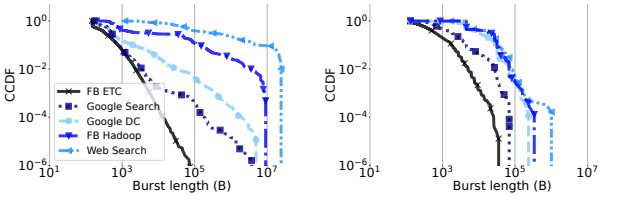
(e) In-network microburst sizes (f) In-network Hurst exponents

Figure 6: Microburst sizes and Hurst exponents of different synthetic workloads for simulated and testbed experiments. The interference of host networking elements is visible in the difference between the three scenarios.

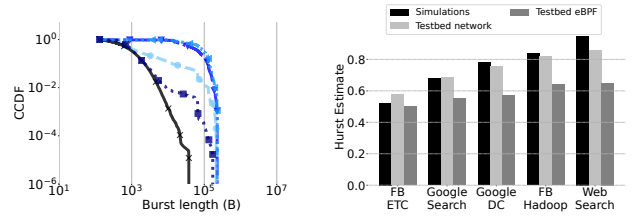
of correlation structures for short-tailed flow size distributions such as an exponential distribution (H close to 0.5).

We first replicate this result using OMNET [1], an extensively used simulator [8, 14, 47], and observe an almost linear relation between α and H—consistent with the findings of prior work [53], the estimated H values closely track the $(3-\alpha)/2$ line. In a setup where the two simulated servers are connected via a network switch, we establish 32 long-running TCP connections and use Pareto and exponential flow size distributions (Figure 5a shows the flow size distributions). To achieve a target offered load of 6 Gbps, flows are initiated exponentially with a mean interarrival time of 87μs. We repeat each experiment five times. In the box and whisker plots, each box depicts the 1st and 3rd quartiles, the whiskers represent the upper and lower extremes, the circles are outlier points, and the orange dashes show the median Hurst estimates. Figure 6b shows that heavy-tailed flow size distributions generate self-similar traffic. Figure 6a shows that these distributions also result in larger microbursts with heavier tails.

Next, we repeat the above scenario in a testbed, using Valinor to analyze burstiness after the software stack and on the wire. Using Valinor-H for in-host analysis, we observe that



(a) Simulation microburst sizes (b) In-network microburst sizes



(c) In-host microburst sizes (d) Estim. Hurst exponents

Figure 7: Self-similarity and microburst sizes vary across workloads and between testbed and simulation results. Positioned before the NIC, Valinor-H captures a smoother snapshot of traffic than in-network measurements.

the impact of the heavy-tailed distributions on self-similarity is barely visible at this stage with distributions with varying α parameters behaving similarly and close to a light-tail exponential distribution (Figure 6d), e.g., the software stack greatly diminishes the degree of self-similarity of heavy-tailed Pareto distribution with $\alpha = 1.05$ from $H = 0.88$ in the simulations (Figure 6b) to $H = 0.64$ at the eBPF hook (Figure 6d). We observe a similar effect on the microburst size distributions that are much more similar across different workloads and have shorter tails (Figure 6c).

We next use Valinor-N for analyzing traffic as observed on the wire. The patterns again change in interesting and non-uniform ways. Similar to in-host measurements, the in-network measurements indicate that the influence of flow size on self-similarity is lower than the simulated experiments, e.g., $H = 0.80$ and $H = 0.78$ for $\alpha = 1.05$ and $\alpha = 1.65$, respectively, on the wire in the testbed experiments compared to $H = 0.88$ and $H = 0.63$ for the same workloads in the simulated experiments (Figure 6f). The more amplified long-range burstiness in the network compared to in-host experiments is due to the intervention of driver and NIC functions (such as segmentation offloading scheduling) that reside below Valinor-H. We investigate the roles of these functions in §5.3. Figure 6e shows that the flow size distribution has a relatively subdued impact on the ultimate size of microbursts on the wire once the traffic traverses the host networking stack.

Summary: The shape of the traffic in the testbed experiments (in-network and in-host) is substantially different compared to the simulated experiments with identical setups. This suggests that host networking elements (e.g., qdiscs, process schedulers, and NIC schedulers, not modeled in common simulators) alter burstiness.

5.2 Impact of workloads

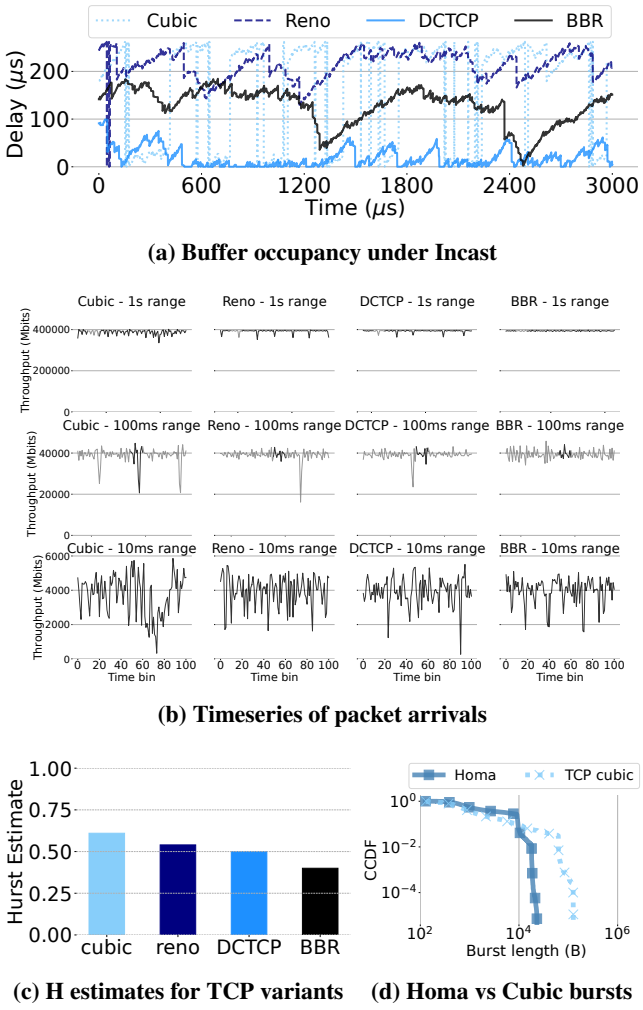
Next, we repeat the above experiments (using both the simulator and the testbed) by replaying the traces of five classes of workloads from [47]: (1) Facebook's ETC workload, (2) Google search workload, (3) Google's aggregated internal data center workload, (4) Facebook's Hadoop workload, (5) DCTCP's web search workload [9]. Figure 5b shows the flow size distributions of these traces. Similar to the previous experiments, in simulations, there exists a direct correlation between the flow size distributions, self-similarity, and the burst lengths (Figure 7). In the testbed, however, the difference in burst lengths starts to fade away as host networking components come into play. We also observe that the scaling behavior varies substantially across different workloads and between the simulated and testbed experiments. Hurst coefficients are larger for the more heavy-tailed distributions in the network but mostly homogeneous before reaching the driver. For example, the self-similarity estimates for the ETC workload (p99th flow size = 1.8 KB), the Google DC workload (p99th flow size = 31 KB), and the web search workload (p99th flow size = 27 MB) are 0.57, 0.75, and 0.85, respectively for in-network measurements and 0.50, 0.57, and 0.65, respectively for in-host measurements.

5.3 Sources and implications of burstiness

The previous section shows the aggregate impact of host networking elements on bursts. In this section, we measure the impact of each element, starting with the transport layer and moving to the elements that operate *below* the TCP/IP stack (e.g., qdiscs) and *in parallel* to it (e.g., the process scheduler).

5.3.1 Transports and congestion control

Starting with transports, we evaluate four TCP congestion control variants under a mixture of background traffic and a small-scale incast traffic pattern where two sender machines target one receiver. The background traffic consists of two *iperf* flows each taking 18Gbps of bottleneck link bandwidth. The incast traffic follows the map-reduce workload size distribution. For this experiment only, we run both the workload generators and the applications outside the container environment. Figure 8a shows how TCP Cubic [28], TCP Reno, DCTCP [9], and BBR [17] react to queue buildups in the network. Compared to Reno, TCP Cubic (the default congestion control setting in recent versions of Linux kernels) uses a more aggressive function for increasing its congestion window upon receiving acknowledgments. Therefore, it experiences larger queueing oscillations than Reno. BBR uses round-trip times to adjust its transmission window and varies its pacing rate to keep the in-flight bytes near its estimated bandwidth-delay product. Thus, it experiences a more steady queueing behavior while trying to keep the buffer half full.



(a) Buffer occupancy under Incast

(b) Timeseries of packet arrivals

(c) H estimates for TCP variants

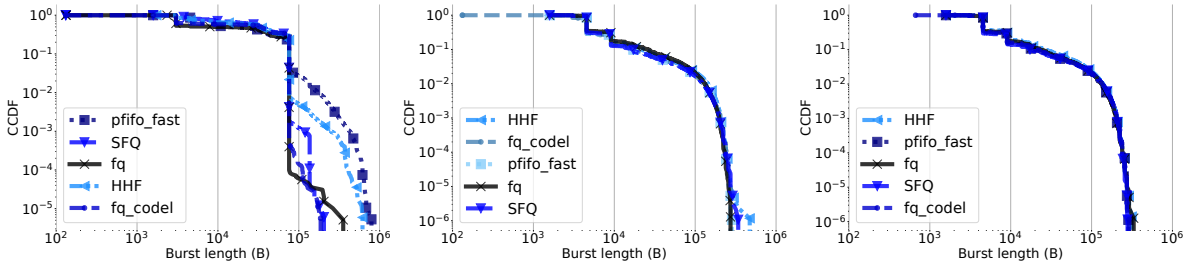
(d) Homa vs Cubic bursts

Figure 8: (a) Valinor captures the in-network buffer occupancy for different transport protocols. (b), (c) Timeseries and H coefficients show that burstiness (at both short and long timescales) varies significantly across transport protocols. (d) A receiver-driven transport, Homa, is less bursty than TCP Cubic.

Finally, DCTCP uses explicit congestion notifications from switches to maintain consistently low queuing.

Figure 8b presents the throughput timeseries of the four congestion control variants at different timescales followed by their Hurst exponent estimates in Figure 8c. With the help of pacing and RTT estimations, BBR is able to maintain a steady throughput and a non-bursty traffic shape, reflected by $H = 0.40$. On the other hand, Cubic's less conservative transmissions incur a self-similarity estimate of 0.60.

Finally, we deploy Homa's kernel module [47] as a representative implementation of receiver-driven transports in the Linux kernel. In receiver-driven transports, the destination initiates more packets by issuing *grant* control packets for the sending host. In our setup, Homa sends the first 90 KB of each flow unscheduled as an attempt to initiate the communication and retrieve the path's congestion status. The



(a) SQ w/out Offloading **(b) SQ w/ Offloading** **(c) MQ w/ Offloading**

Figure 9: Burst behavior of Linux queueing disciplines in the absence and presence of offloading (Offloading) and NIC scheduling (SQ=single-queue, MQ=multi-queue). Both MQ and TCP segmentation offload compromise the intended shape of software packet scheduling.

following packets are then scheduled using grants. Due to its limited implementation scope, the Homa module is not able to achieve line rate performance. Therefore, we limit our observation to the map-reduce workload tuned down to 6 Gbps offered load. Figure 8d presents the burst lengths for Homa and TCP Cubic as observed by Valinor-H eBPF framework. We observe that the p99 burst length under Homa is 9× lower than Cubic which might reflect two facts. First, unlike Cubic which sends up to 64 KB long data chunks, Homa's prepared *sk_buff* chunks are mostly as large as its MTU (9 KB in this experiment). This is also due to the fact that Homa kernel module is not making use of TSO because of certain Intel NIC limitations. Secondly, Homa uses pacing to keep the NIC fully saturated in its Linux implementation which further controls the spacing between its transmissions [52]. Combined, these factors result in Homa's less bursty behavior compared to TCP Cubic, not just at small timescales (Figure 8d) but also at large timescales ($H = 0.54$ for Homa *vs.* 0.62 for Cubic). However, we suspect a different behavior from Homa on different setups that can make use of NIC offloading.

5.3.2 Software switching

Linux leverages queueing disciplines (*qdiscs*) to enforce scheduling among segments originating from different applications in the system. If generic segmentation offload is not in use, *qdiscs* are the last software components to decide the order of data entities on NIC's FIFO rings. We study five representative queueing disciplines implemented in Linux:

1) Fair queue (fq) is the default scheduler in recent Linux kernels and is mainly used to enforce pacing on a per-flow (per socket) basis. The appropriate pace among flows is either explicitly enforced via socket options, or is determined by the TCP congestion control (e.g., BBR). By default, *fq* uses deficit round-robin with a default quantum of 3028 bytes to drain flow queues, with an initial quantum equalling TCP's initial 10-packet window.

2) fq_CoDel. The controlled delay (CoDel) algorithm, combined with fair queue, enforces CoDel on per-flow sub-queues. CoDel, a more recent AQM algorithm, uses packet sojourn time inside each flow queue to detect slow flows and prevents the queueing delay to exceed a user-specified target by dropping excess traffic.

3) Stochastic Fair Queueing (SFQ) extends flow-queuing with random-early marking/drop semantics with small default queue sizing to control the queueing delay. Similar to *fq*, it uses round-robin scheduling on per-flow sub-queues. SFQ uses a default deficit of one MTU.

4) pfifo_fast is a First-In First-Out priority queue. Higher priority packets are distinguished by their Type of Service (TOS) fields in IP headers which are set by upper layers.

5) Heavy Hitter Filter (HHF) attempts to identify and separate short flows from heavy hitters to prevent head-of-the-line blocking and increased delays for latency-sensitive flows. Such flows are given a higher deficit compared to heavy hitters in each transmission round.

We study qdiscs under three scenarios: First, to see the actual contribution of qdiscs to the traffic shape, we disable segmentation offload and serialization offload and limit the number of the transmit rings to one (single-queue). Segmentation is the process of breaking large *sk_buffs* into MTU-sized segments and is usually deferred to the last processing stages to reduce CPU utilization and improve flow performance. Segmentation offload can either be performed in the hardware (TCP Segmentation Offload or TSO) or just before passing the data to the hardware (Generic Segmentation Offload or GSO). Additionally, in a multi-queue architecture, the network stack communicates to the NIC via separate ring buffers pinned to each CPU core to reduce inter-core communication overheads and improve throughput. When enabled, a (reportedly, round-robin [65]) packet scheduler in the hardware will decide the order in which packets are drained from ring buffers.

Initially, we run 1000 Iperf instances spread across 200 containers, simulating the map-reduce workload on the single-queue server without offloading. Figure 9a demonstrates how, *in isolation*, per-flow queuing can significantly shorten the size of egress bursts. Techniques such *pfifo_fast*, and HHF use one large buffer containing packets from all egress flows, allowing multiple data segments of one flow to be enqueued simultaneously. On the other hand, per-flow queueing allows

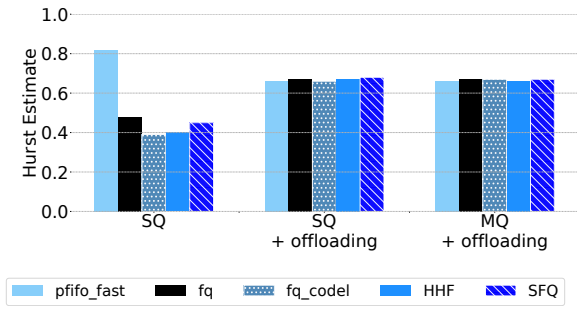


Figure 10: Hurst estimates for different queueing disciplines.

the scheduler to interleave among packets of different flows, primarily to maintain fairness and prevent head-of-the-line blocking [62].

To verify the impact of round-robin scheduling on blunting bursts, we repeat the same experiment with *fq* qdisc, increasing the per-flow deficit from one MTU (1514 bytes) to 16 MTUs, and observe a linear correlation between fq deficits and burst lengths. For example, the 90th percentile of burst lengths under a deficit of 16 packets is increased to 25 KB from 13 KB under that of 8 MTUs (92% increase).

While qdisc is the last layer to perform packet scheduling in the software, the traffic ultimately often passes through segmentation offloading and NIC scheduling before reaching the wire. Is the impact of qdiscs on the wire preserved *after the interaction* with these lower layers? To investigate, we enable all the offloading features and perform our measurements again. Figure 9b demonstrates the impact of offloading segmentation and serialization on lengthening the egress bursts. With TSO at work, qdiscs no longer serve packets. Instead, they schedule between dynamically sized *sk_buffs*. Hardware offloading then helps increase the throughput by nearly 50% for all cases while moving the buffering to the hardware where large segments are broken into MTU-sized packets and sent on the wire. This significantly undermines qdisc's decisions on shaping the traffic. With offloading in action, the median burst sizes for *fq*, *fq_codel*, and *pfifo_fast* are, 132 KB, 127 KB, and 127 KB, respectively. While without offloading, these systems experienced a median burst length of 76 KB, 76 KB, and 172 KB[8], respectively.

To further ruffle the output of qdiscs, we enable the default multi-ring root qdisc which assigns a separate qdisc instance to each CPU core and enables the NIC scheduler to perform last-level scheduling on transmit rings (multi-queue architecture). Figure 9c presents the outcome. *With NIC scheduling and segmentation offloading at work, the shape of the qdisc's outgoing traffic is barely preserved on the wire.* That is because, NICs are equipped with internal round-robin schedulers to drain the software rings, further reducing the chances of creating long bursts. Finally, Figure 10 demonstrates the estimated Hurst exponents for the three scenarios. Without segmentation offloading, the degree of burstiness is considerably reduced ($H < 0.5$) for all but one case. Only *pfifo_fast* which does not offer any form of fair queueing suffers from heavier burstiness ($H = 0.8$) under the single-queue scenario.

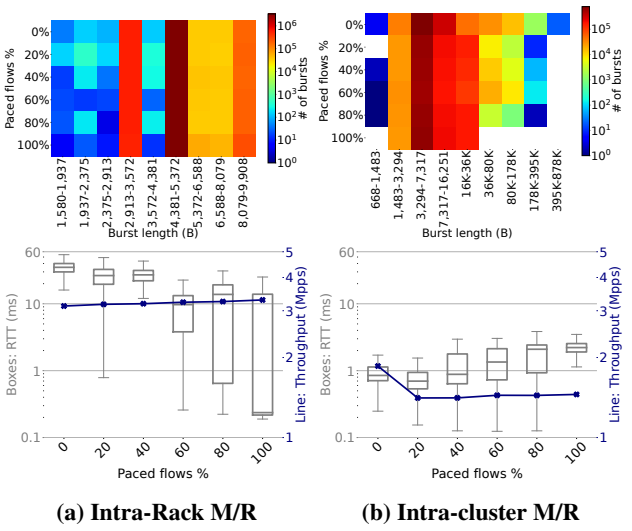
Implications of disabling offloading and multi-ring scheduling. Apart from burstiness, both offloading and NIC scheduling have a profound impact on flow performance metrics. Our measurements demonstrate that disabling TCP segmentation offload for a workload consisting of 1000 same-size flows results in 71% decline in median flow throughput, 46% increase in median packet RTTs, and 3× increase in sender CPU utilization. Therefore, disabling offloading, in order to enable software control is not always a viable option. Multi-queue NICs are also considered a quick solution with potential side effects. While enabling multi-queue reduces resource contention, they can increase response times and are usually fixed-function [65].

5.3.3 Software pacing

The above observations raise another important question on host networking design decisions. While many congestion control techniques [7, 17, 37, 47, 57] advocate for pacing in order to achieve accurate control over in-transit data, existing pacing implementation in the Linux kernel is deeply away from the wire, at fq qdisc. *Are qdiscs a suitable place for enforcing pacing?* To investigate, we repeat the map-reduce (M/R) workloads on the server with both offloading and NIC scheduling enabled and observe that *for workloads with large flows (intra-rack M/R), pacing doesn't have a significant impact on burstiness, and for those with short flows (intra-cluster M/R), pacing results in throughput reduction.* Overall, our results highlight the limitations of software pacing for data center workloads.

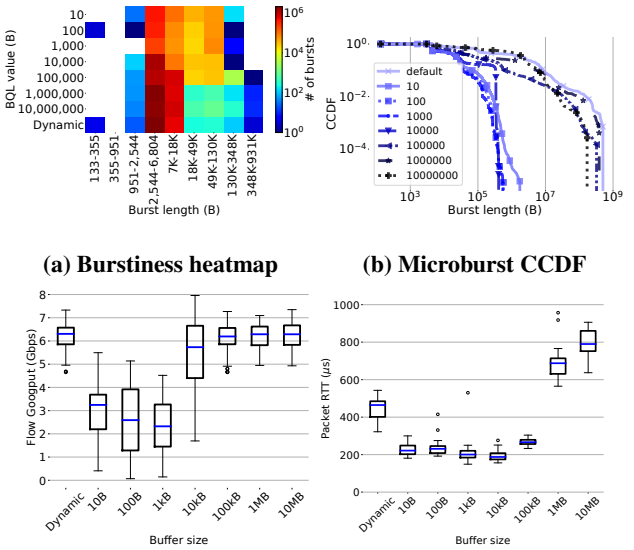
Concretely, we configure *fq* to pace 200 flows based on their fair share of bandwidth (200 Mbps), and gradually increase the portion of the flows that are counted as heavy hitters from 0% (no flow is paced) up to 100% (all flows are paced). Figure 11 compares the bursts for (a) workload with mostly large flows and (b) workload with a mix of small and large flows. In the former workload, we observe that while the impact of pacing ratio is less evident, pacing allows for better bandwidth allocation and the line rate is preserved for all rows. On the other hand, in the latter workload, the throughput is reduced by 22% under pacing. This is because short flows are not able to make up for the freed bandwidth that pacing creates. We also compare packet RTTs and find that pacing large heavy-hitters helps reduce median RTTs by two orders of magnitude as short flows experience less head-of-the-line blocking. This behavior changes in the intra-cluster workload as we do more pacing, as the increased RTT of paced flows drives the overall median RTT up by 160%. Further details on the theoretical analysis of burstiness under software pacing can be found in Appendix §B.

[8] pfifo_fast combined with offloading can exacerbate burstiness as both layers are prone to creating large, uncontrolled bursts.



(a) Intra-Rack M/R (b) Intra-cluster M/R

Figure 11: Software pacing is workload dependent. For workloads consisting of large flows, its impact on smoothing bursts is unmade by lower layers. For workloads with both short and long flows, it reduces throughput.



(a) Burstiness heatmap (b) Microburst CCDF

(c) Flow goodput (d) Packet RTTs

Figure 12: Larger BQL settings produce longer bursts. Also, the dynamic BQL algorithm presents a similar behavior to a large static ring size.

5.3.4 Byte Queue Limits

Linux kernel employs buffers at various stages of network stack processing to streamline the data movement among various components. The NIC driver queue is the last buffering stage before triggering the hardware. A fixed-size driver queue (a.k.a., TX ring) would ensure that the NIC can always find ready-to-send packets without communicating with the OS. However, due to the unpredictable size of packet buffers in the Linux kernel (ranging from 64B up to tens of kilobytes), the queueing time will considerably add to the overall RTT of packets. To prevent that, OS developers propose a dynamic bound on TX rings that adjusts the limit based on NIC's transmission rate and the availability of data in the TX rings [29]. To that end, after every transmission, BQL uses time intervals to check whether the NIC was starved in previous transmissions. If the NIC was not fully utilized during any interval while data was available at higher layers, the BQL algorithm increases the limit on the TX ring. Otherwise, if the NIC was fully busy, the BQL is decreased to reduce the queueing overheads. Enforcing smaller queue limits also ensures that the main queuing occurs at the qdisc-level where more advanced queuing disciplines can be employed.

Apart from Linux, NIC buffer sizing is also an important consideration for kernel-bypass runtimes that are less inclined to distribute TX processing among multiple ring buffers [36, 51]. Figure 12 demonstrates the impact of driver queue size on performance and burstiness. Intuitively, as we increase the size of the driver's buffer, we greatly increase the queueing time experienced by egress traffic, therefore, preventing the bursts of packets from arriving at the NIC. On the other hand, a larger driver queue is more prone to creating longer bursts as it is more susceptible to triggering segmentation offload (99th percentile burst length for 1 KB buffers and 1 MB buffers are 68 KB and 9 KB, respectively, but 99.99^{th} lengths shift to 68 KB and 86 KB, respectively). The microburst length distributions in Figure 12b further suggest that the default dynamic buffer sizing algorithm tends to maintain larger ring buffers which leads to longer bursts.

5.3.5 Linux process scheduling

Apart from the network stack, the operating system features various internal components that might change the traffic shape. For example, Linux offers a range of process scheduling classes suited for various use cases:

Completely Fair Scheduler (CFS) is the default process scheduling class in Linux which aims at achieving fairness among active processes in the system while maintaining responsiveness for I/O-bound applications. when running a mix of compute-intensive and network-intensive workloads, CFS attempts to proportionally share the CPU among workloads leading to longer response times [39].

Real-time scheduler supports two policies: *Round-robin* and *First-In-First-Out (FIFO)* scheduling. Both policies give strict priority to I/O-bound applications (if configured properly). By default, the round-robin policy preempts high-priority processes every 100ms while the FIFO policy is non-preemptive. We also deploy Microquanta [46] a semi-real-time scheduling class with microsecond time precision.

Valinor's picture of traffic burstiness is consistently similar when the network application is running alone as Hurst estimates vary between 0.51 and 0.54 for all the schedulers. How-

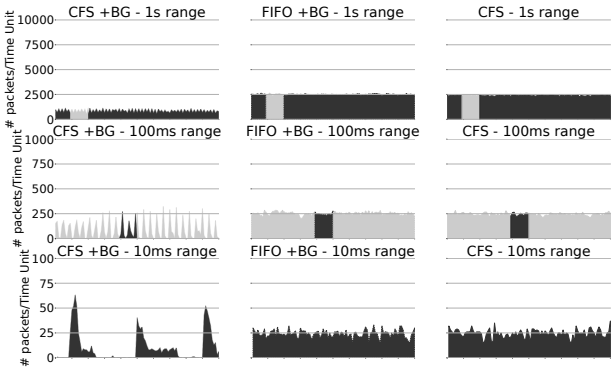


Figure 13: Impact of process scheduling on traffic bursts.

ever, when the background process is introduced, a course-grained process scheduler like CFS must enforce fair CPU time sharing, resulting in the leap of its H estimate to 0.74 while other schedulers are able to schedule the network application's threads in short timescales and result in smooth transmissions. To validate the self-similarity estimates, we plot the time series of CFS (running only the network workload), CFS+BG (running the network workload alongside background threads), and the real-time FIFO scheduler running both applications (FIFO+BG) in Figure 13. The self-similar nature of the CFS+BG scenario is noticeable in the leftmost column as CFS causes the network packets to be sent in larger chunks, causing intermittent but larger bursts at short time spans.

6 Related work

Traffic self-similarity. A large group of studies rely on quantifying bursts by using the notion of self-similarity in the time series [5, 24, 38, 50, 53–55] but overlook the role of host networking on shaping bursts. Valinor leverages the theoretical frameworks developed in these works to uncover *the impact of host networking on bursts.*

Detecting bursts. A growing number of proposals try to identify *what flows* are bursty [18, 19, 40, 48] but they cannot identify *why* those flows are bursty. Crucially, they cannot identify the elements on the traffic path that contribute to or blunt traffic burstiness. Frameworks such as BurstRadar [33] and SynDB [34] rely on buffer congestion or external triggers to capture packet arrivals, which prevents them from capturing long-range dependency patterns in host egress traffic.

Similar to Valinor, a few proposals study the causes of bursts. Some papers pinpoint transport protocol internals such as segmentation, slow start, bulk acknowledgments, and fast re-transmit as potential sources of bursts at the source level [10, 31, 69]. Another category of works study the impact of offloading techniques like segmentation offload on microbursts [35, 72]. Specifically, [35] investigates the impact of application behavior, operating system syscalls, and NIC offloading features on both sender and receiver hosts on burstiness and further show that burstiness imposed by TCP segmentation offload can marginally be controlled by configuring the kernel's maximum GSO size. Compared to these studies, Valinor has a broader scope; it studies the impact of various host elements (not just transport protocols), the effects of low-level offloading mechanisms on software scheduling and pacing, and bursts at various timescales (not just microsecond-scale). Finally, [25] introduces Millisampler, a host-centric burst characterization tool to study the impact of service placement on buffer contention and packet loss. Valinor uses its switch framework to detect synchronized flow arrivals at points of interest and unlike Millisampler which operates at *sk_buff* granularity, can attribute bursts at packet resolution. We believe that Valinor and Millisampler combined can assist data center network operators in accurately detecting the sources of bursty traffic at various timescales.

Burst control. A large and growing number of proposals [9, 27, 32, 37, 41, 43, 44, 59, 61, 62, 70] focus on *controlling* bursts, e.g., via rate-limiting at the switch [44], fine-grained pacing [61], and high-precision transport protocols [37, 41]. These studies are orthogonal to Valinor. Understanding the temporal properties of bursts and the causal mechanisms contributing to burstiness will benefit the design of effective burst control mechanisms.

7 Conclusions

We presented the design of Valinor, a burst measurement framework that consists of an in-host eBPF framework and an in-network timestamping module for programmable switches. Valinor can capture burstiness at different scales (ranging from nanoseconds to seconds). We use Valinor to demonstrate how host networking elements affect bursts. We show that the scaling behavior of traffic at long timescales and burstiness at fine timescales vary significantly across different host networking configurations (process schedulers, congestion control algorithms, single vs. multi-queue NICs, etc.) and across different classes of practical workloads. In particular, we show the impact of hardware-resident functions (e.g., NIC schedulers) that are largely overlooked in characterizing burstiness. This variability of burstiness and the implications of bursts on performance underscore the need for measurement systems to perform periodic burst analysis.

Acknowledgements

We would like to thank our shepherd, Srinivas Narayana, and the anonymous NSDI reviewers for their insightful feedback. We would also like to thank John Ousterhout for his input and feedback, and Xin Jin for his equipment support. This project was partially supported by an Intel Fast Forward award, a Facebook faculty research award, and NSF CNS grant 1910821.

References

[1] Omnet++ simulator. https://omnetpp.org/, 2022.

[2] Open vswitch. http://openvswitch.org/, 2022.

[3] Redis: an open source, in-memory data structure store. https://redis.io, 2022.

[4] ABDOUS, S., SHARAFZADEH, E., AND GHORBANI, S. Burst-tolerant datacenter networks with Vertigo. In *CoNEXT* (2021).

[5] ADAS, A., AND MUKHERJEE, A. On resource management and qos guarantees for long range dependent traffic. In *INFOCOM* (1995).

[6] ADDIE, R. G., ZUKERMAN, M., AND NEAME, T. Fractal traffic: measurements, modelling and performance evaluation. In *INFOCOM* (1995).

[7] AGGARWAL, A., SAVAGE, S., AND ANDERSON, T. Understanding the performance of TCP pacing. In *INFOCOM* (2000).

[8] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: distributed congestion-aware load balancing for datacenters. In *SIGCOMM* (2014).

[9] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[10] ALLMAN, M., AND BLANTON, E. Notes on burst mitigation for transport protocols. *SIGCOMM CCR* (2005).

[11] ARASHLOO, M. T., LAVROV, A., GHOBADI, M., REXFORD, J., WALKER, D., AND WENTZLAFF, D. Enabling programmable transport protocols in high-speed NICs. In *NSDI* (2020).

[12] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *SIGMETRICS* (2012).

[13] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[14] BESTA, M., SCHNEIDER, M., KONIECZNY, M., CYNK, K., HENRIKSSON, E., GIROLAMO, S. D., SINGLA, A., AND HOEFLER, T. FatPaths: Routing in supercomputers and data centers when shortest paths fall short. In *SC* (2020).

[15] BURSTEIN, I. Nvidia data center processing unit (DPU) architecture. In *IEEE HCS* (2021).

[16] CAI, Q., CHAUDHARY, S., VUPPALAPATI, M., HWANG, J., AND AGARWAL, R. Understanding host network stack overheads. In *SIGCOMM* (2021).

[17] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: congestion-based congestion control. *ACM Queue* (2016).

[18] CHEN, X., FEIBISH, S. L., KORAL, Y., REXFORD, J., AND ROTTENSTREICH, O. Catching the microburst culprits with snappy. In *SelfDN* (2018).

[19] CHEN, X., FEIBISH, S. L., KORAL, Y., REXFORD, J., ROTTENSTREICH, O., MONETTI, S. A., AND WANG, T.-Y. Fine-grained queue measurement in the data plane. In *CoNEXT* (2019).

[20] CORBET, J. TCP small queues. https://lwn.net/Articles/507065/, 2012.

[21] CROVELLA, M. E., AND BESTAVROS, A. Self-similarity in World Wide Web traffic: evidence and possible causes. *ToN* (1997).

[22] DUFFIELD, N. G., AND O'CONNELL, N. Large deviations and overflow probabilities for the general single-server queue, with applications. In *Mathematical Proceedings of the Cambridge Philosophical Society* (1995).

[23] FELDMANN, A., GILBERT, A. C., HUANG, P., AND WILLINGER, W. Dynamics of ip traffic: A study of the role of variability and the impact of control. In *SIGCOMM* (1999).

[24] GARRETT, M. W., AND WILLINGER, W. Analysis, modeling and generation of self-similar VBR video traffic. *SIGCOMM CCR* (1994).

[25] GHABASHNEH, E., ZHAO, Y., LUMEZANU, C., SPRING, N., SUNDARESAN, S., AND RAO, S. A microscopic view of bursts, buffer contention, and loss in data centers. In *IMC* (2022).

[26] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or die: high-availability design principles drawn from googles network infrastructure. In *SIGCOMM* (2016).

[27] GOYAL, SHAH, ZHAO, NIKOLAIDIS, AND OTHERS. Backpressure flow control. In *NSDI* (2022).

[28] HA, S., RHEE, I., AND XU, L. CUBIC: a new TCP-friendly high-speed TCP variant. *SOSR* (2008).

[29] HERBERT, T. bql: Byte Queue Limits. https://lwn.net/Articles/469652/, 2011.

[30] HURST H. E. Long-Term storage capacity of reservoirs. *Trans. of the American Soc. of Civil Eng.* (1951).

[31] JIANG, H., AND DOVROLIS, C. Source-level IP packet bursts. In *IMC* (2003).

[32] JIN, P., GUO, J., XIAO, Y., SHI, R., NIU, Y., LIU, F., QIAN, C., AND WANG, Y. PostMan: Rapidly mitigating bursty traffic by offloading packet processing. In *SoCC* (2019).

[33] JOSHI, R., QU, T., CHAN, M. C., LEONG, B., AND LOO, B. T. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *APSys* (2018).

[34] KANNAN, P. G., BUDHDEV, N., JOSHI, R., AND CHAN, M. C. Debugging transient faults in data centers using synchronized network-wide packet histories. In *NSDI* (2021).

[35] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., AND PORTER, G. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *CoNEXT* (2013).

[36] KAUFMANN, A., STAMLER, T., PETER, S., SHARMA, N. K., KRISHNAMURTHY, A., AND ANDERSON, T. TAS: TCP acceleration as an OS service. In *EuroSys* (2019).

[37] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M. G., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., WETHERALL, D., AND VAHDAT, A. Swift: delay is simple and effective for congestion control in the datacenter. In *SIGCOMM* (2020).

[38] LELAND, W. E. On the self-similar nature of Ethernet traffic (extended version). *ToN* (1994).

[39] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *SoCC* (2014).

[40] LI, Y., MIAO, R., KIM, C., AND YU, M. FlowRadar: a better NetFlow for data centers. In *NSDI* (2016).

[41] LI, Y., MIAO, R., LIU, H. H., ZHUANG, Y., FENG, F., TANG, L., CAO, Z., ZHANG, M., KELLY, F., ALIZADEH, M., AND YU, M. HPCC: high precision congestion control. In *SIGCOMM* (2019).

[42] LIKHANOV, N., TSYBAKOV, B., AND GEORGANAS, N. D. Analysis of an atm buffer with self-similar (" fractal") input traffic. In *INFOCOM* (1995).

[43] LIM, H., BAI, W., ZHU, Y., JUNG, Y., AND HAN, D. Towards timeout-less transport in commodity datacenter networks. In *EuroSys* (2021).

[44] LIU, K., TIAN, C., WANG, Q., ZHENG, H., YU, P., SUN, W., XU, Y., MENG, K., HAN, L., FU, J., DOU, W., AND CHEN, G. Floodgate: taming incast in datacenter networks. In *CoNEXT* (2021).

[45] MANDELBROT, B. B. Self-Affine Fractals and Fractal Dimension. *Physica Scripta* (1985).

[46] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RUBOW, E., RYAN, M., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: A Microkernel Approach to Host Networking. In *SOSP* (2019).

[47] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM* (2018).

[48] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM* (2016).

[49] NICHOLS, K., AND JACOBSON, V. Controlling Queue Delay: A modern AQM is just one piece of the solution to bufferbloat. *ACM Queue* (2012).

[50] NORROS, I. A storage model with self-similar input. *Queueing systems* (1994).

[51] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI* (2019).

[52] OUSTERHOUT, J. A Linux kernel implementation of the Homa transport protocol. In *ATC* (2021).

[53] PARK, K., KIM, G., AND CROVELLA, M. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *ICNP* (1996).

[54] PARK, K., KIM, G., AND CROVELLA, M. E. Effect of traffic self-similarity on network performance. In *Performance and Control of Network Systems* (1997).

[55] PARK, K., AND WILLINGER, W. Self-similar network traffic: An overview. *Self-Similar Network Traffic and Performance Evaluation* (2000).

[56] PAXSON, V., AND FLOYD, S. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM ToN* (1995).

[57] PRAKASH, P., DIXIT, A., HU, Y. C., AND KOMPELLA, R. The TCP outcast problem: exposing unfairness in data center networks. In *NSDI* (2012).

[58] RAGHAVAN, D., LEVIS, P., ZAHARIA, M., AND ZHANG, I. Breakfast of champions: towards zero-copy serialization with NIC scatter-gather. In *HotOS* (2021).

[59] REZAEI, H., AND VAMANAN, B. Superways: A datacenter topology for incast-heavy workloads. In *WWW* (2021).

[60] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[61] SAEED, A., DUKKIPATI, N., VALANCIUS, V., THE LAM, V., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM* (2017).

[62] SANAEE, A., SHAHINFAR, F., ANTICHI, G., AND STEPHENS, B. E. Backdraft: a lossless virtual switch that prevents the slow receiver problem. In *NSDI* (2022).

[63] SHAN, D., REN, F., CHENG, P., SHU, R., AND GUO, C. Micro-burst in data centers: observations, analysis, and mitigations. In *IEEE ICMP* (2018).

[64] STEPHENS, B., AKELLA, A., AND SWIFT, M. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI* (2019).

[65] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue NICs. In *ATC* (2017).

[66] TUAN, T., AND PARK, K. Multiple time scale congestion control for self-similar network traffic. *Performance Evaluation* (1999).

[67] WERON, R. Estimating long-range dependence: finite sample properties and confidence intervals. *Physica A: Statistical Mechanics and its Applications* (2002).

[68] WOODRUFF, J., MOORE, A. W., AND ZILBERMAN, N. Measuring burstiness in data center applications. In *ACM BS* (2019).

[69] WU-CHUN FENG, TINNAKORNSRISUPHAP, P., AND PHILIP, I. On the burstiness of the TCP congestion-control mechanism in a distributed computing system. In *ICDCS* (2000).

[70] YAN, S., WANG, X., ZHENG, X., XIA, Y., LIU, D., AND DENG, W. ACC: automatic ECN tuning for high-speed datacenter networks. In *SIGCOMM* (2021).

[71] ZHANG, M., ZHANG, J., WANG, R., GOVINDAN, R., MOGUL, J. C., AND VAHDAT, A. Gemini: Practical Reconfigurable Datacenter Networks with Topology and Traffic Engineering. *arXiv cs.NI 2110.08374* (2021).

[72] ZHANG, Q., LIU, V., ZENG, H., AND KRISHNAMURTHY, A. High-resolution measurement of data center microbursts. In *IMC* (2017).

[73] ZHOU, Y., ZHANG, Y., YU, M., WANG, G., CAO, D., SUNG, E., AND WONG, S. Evolvable Network Telemetry at Facebook. In *NSDI* (2022).

A Rescaled-range analysis for estimating H

For a weak stationary stochastic process $X = (X_t : t = 0,1,2,...,N)$, the mean-adjusted series is defined as Y, $Y_t = X_t - m$ where m is the empirical mean of the process X. Let Z denote the cumulative deviate of Y where $Z_t = \sum_{t=1}^{N} Y_t$. We also define m_t as the cumulative mean of the series X through time t.

The rescaled range of X is denoted by

$$(R/S)_t = \frac{R_t}{S_t}, t \in \{0,1,2,...,N\} \quad (3)$$

where the Range series R is defined as

$$R_t = Max(Z_1, Z_2, ..., Z_N) - Min(Z_1, Z_2, ..., Z_N), \quad t \in \{0,1,2,...,N\} \quad (4)$$

and the standard deviation series S is defined as

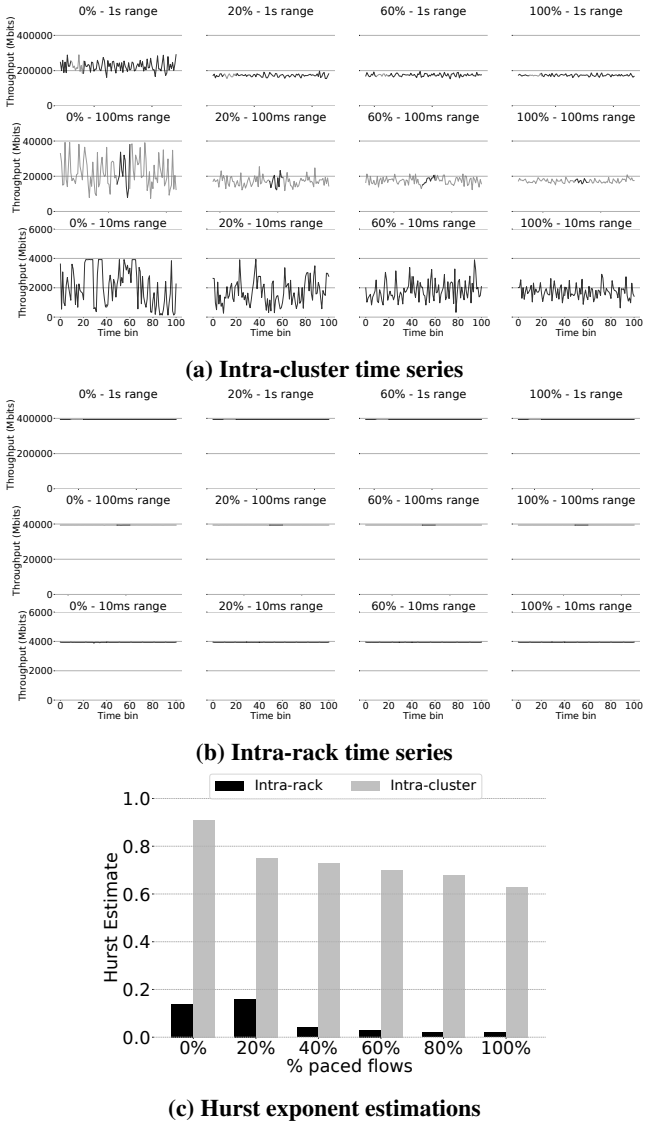
$$S_t = \sqrt{\frac{1}{t}\sum_{i=1}^{t}(X_i - m_t)^2}, t \in \{0,1,2,...,N\} \quad (5)$$

According to [30], R/S scales with the power law of t. Therefore, to estimate H, the slope of the least-squares linear regression of R/S over t in a log-log scale is used. The resulting exponent is in the 0-1 range and a value between 0.5 to 1 indicates low to strong long-range dependence (self-similarity), respectively. In other words, an H estimate close to one indicates a strong desire to maintain the previous trend or more burstiness. As the H estimate nears 0.5, the time series becomes indistinguishable from random noise, and a value close to zero signifies the traffic's aim at reverting to its mean value.

B Theoretical analysis of software pacing under different workloads

We presented the size of per-flow bursts for explicit software pacing in §5.3.3. To further verify our findings using the notion of self-similarity, we first plot the time-series of packet arrivals in 1s, 100ms, and 10ms time scales in Figure 14 for both the intra-cluster (Figure 14a) and intra-rack (Figure 14b) workloads. One can notice the gradual decay of burstiness in all time scales as higher degrees of pacing are enforced to the intra-cluster workload. On the other hand, we can observe that the intra-rack traffic follows a non-bursty, steady trend in all time scales regardless of pacing.

Next, we calculate the Hurst exponents for the intra-rack and intra-cluster workloads. According to Figure 14c, self-similarity in the latter workload follows the degree of pacing (i.e., percentage of the paced flows) where 100% pacing results in 31% reduction in the Hurst estimate compared to the no-pacing case. For example, the Hurst estimates are 0.91, 0.73, and 0.63 for 0%, 40%, and 100% pacing ratios, respectively. However, under the intra-rack workload pacing seems



(a) Intra-cluster time series

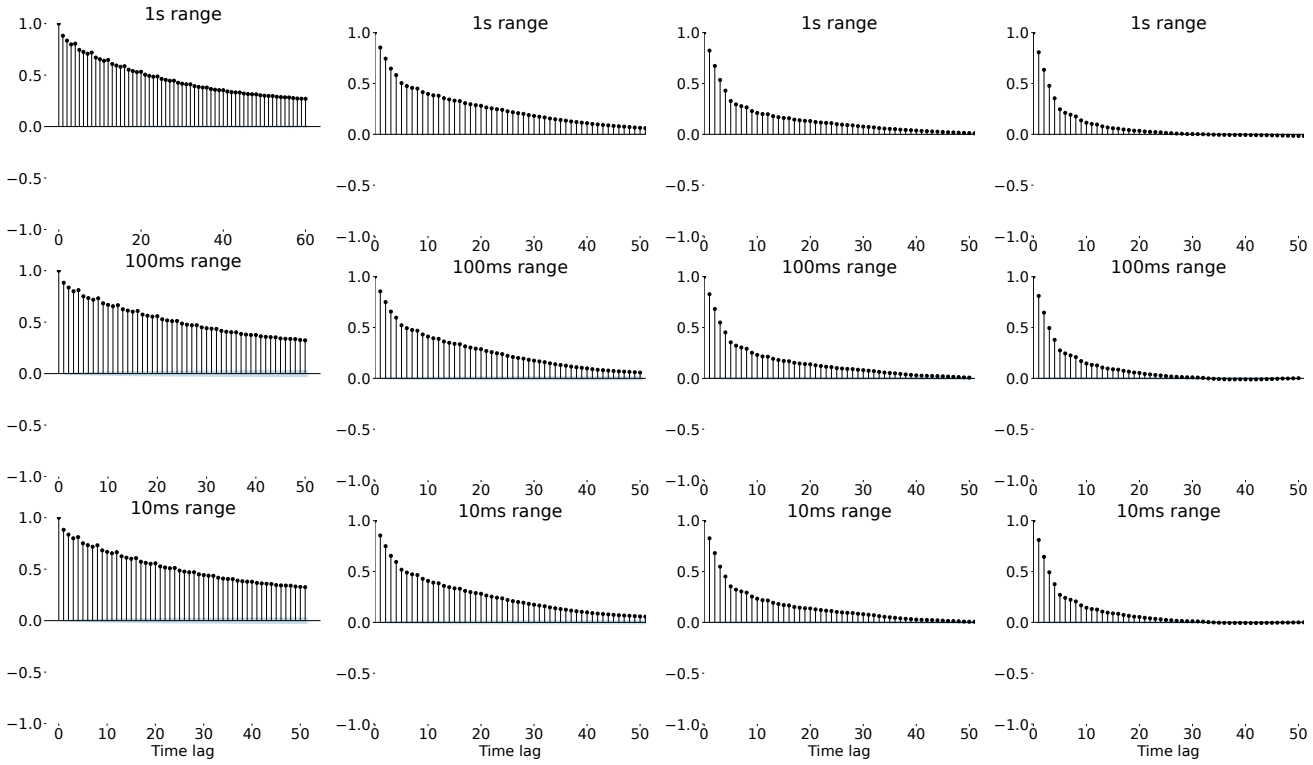
(b) Intra-rack time series

(c) Hurst exponent estimations

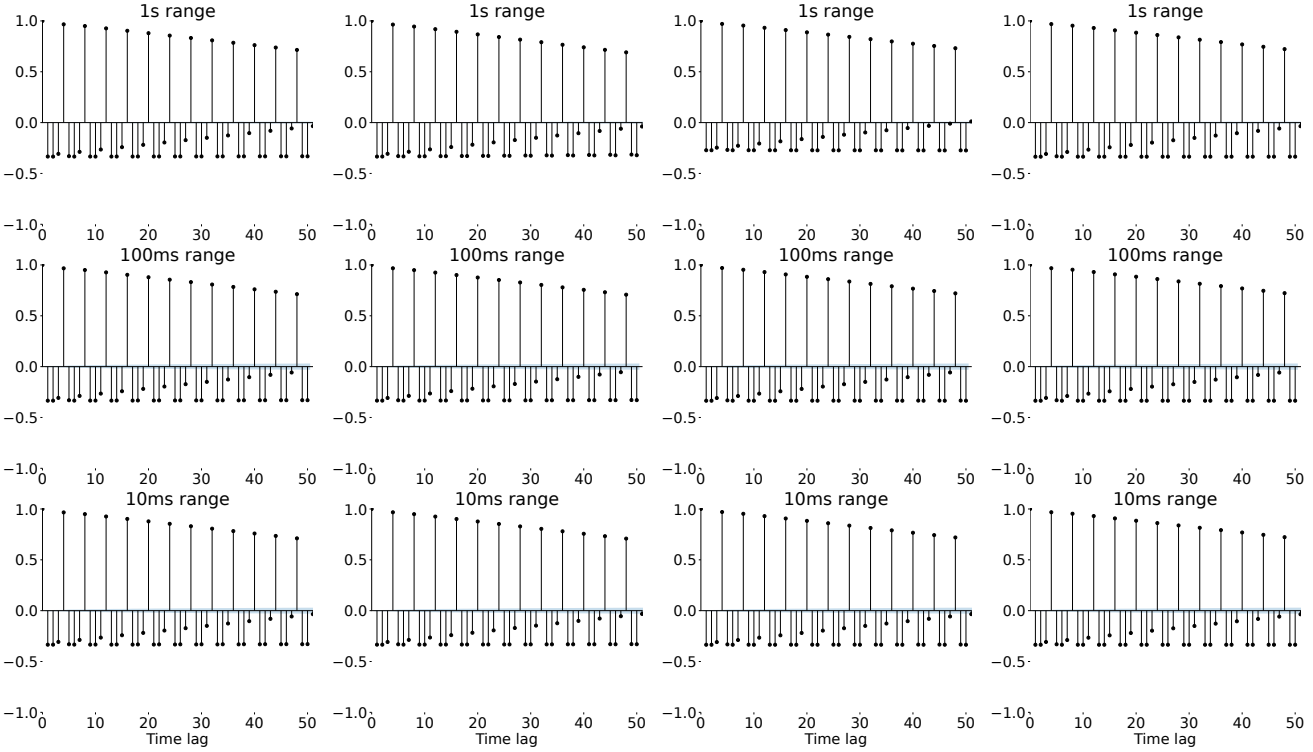
Figure 14: Time-series graphs and Hurst exponents for the software pacing experiments presenting the traffic behavior at three time ranges.

to have little to no effect as the egress traffic follows a mean-reverting behavior during 1-second time ranges (H < 0.20 for all the cases).

Finally, Figure 15 presents the corresponding auto-correlation functions (ACFs) for the above time series. While the cycling trend of bars between positive and negative correlations suggests a strong mean-reverting behavior for the intra-rack workload (Figures 15e-15h), the intra-cluster ACF features a slow-decaying, strong positive correlations across time lags, suggesting strong self-similarity (Figures 15a-15d). As we increase the pacing ratio (from 0% gradually to 100%), the correlations start to decline.



(a) Intra-cluster 0% paced (b) Intra-cluster 20% paced (c) Intra-cluster 60% paced (d) Intra-cluster 100% paced



(e) Intra-rack 0% paced (f) Intra-rack 20% paced (g) Intra-rack 60% paced (h) Intra-rack 100% paced

Figure 15: Comparing the auto-correlation functions (ACFs) for two workloads when we increase the ratio of paced flows from 0% to 100%. For the intra-cluster workload, enforcing pacing on flows can significantly reduce the self-similarities. For the intra-rack workload, the correlations between consecutive time lags oscillate between positive and negative numbers, signifying the mean-reverting nature of the workload irrespective of the pacing.

Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT

Weitao Wang[*] [†], Masoud Moshref[*], Yuliang Li[*], Gautam Kumar[*],
T. S. Eugene Ng[†], Neal Cardwell[*], and Nandita Dukkipati[*]

[*]Google LLC, [†]Rice University

Abstract

The difficulty in gaining visibility into the fine-timescale hop-level congestion state of networks has been a key challenge faced by congestion control (CC) protocols for decades. However, the emergence of commodity switches supporting in-network telemetry (INT) enables more advanced CC. In this paper, we present *Poseidon*, a novel CC protocol that exploits INT to address blind spots of CC algorithms and realize several fundamentally advantageous properties. First, Poseidon is *efficient*: it achieves low queuing delay, high throughput, and fast convergence. Furthermore, Poseidon decouples bandwidth fairness from the traditional AIMD control law, using a novel adaptive update scheme that converges quickly and smooths out oscillations. Second, Poseidon is *robust*: it realizes CC for the actual *bottleneck hop*, and achieves max-min fairness across traffic patterns, including multi-hop and reverse-path congestion. Third, Poseidon is *practical*: it is amenable to incremental brownfield deployment in networks that mix INT and non-INT switches. We show, via testbed and simulation experiments, that Poseidon provides significant improvements over the state-of-the-art Swift CC algorithm across key metrics – RTT, throughput, fairness, and convergence – resulting in end-to-end application performance gains. Evaluated across several scenarios, Poseidon lowers fabric RTT by up to 50%, reduces time to converge up to 12×, and decreases throughput variation across flows by up to 70%. Collectively, these improvements reduce message transfer time by more than 61% on average and 14.5× at 99.9p.

1 Introduction

Effective datacenter congestion control (CC) needs to provide high throughput, low latency, fairness, and fast convergence across varied workloads. CC is becoming more and more critical as applications increasingly demand low-latency operations at datacenter scale. Examples of such applications include memory and storage disaggregation [9, 21, 25, 31], which require latencies as low as O(10μs) at 1M+ IOPs per server [14], and ML applications that require high network utilization to keep expensive accelerators busy [37, 45]. Large scale incasts with O(5000) flows [35] caused by shuffle operations [1] and partition-aggregate workflows continue to be prevalent and need CC to be fair across flows in order to avoid starvation and control the tail latency, which is critical for the performance of such applications [20]. Simultaneously, CC is becoming more challenging because link bandwidths are growing faster than buffers at switches [5], and high-packet-rate servers [3, 24] benefit from simple CC algorithms offloaded to NICs to save CPU for applications.

Datacenter CC algorithms in deployment today rely on either end-to-end signals (e.g., delay [35]) or quantized in-network feedback (e.g., ECN [8]), owing to their simplicity. An underlying problem with these signals is that they are aggregated *end-to-end* across all hops on a flow's path. Thus, these CC algorithms react to collective congestion along the path (for delay) or congestion at *any* hop on the path at different times (for ECN), leading to reducing a flow's rate before reaching its fair share in the network. This leads to underutilization, slow ramp-up, and/or unfairness in multiple scenarios shown in §2.1 and §5.

However, with the emergence of commodity switches that support in-network telemetry (INT), a new opportunity has emerged. INT-enabled switches can modify or append to packet headers to convey information local to the switch, such as the time the packet spent in the queue. Some state-of-the-art CC algorithms [7, 40], use INT to gather telemetry information for every hop to gain more visibility into the network and control the outstanding packets at each hop. Still, such solutions react to congestion at *any* hop, which leads to the unfairness and ramp-up problems mentioned above.

In the last few decades, several schemes have been introduced that leverage help from network switches for better CC [13, 22, 27, 34, 40] but almost none have been deployed widely in datacenters. Based on the successful deployment of ECN-based solutions [8] and no deployments of XCP [34], RCP [22], and similar AQM solutions, we believe a *deployable* CC scheme using INT should also have the following properties: 1) works seamlessly in heterogeneous brownfield deployments where new switches and old switches co-exist and provides benefit even if a subset of switches support INT. 2) uses a simple, low-overhead, non-intrusive INT scheme that requires minimal coordination among applications, networking stacks, NICs, and switches.

Therefore, in this paper, we ask the question: *How can we harness the power of INT to design a datacenter CC algorithm that is* ***efficient*** *(high throughput, low latency, and fast convergence),* ***robust*** *(max-min fairness across traffic patterns including multi-hop and reverse-path congestion), and* ***practical*** *(simple and deployable)?*

We find that learning the congestion state of every hop of a flow is unnecessary. Instead, an efficient and practical

CC can be realized based on the congestion state of *only the bottleneck hop* of a flow – the hop that limits the rate of the flow as per the max-min fair allocation. It's worth noting that a *congested hop* is not the *bottleneck hop* for all flows passing through it, but only for flows that send more than their max-min fair-share rate, and thus, CC should ideally decrease the rate of only those flows that send above their fair-share.

Armed with this key insight, we develop a novel INT-based CC protocol called *Poseidon*. Poseidon grounds itself in Swift [35], the state-of-the-art CC that's deployed in production at scale for TCP [17] and kernel-bypass stacks [41]. But Poseidon advances beyond Swift by leveraging INT instead of purely E2E measurements, and formalizes an adaptive congestion window update function that compares the *max* per-hop delay (obtained via INT) against a *rate-adjusted target* bottleneck hop delay.

This paper makes contributions in two main areas:

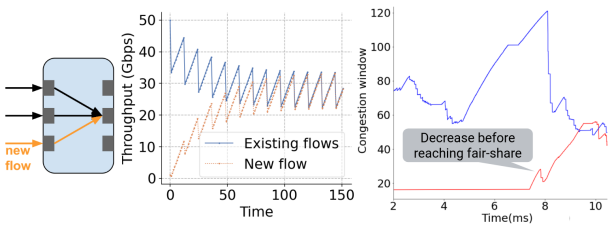
First, Poseidon utilizes the power of INT to provide unique properties, like network-wide max-min fairness, monotonic fast convergence, and stable rate under high concurrency.

- By comparing the max per-hop delay against a dynamic target, Poseidon converges to the network-wide max-min fair allocation, where flows only react to congestion on their bottleneck hop. A corollary to this is that flows in Poseidon do not decelerate before reaching their fair-share rate, resulting in fast convergence.
- Poseidon provides a characterization for the spectrum of *cwnd update* and *target max per-hop delay* functions that guarantee both fairness and high utilization. This allows us to explicitly decouple the fairness objective from the rate increase-decrease function (e.g., AIMD); Poseidon leverages this to use an *adaptive* increase-decrease function (without an AI component) that accelerates arriving at the fair-share allocation and smooths oscillations around it in the presence of many flows. Poseidon uses a novel target function, which achieves low queuing delay and high utilization for both sparse workloads (a few fast flows) and high-concurrency workloads (many slow flows).
- Poseidon is amenable to incremental deployment, including seamless coexistence in brownfield scenarios.

Second, Poseidon provides a simple, practical, and deployable design for enabling INT in datacenters for CC.

- We detail an efficient INT mechanism where switches signal the maximum per-hop queuing delay on a packet's path, using only a small and fixed amount of packet header space, at line rate.
- We analyze requirements for deployable INT for CC and compare proposed formats against those requirements.

We implemented Poseidon in a production networking stack (similar to Pony Express [41]) and a testbed with commodity programmable switches, with no changes to the NIC or applications. Our testbed evaluation shows that Poseidon is robust to reverse-path and multi-hop congestion scenarios explained in §2.1. In addition, we have evaluated Poseidon extensively in packet-level simulations (§5) and show that, compared to Swift and HPCC [40], it is robust to the above scenarios. Relative to Swift, Poseidon improves application-level message transfer latency by 61% at median and 14.5× at 99.9p. This is achieved by lowering fabric RTT by more than 50%, reducing congestion window ramp-up time up to 12×, and decreasing throughput variation for flows with small windows by up to 70%. In brownfield, Poseidon achieved at least 50% of the op latency gain of full deployment.



(a) Topology (b) AIMD algorithms (c) Swift experiment

Figure 1: MD on ramping-up flows delays convergence.

2 Motivation

In this section, we first show how congestion control (CC) algorithms are inefficient if they cannot distinguish the bottleneck hop of a flow from a merely congested hop. Then, we motivate the importance of brownfield deployment to support incremental roll-out and highlight why the format of INT is important for deployment.

2.1 CC Challenges in Datacenters

We use several scenarios in datacenter networks to highlight how two classes of issues – reacting to signals from hops other than the bottleneck hop, and increasing with a fixed value – cause unfairness, low link utilization, and slow ramp-up.

2.1.1 Decelerating Before Reaching Fair-share

Traditionally, when a hop is congested, a flow with a lower rate (e.g., a new flow) does not increase its rate monotonically to the fair share; instead, with every congestion signal, its rate decreases. Figure 1(a) draws an example where a new flow competes with two existing flows, Figure 1(b) shows the typical behavior for AIMD algorithms, and Figure 1(c) shows the data from that experiment in production using Swift. This behavior prolongs the time for the lower-rate flow to ramp up and leads to a longer tail flow completion time. The root cause is that in current CC algorithms, all flows must react *the same way* to the congested hop (either increase/decrease) regardless of their rate. This mechanism is designed to achieve fairness and stability given an end-to-end signal (e.g., delay, loss, ECN) without coordination across flows [19]. Poseidon leverages INT to get a richer signal and allows flows to increase their rates monotonically until reaching the fair-share rate.

2.1.2 Multi-hop Congestion

Datacenter networks are usually oversubscribed at ToR and Spine layers [46], thus it is common for a flow to see multiple

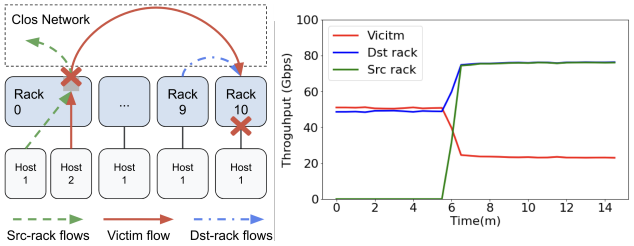


Figure 2: A Swift flow facing congestion at multiple hops (red) cannot compete at congested hops

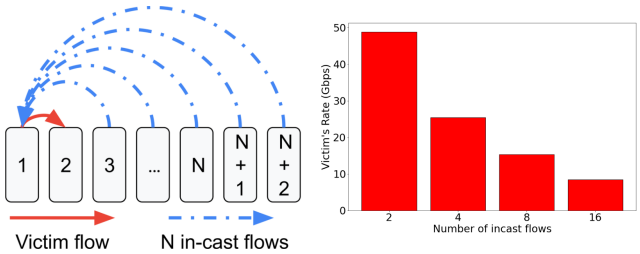


Figure 3: Flows react to reverse-path congestion.

congested hops in its path, especially in an incast. However, when a flow faces congestion in more than one hop, it gets lower throughput than other competing flows that traverse a single congested hop. The reason is that flows reacting to loss and ECN [8, 28] from multiple hops see more losses or marked packets on average, as the drops or markings happen asynchronously across different congested hops. In Swift, the fabric delay of such flows is higher, since every congested hop introduces more delay to the sum. HPCC [40], even though it uses INT, will also react to congestion at *any* hop with high in-flight bytes even if the flow is not contributing much to it.

Figure 2 shows this scenario in an experiment in production settings using Swift. The red flow (victim), that competes with the blue flow at the destination, gets much lower throughput once the green flow starts at the source ToR. The root cause is that the victim flow reacts to the congestion on Rack 0 uplink or Rack 10 downlink even when it didn't get the fair-share. This is because Swift looks at the end-to-end fabric delay and the victim's fabric delay includes both the delay at Rack 0 and Rack 10.[1] We observe the same problem even if the flow reacts to the max hop delay [8, 40] as shown in §5.6. Ideally, the victim flow should always only react to the congestion at the hop where it got more than the fair-share.

2.1.3 Reverse-path Congestion

In Figure 3, as we increase the number of flows on the reverse-path (blue), the forward traffic (red) gets lower throughput and cannot utilize the bandwidth. The root cause is that *the end-to-end delay* used in Swift includes the delay of ACKs in the reverse-path. Thus, Swift decreases the congestion window as if it is competing for the forward *and* reverse path bandwidth. This issue can happen because of congestion on any hop in the reverse-path, and can also cause unfairness if only a subset of flows on a bottleneck see reverse-path congestion, but is special for CC algorithms that use the end-to-end delay. A solution is to use synchronized timestamps at hosts (at μs level) in order to break fabric delay into forward and backward delays [39], but we show that CC can use INT to separate congestion signals of forward and reverse path and avoid the overhead of maintaining a synchronized clock.

Summary of the above three scenarios: many existing CC algorithms – when using loss, ECN, delay, or INT signals – react to every congested hop along the path, rather than only the congestion on the bottleneck hop. To put it another way, all flows going through a congested hop react the same way, either increase or decrease their rate, regardless of whether they have achieved their fair share or not. In §3.1, we show how Poseidon solves this problem by reacting to congested hops only for flows that reached their fair share.

2.1.4 Slow Convergence and Throughput Oscillation

An efficient CC algorithm should converge quickly to the right rate when the flow's rate is far from it and stay near it in a stable fashion. However, because many existing CC algorithms [8, 32, 35, 40] do not know the fair-share rate or how far they are from that rate, they rely on an AIMD, a well-understood algorithm that converges to fairness.

However, AIMD causes slow convergence for large windows and an unstable rate for small windows because AIMD increases the congestion window (*cwnd*) by a *fixed* amount every RTT. On the one hand, as *cwnd* becomes larger, the increase ratio compared to the window size becomes smaller: An increase of 1, takes 5 RTTs to double a window of 5, but 50 RTTs to double a window of 50. Slow *cwnd* growth can be particularly detrimental in workloads that desire high throughput from a few flows per host (e.g., ring topology in ML applications). On the other hand, as we increase the number of flows and get smaller *cwnd*, the effect of the increase amplifies for windows close to the additive factor. (Each one of 500 flows with a window of 1 may double its rate.) This causes oscillating *cwnd* in high-degree incast applications (e.g., shuffle [1]). A CC algorithm may use a combination of a multiplicative factor and additive factor [7, 40] for faster ramp-up, but still, the disproportionate effect of the additive increase component will manifest for a small *cwnd*.

The root cause is that AIMD was designed to provide fairness regardless of the quality of the signal (e.g., a binary loss signal in TCP Reno). Yet, it is used in many modern datacenter CC [8, 35], including the ones based on INT [40]. If we knew the fair-share from switches, we could converge faster [22, 34], but such solutions are hard to deploy. Instead, in Poseidon, flows can estimate if they are close or far from the fair-share and adjust the step size accordingly to converge faster and have a more stable throughput around the fair-share rate (§3.3), similar to some previous CC algorithms designed to facilitate large WAN BDPs [18, 28].

[1] Although the victim flow always faces a higher delay than the other two flows, its throughput didn't reach 0. The reason is that flow-scaling, designed for windows<10 [35], rises victim's target delay.

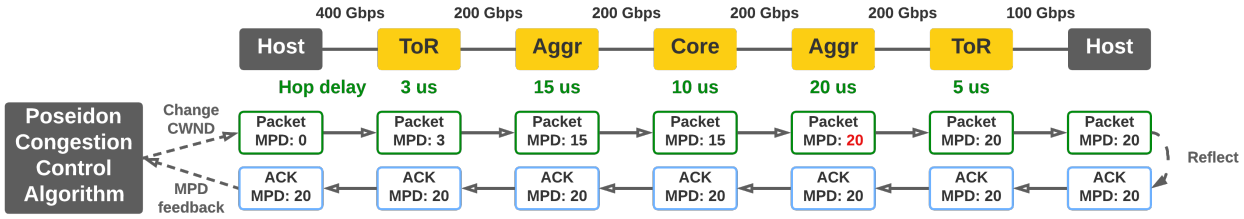


Figure 4: An example of Poseidon MPD signal propagation.

2.2 Deployment

Brownfield deployment. An important requirement for deploying INT in production is to support brownfield deployments. Hardware may be replaced gradually, from the ToR level to higher levels, or from one pod to another [26]. This transition phase can last for years [47]. INT may not be enabled on some switches, and at any point, we may want to roll back to disable INT without coordinating hosts and switches. Therefore, even if we use a separate queue for the new traffic [35], we still have to address the following requirements:

1. Being able to route the traffic regardless of whether a switch has INT enabled or not: While two hosts can coordinate their capabilities during a connection's initial handshake, we don't want any coordination between hosts and switches or switches with each other in order to forward packets and use ECMP. This places a tight requirement on the format of INT packets, as discussed in §4.2.
2. Getting some gains on an incremental INT deployment: Even though only a subset of hops supports INT, the CC algorithm should still benefit from that partial information.
3. Fair interaction between flows that have INT support on every hop and those that have it only on a subset of hops.

In §4.1, we explain why adjusting the target helps deploy Poseidon in brownfield. We also present our solution to combine end-to-end delay and max-hop delay to keep fairness while providing some incremental benefit in brownfield.

Low-overhead non-intrusive INT. For easy deployment, we prefer coordinating the least number of components and sustaining minimum overhead. Above, we mentioned that the traffic must go through the brownfield without any coordination between hosts and switches. At the end-host, we also want minimum coordination between applications, networking stack, and NIC. For example, a fixed INT length is preferred as it doesn't change MTU.

We want INT on all packets, so its overhead regarding bandwidth and packet processing in the hosts, NIC, and switches is important. Small INT length is preferred for low bandwidth overhead and easy deployment in offloaded NICs [10, 11]. Finally, INT information cannot be encrypted, require complex functions, or rely on the per-flow state in the switch.

There are multiple formats for supporting INT, two of which are IFA [36] and P4-INT [2]. These formats differ in multiple aspects. Instead of proposing yet another format, we describe the features required for an INT format to be deployable in a production datacenter for CC. §4.2 covers these requirements and how the formats satisfy them.

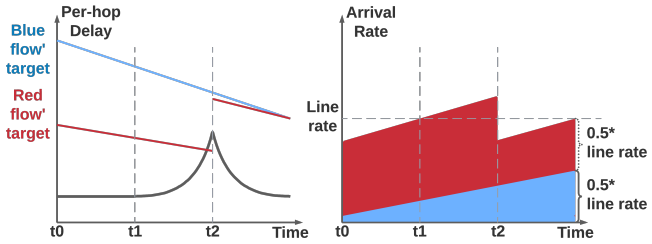


Figure 5: Delay is bounded by the faster flow's target.

3 Design

Poseidon achieves high link utilization, low queuing delay, network-wide max-min fairness, with fast convergence and stable per-flow throughput. In this section, we describe the design of Poseidon: First, we introduce a key idea that allows Poseidon to only react to the bottleneck hop (§3.1). Next, we demonstrate how Poseidon guarantees fairness on a single hop (§3.2) and how decoupling the fairness from the *fixed* increase in AIMD allows us to introduce an adaptive increase/decrease algorithm that achieves faster convergence and more stable throughput than AIMD (§3.3). Finally, we show that Poseidon achieves network-wide max-min fairness (§3.4).

3.1 Key-idea: Only React to Bottleneck Hop

Poseidon only reacts to the bottleneck hop by decreasing the congestion window only if the flow got the fair-share on congested hops over its path. We explain how to do that without knowing the fair-share. Poseidon compares a delay signal with a target delay to increase or decrease the window. The key idea is in the definition of the delay signal and target:

1. It applies the target to the **maximum per-hop delay (MPD)** to allow flows to react to the most congested hop.
2. It adjusts the **target** based on the **throughput** of the flow to make sure only the flows that get the highest rate on the hop reduce their windows.

Figure 4 illustrates an example of how max per-hop delay is propagated. Each packet carries the MPD and each hop updates it. The ACK packet will reflect MPD back to the source. Note that Poseidon could naturally support heterogeneous link bandwidth in the network.

Now we describe each point in more detail. Every flow tries to maintain the **maximum per-hop delay (MPD)** close to a **maximum per-hop delay target (MPT)**, namely, increasing the congestion window when $MPD \leq MPT$ to keep the link busy and decreasing the window when $MPD > MPT$ to limit the congestion. MPD adds small and fixed overhead to packets and is one of the important designs to find the bottleneck hop:

In the max-min fair state, the hop with maximum latency is the bottleneck hop of the flow for Poseidon; otherwise, the flow has not reached its fair-share along its path (§3.4). The former case must decrease the congestion window, and the latter must ignore the congestion and increase the window. We achieve that by adjusting the target.

Poseidon calculates MPT for each flow based on its rate: **the larger the rate is, the smaller MPT will be** (§3.3 defines the function). This means that flows with higher rates have lower targets, thus decreasing their window earlier and more aggressively[2]. This became possible using INT, as now all flows competing in the same queue tend to observe the same congestion signal (per-hop delay). Figure 5 shows an example: As the arrival rate on the link goes over the line rate at time $t1$, a queue builds up. The hop delay grows over the target of faster flow, red, and forces it to reduce its window at $t2$. However, the slower flow, blue, can still increase its window (solves §2.1.1). Interestingly, this means that given the same congestion signal from the network, some flows increase and some decrease their rate. In the next section, we demonstrate how Poseidon achieves fairness given this flexibility without relying on an additive increase.

Algorithms 1 shows how Poseidon updates the congestion window (*cwnd*). The pacing only happens when the *cwnd* is less than 1, similar to Swift [35]. Note that the multiplicative-increase (MI) happens per packet, thus Line 5 in Algorithm 1 has to approximate the ratio for each packet, while *cwnd* decreases happen only once per RTT, thus it is a simple multiplication. The retransmit and recovery functions are included in Appendix A.

3.2 Single-hop Fairness

We show that with the right increase/decrease functions, Poseidon can achieve fairness on a single hop. The AIMD algorithm benefits from the fact that *all* flows either increase rate with the same amount or decrease rate with the same ratio [19]. However, because of Poseidon's rate-adjusted target delay and delay-based increase/decrease function, Poseidon has a new case, where *faster flows decrease rate* while *slower flows increase rate*. This happens if the queuing delay is higher than the faster flow's target, but lower than the slower flow's target.

To prove that Poseidon can achieve fairness, we show that fairness improves in all possible cases:

1. MPD is low, and all flows increase rate.
2. MPD is high, and all flows decrease rate.
3. MPD is high, some faster flows decrease, other slower flows increase their rate.

Assume a queue with two flows A and B with rates a and b where $b > a$. As a result, the target of A is larger than the target of B ($T(a) > T(b)$). In Figure 6, the fairness is graphically defined as the angle between the actual bandwidth share and fair-share line. We define the update function $U(T(rate), delay)$ as the multiplicative factor (where $new_cwnd = cwnd \times U()$) with a specific flow rate and network delay. In order to converge to the line rate, it is ≥ 1 if the delay is less than or equal to the target and < 1 if the delay is more than the target[3].

$$U(T(rate), delay) = \begin{cases} \geq 1, delay \leq T(rate) \\ < 1, delay > T(rate) \end{cases} \quad (1)$$

In all three cases, if we want to guarantee that the fairness improves, the updated rates should stay in the red triangle

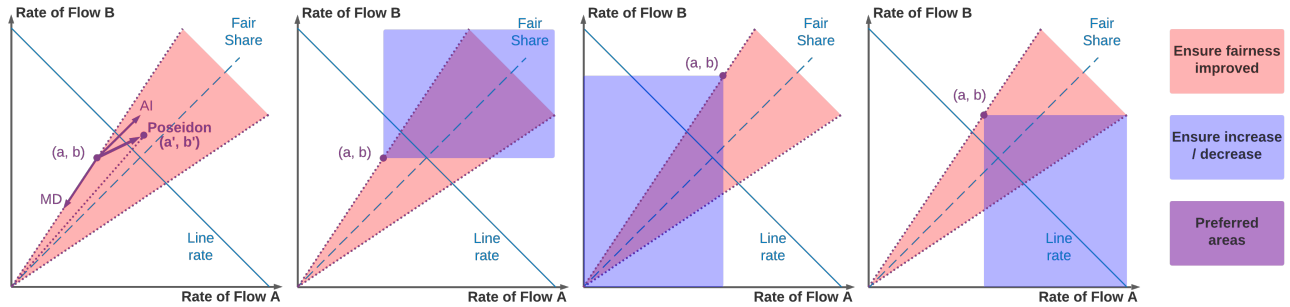
Algorithm 1: Poseidon's Main Algorithm

Input: *mpd*: maximum per-hop delay,
cwnd: flow's congestion window size,
rtt: round-trip time,
now: current timestamp
Parameter : *min_md*: minimum MD ratio,
max_mi: maximum MI ratio,
min_cwnd: minimum cwnd,
max_cwnd: maximum cwnd

1 **Function** `ReceiveACK()`**:**
2 $mpt \leftarrow T(\frac{cwnd}{rtt})$
3 $update_ratio \leftarrow U(mpt, mpd)$
4 **if** $mpd \leq mpt$ **then**
5 $cwnd \leftarrow cwnd * \left(1 + \frac{update_ratio - 1}{cwnd} * num_acked\right)$
6 **else**
7 **if** $now - t_last_decrease > rtt$ **then**
8 $cwnd \leftarrow cwnd * update_ratio$
9 **return** *cwnd*
10 **Function** `Poseidon()`**:**
11 $cwnd_prev \leftarrow cwnd$
12 **if** *is_ack* **then**
13 $cwnd \leftarrow$ `ReceiveACK()`
14 **else if** *is_retransmit* **then**
15 $cwnd \leftarrow$ `RetransmitTimeout()`
16 **else if** *is_fast_recovery* **then**
17 $cwnd \leftarrow$ `FastRecovery()`
18 $cwnd \leftarrow clamp(cwnd, min_cwnd, max_cwnd)$
19 **if** $cwnd < cwnd_prev$ **then**
20 $t_last_decrease \leftarrow now$
21 $pacing_delay \leftarrow 0$
22 **if** $cwnd < 1$ **then**
23 $pacing_delay \leftarrow \frac{rtt}{cwnd}$
24 **return** $cwnd, pacing_delay$

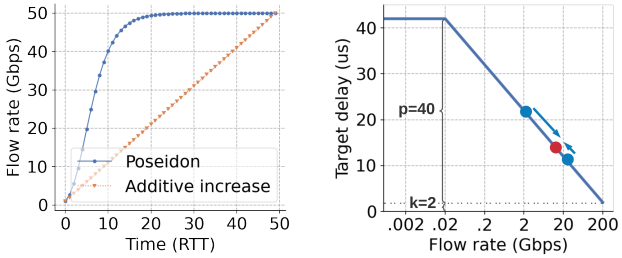
[2] In rare cases, the queuing delay of a port may jump over the target of both fast and slow flows because of synchronized packet arrival. We make sure that faster flows with smaller targets decrease more aggressively (§3.3)

[3] We assumed, in average, if arrival rate < line rate, delay is low, and if arrival rate > line rate, delay increases.



(a) Ensure fairness improved (b) All flows increase (c) All flows decrease (d) Only faster flows decrease

Figure 6: Poseidon updates the rate (in the purple area) such that it increases fairness (red) toward the line rate (blue). b) the queue is under-utilized and both flows increase rates; c) the queue is overloaded, and both flows decrease rates; d) the faster flow decreases, and the slower flow increases its rate.



(a) AI takes 50 RTTs from 1G to 50G, Poseidon takes around 15 RTTs and is stable at the fair-share rate.

(b) Poseidon target function has high resolution over all spectrum of rate (min-rate=0.02G, max-rate=200G).

Figure 7: The ramp-up using adaptive step sizes is fast and slows down near the target for stability.

(Figure 6(a)). One side of the triangle is defined by the current ratio of rates, and the other side is symmetric across the fair-share line. If we assume $a < b$ and the delay is D, this requirement can be written as:

$$\frac{a}{b} < \frac{b \cdot U(T(b),D)}{a \cdot U(T(a),D)} < \frac{b}{a}, \forall a < b, \forall D > 0$$
$$\frac{a^2}{b^2} < \frac{U(T(b),D)}{U(T(a),D)} < 1, \forall a < b, \forall D > 0 \quad (2)$$

In summary, Poseidon achieves high link utilization and fairness if the functions $T()$ and $U()$ satisfy Eq. 1 and Eq. 2. Figure 6 illustrates Eq. 1, updates that allow full link utilization, in blue color, and Eq. 2, updates that converge toward fairness, in red. The desirable overlapped area is marked in purple. The additive increase will be in parallel to the fair-share line, and the multiplicative increase/decrease with the same ratio stays on the same edge of the red triangle where the node (a,b) is (Figure 6(a)). For case 1 in Figure 6(b), the red area ensures the fairness is improved, and the blue area ensures all flows increase their rate; for case 2 in Figure 6(c), the blue area ensures all flows decrease their rate; for case 3 in Figure 6(d), the blue area ensures the slower flow increases rate while the faster flow decreases rate. Next, we introduce a target function $T()$ and the update function $U()$ which satisfy the above requirements and have more desirable properties.

3.3 Adaptive Update Steps

Based on §3.2, Poseidon can use any function that satisfies Eq. 1 and Eq. 2. But we designed the following functions to leverage the *distance* between the target and max-hop delay to not only decide whether to increase or decrease, but also adjust *the update ratio adaptively* to reach a better trade-off between stability and fast convergence. Appendix B proves that they satisfy Eq. 1 and Eq. 2:

$$T(rate) = p \cdot \frac{ln(max_rate) - ln(rate)}{ln(max_rate) - ln(min_rate)} + k$$
$$min_rate < rate < max_rate, p > 0, k > 0 \quad (3)$$

$$U(T(rate), delay) = \exp\left[\frac{T(rate) - delay}{p} \cdot \alpha \cdot m\right]$$
$$\text{where } \alpha = ln(max_rate) - ln(min_rate) \quad (4)$$

$rate$ is $cwnd * MTU/RTT$. k defines the minimum target delay; p tunes the maximum target when the rate is equal to min_rate and decides how far-apart the target of two flows with close rate can be. In practice, the target cannot be lower than a limit without decreasing the throughput because synchronized arrivals can cause premature window decrease. The target cannot be very large too because a) it can cause packet drops in switches when the target delay exceeds the queue capacity; b) as long as we achieve high utilization, we prefer to back-pressure hosts to leverage other mechanisms such as load-balancing and admission control for isolation. We use min_range and max_range to not waste the target range for differentiating rates that only happen rarely [35]. m defines the "step" when updating the rate. The larger m is, the slower the rate of update will be (sensitivity analysis is in §5.6.2).

When $|T(rate) - delay| \to 0$, then $U(rate, delay) \to 1$. This means when the delay is far away from the target, flows increase/decrease more drastically for faster convergence, and when the delay approaches the target delay, the steps will be more gentle to achieve stable flow rates (solves §2.1.4). Figure 7(a) shows how the flow can quickly increase its rate to reach 50 Gbps using the adaptive solution. We explain the intuition behind the update function with an example. Assume

the rate of a flow is x, and the target delay is D. We define the target rate r, such that $T(r) = D$ thus $U(r,D) = 1$. We can rewrite the update function for the flow as follows (calculation is in Eq. 11):

$$U(x,D) = \frac{U(x,D)}{U(r,D)} = \left(\frac{r}{x}\right)^m \tag{5}$$

Thus, the update function is only related to the ratio of r and x; when x is far-away from r, the change will be larger. Poseidon updates the rate of flow from x to r in one RTT, because $x \cdot U(x,D) = r$ if $m = 1$, and for $m < 1$, it will take more RTTs because $x \cdot U(x,D)^m = r$. In this way, the parameter m controls how fast Poseidon converges to the fair-share rate.

A legitimate alternative for $T(rate)$ is $\frac{\alpha}{\sqrt{rate}} + \beta$ which is an extension of the Swift flow-scaling (Appendix §C). However, we designed Eq. 3 because it gives a meaningful difference between the target of flows over *all* rates: The target of a flow with rate a and $c \cdot a$ have a fixed difference $T(a) - T(c \cdot a) = ln(c)/p$, providing uniform resolution across all ranges of rates (Figure 7(b)). This generalizes Swift's use of $1/\sqrt{cwnd}$ for target flow scaling (§3.5 of [35]), which only provides high resolution for small windows. Similarly, an option for the update function is to use the ratio of target over delay, similar to Swift. Appendix B.3 shows why distance provides a better result in high concurrency scenarios.

3.4 Network-wide Max-min Fairness

The key designs of Poseidon to achieve network-wide max-min fairness are: 1) only react to the max-hop delay; 2) the target delay of a flow increases when the flow rate decreases. We will start from the definition of max-min fair and then show how the above two designs achieve max-min fairness.

Definition 1 (Max-min Fairness [16, 38])**.** A feasible allocation of rate $\vec{x}$ is "max-min fair" if and only if an increase of any rate within the domain of feasible allocations must be at the cost of a decrease of some already smaller rate. Formally, for any other feasible allocation $\vec{y}$, if $y_s > x_s$ (s is a flow), then there must exist another flow s' such that $x_{s'} \leq x_s$ and $y_{s'} < x_{s'}$.

For a certain network and workload, the max-min fair allocation is unique [38]. In the max-min fair allocation, for each flow, there is a unique queue (switch port), which restricts the rate for that flow. We denote this queue as a flow's **bottleneck**, and the flow's rate should be the fair-share rate of that queue. (As a special case, a flow's bottleneck can also be the source or destination host, if either of them restricts the rate of the flow.) Specifically, we can conclude the following Lemma from the above definition (proved in Appendix D):

Lemma 1. *When achieving network-wide max-min fairness, each flow will have the largest rate among all flows on its bottleneck hop and not on any other saturated hop.*

Formally, for the "max-min fair" allocation $\vec{x}$, for any flow s, denote the flows that traverse s' bottleneck as $\{b_1, b_2, ..., b_k\}$,

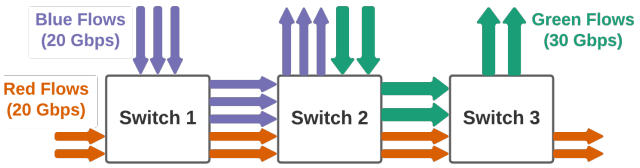


Figure 8: The stable state of max-min fairness among 3 switches with 100 Gbps links.

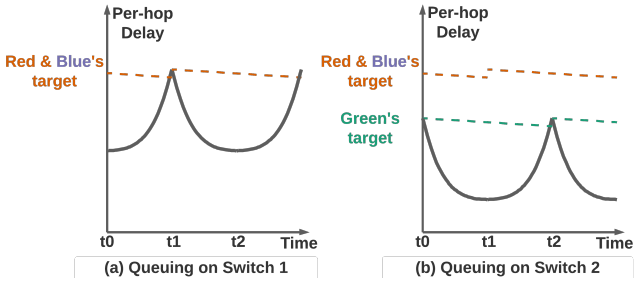


Figure 9: Only the queuing delay on red flows' bottleneck (switch 1) can reach red flows' target.

then for any flow b_i, $x_s \geq x_{b_i}$. Denote the flows that traverse one of the saturated non-bottleneck hops of s as $\{c_1, c_2, ..., c_k\}$, then there must exist some c_j such that $x_{c_j} > x_s$.

With the above definition and Lemma, we first give an *intuition* about why Poseidon could converge to the max-min fair state from any initial state.

With other CC algorithms, the hop with max queuing delay for a flow may not be the bottleneck hop based on the max-min fairness. Thus, using INT naively and reacting to the max delay cannot lead to max-min fairness. However, Poseidon uses a monotonically decreasing target function, which lets faster flows have lower target delay. With this design and Lemma 1, a flow should have the smallest target among all other flows on its bottleneck, and its target is never the smallest on other congested hops. Moreover, the delay on a queue will generally remain close to the minimum target among all flows on that queue. So gradually, the delay may reach a flow's target on its bottleneck; but on other congested non-bottleneck hops, the delay is not able to reach its target. Thus, **in Poseidon's final stable state, the max hop delay must come from flow's bottleneck**. And because each flow only reacts to its bottleneck, it achieves fairness on the bottleneck with other flows that have the same bottleneck (§3.2). Then, the network-wide max-min fairness is achieved by Poseidon.

We provide an example in Figure 8 where green flows have higher rates than red and blue flows in max-min fair state, $r_{green} > r_{red} = r_{blue}$, so green flows also have smaller targets, namely, $T(r_{green}) < T(r_{red}) = T(r_{blue})$. Switch 1 is the bottleneck of red and blue flows, and switch 2 is the bottleneck of green flows. On switch 2, the delay is similar to the target of green flows, $d_{sw2} \approx T(r_{green})$, because the moment the delay passes the target, green flows reduce their rate. Meanwhile, red and blue flows have higher targets than the delay d_{sw2}, as shown in Figure 9(b). This prevents red flows from reacting to the queuing on switch 2, which means every flow only

reacts to its bottleneck and maintains max-min fairness. This property of Poseidon solves the problem mentioned in §2.1.2.

Theorem 1. *Poseidon converges to the max-min fairness.*

To formally prove that the network converges to the max-min fair state, we use induction to prove that each queue achieves max-min fair. Denote the max-min fair rate allocation as $\vec{x}$, and for each queue, we denote the fastest flow's rate on that queue as R_q^x. Then we sort all the k queues in the network according to R_q^x from smallest to largest: $R_{q_1}^x \leq R_{q_2}^x \leq ... \leq R_{q_k}^x$. For any other flow rate allocation $\vec{y}$, with induction: **(1)** we prove that the queue q_1 will converge to the max-min fair allocation; **(2)** assuming the queue q_1 to queue q_m have already converged to the max-min fair allocation $\vec{x}$, we prove that the queue q_{m+1} will also converge to $\vec{x}$. A detailed proof is provided in Appendix §E.

4 Deployment

Here, we discuss the design decisions that facilitate the deployment of Poseidon in a large-scale datacenter network. Firstly, Poseidon provides benefits even if only part of the network supports INT (incremental deployment), and bounds the unfairness between flows that see INT vs those that do not. Secondly, Poseidon allows old switches to transparently route INT traffic, adds minimum overhead to packets and switches, and requires no changes in applications or NICs.

4.1 Brownfield Deployment

For a network where a subset of switches can provide hop delay information, Poseidon splits the fabric delay into two parts: the MPD from switches equipped with INT; and the delay from the rest of the path. This is calculated based on the end-to-end delay, using the NIC timestamp similar to Swift [35], minus the max-hop delay (both forward and backward). Then we apply Poseidon based on the maximum of the two. Note that this solution is not robust to reverse-path or multi-hop congestion happening in the hops that do not have INT, but still provides incremental benefits (§5.5).

The fairness issue is only relevant if the bottleneck hop of the two flows is the same. Consider two flows A and B and three switches, X, Y, and Z. A goes through switch X to Z, and B goes through Y to Z. The common switch, Z, is the bottleneck, X supports INT and Y doesn't. If Z has INT, both flows get the right feedback about Z in max-hop delay. Therefore, we get partial benefits. If Z doesn't have INT, the fabric delay of flow A doesn't include the delay of X, but for flow B it will include the delay of Y. Therefore, flow B observes a high delay and may decrease its window sooner. However, we argue that this decrease will be minimal and bounded because of target scaling. As the rate of flow B goes down, its target will go higher. The moment the target increases by the delay at hop Y, the rate of flow B will stabilize.

Interestingly, the above argument suggests that in order to get most of the benefit, we should prioritize deploying INT in the usual congestion points (ToRs with oversubscribed uplinks or incast in downlinks). We evaluate this in §5.5.

4.2 A Deployable INT Format for CC

In this section, we describe the requirements for deploying INT in datacenters for CC and compare existing INT formats.

4.2.1 Requirements

We consider both the INT metadata that we ask from each hop and how/where we put it inside the packet.

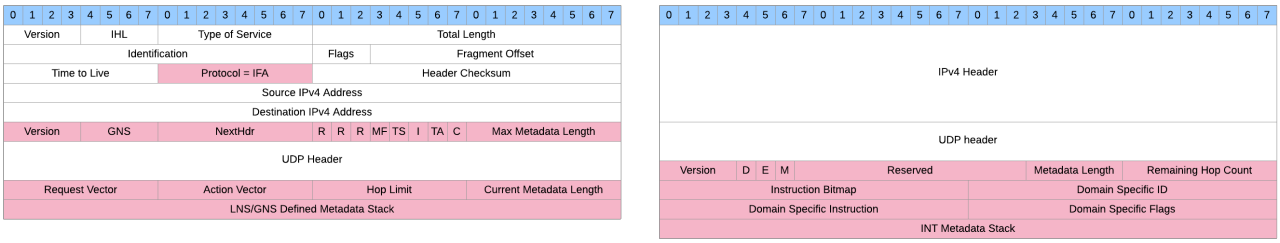
Make INT information available to the sender for CC: INT metadata on the forward path should be reflected in the reverse path ACKs for CC signaling. Ideally, ACKs could reflect opaque information that could be carried in the INT header but not be replaced by switches. Or, similar to ECN, INT could be marked by switches in the forward direction and echoed back to the sender in L4 headers.

Low-overhead INT metadata: For simplicity and precision, we want INT on all packets, thus its bandwidth and processing overhead must be low. Having many metadata fields per packet adds bandwidth overhead [15] and is costly for switches, NICs, and offloaded transports to process [11].

Fixed-sized INT metadata: Per-hop INT metadata makes the number of INT fields not only large but also variable. This is bad for two reasons: a) It is wasteful to reduce MTU for the worst case because link failures may add more hops to packets transiently, so that the number of hops is long-tailed. A smaller MTU means more packets to be processed by hosts and switches. b) Variable-sized INT metadata is more complex to parse at switches, offloaded transports, and middleboxes if they access bytes after the INT header. Therefore, for the CC use case, it is essential for an INT format and its implementation in the switch to use fixed-size INT metadata, and support *aggregation* functions (e.g., max/min/sum) that can overwrite the information from the previous hop.

Implementable in dataplane at line rate: The aggregation function must require minimal state and computation in the switch. This means that it must be simple (e.g., max/min/sum) and not require per-flow state.

Transparent to routing: Many datacenters use a hash-based scheme (ECMP [30], WCMP [48]) to balance the load over multiple ports. Such schemes may use the 5-tuple and/or IPv6 flow label. A brownfield deployment requires a scheme that balances load efficiently for packets with/without INT metadata in switches with/without INT support. For switches without INT support to balance INT traffic, they must be able to find and parse L4 headers. Thus, we either need to a) put INT metadata after L4, b) enable switches to pass over the INT metadata by adding it as a sub-header in headers that support extensions, such as VLAN-tag, MPLS-tag, IP option, GRE shim layer, or VXLAN shim layer. We believe option (a) is easier to deploy as it is transparent to the network, and thus works with different L2/L3 protocols, with virtualized and



(a) IFA packet format

(b) P4-INT packet format

Figure 10: INT packet format in a) In-band Flow Analyzer (IFA) b) P4-INT

non-virtualized traffic, and with other boxes (except special middleboxes, which are usually implemented in extensible software anyway [23, 43], and don't need to parse INT).

Compatible with encrypted packets: Many cloud providers encrypt network traffic inside datacenters [4]. But switches must be able to change INT data. Fortunately, recent NIC encryption modules such as PSP [6] allow passing an offset in the packet descriptor to only encrypt the bytes after. PSP also only authenticates the bytes after its header. For UDP checksum, PSP requires its implementation to support zero values (thus there's no need to rely on switches, even though programmable switches can update that). We also verified that we can change where the NIC expects the PSP header.

4.2.2 INT formats

Figure 10 shows two predominant INT formats in the context of IPv4; the IPv6 format is similar. Poseidon is possible on both formats, but a few improvements help its deployability.

IFA [36] indicates the presence of INT with a special protocol value in the IP header and adds part of the header between the IP and L4 header. P4-INT [2] indicates INT using a DSCP (traffic class) value/bit and puts all metadata after the L4 header. In order to use ECMP on switches without changing their configuration, we prefer to not change the location of L4. Still, IFA can be used on most switches by first changing the expected location of the UDP header for IFA packets (using User Defined Fields, UDF) and then rolling out INT. IFA also supports a format that puts INT metadata at the tail of the packet to avoid changing the location of L4.

Neither of the formats has a place to reflect the forward path INT. But given that the switches in the reverse path don't need to read the forward path INT, the receiver can just reflect the INT metadata in L4 headers, and the sender networking stack will consume it along with the INT metadata.

Neither of the formats defines a max calculation action, however, they allow extending the action vector.

Finally, we believe an overhead of 12B for sending a 2B metadata is excessive for small packets and look forward to working with the community to reduce overhead while maintaining protocol flexibility.

5 Evaluation

First, §5.1 explains our prototype implementation on a testbed with a production networking stack and NIC to highlight the ease of implementation and show the robustness of Poseidon to multi-hop and reverse-path congestion. Then we use simulations to explain how and why Poseidon is robust in those scenarios (§5.2). §5.3 shows that the adaptive window update enables faster convergence and better stability. Next, we present the aggregate benefit of the above techniques on op latency (flow completion) in multiple scenarios (§5.4). Finally, we wrap up with brownfield results (§5.5) and a parameter sensitivity analysis (§5.6). We use **Swift**, a practical CC deployed at scale, and **HPCC**, the state-of-the-art in INT-based CC, as our main points of comparison.

Simulation setup: We implemented Poseidon along with Swift and HPCC in the OMNeT++ packet simulator and simulated a Clos network of 200 Gbps links, with 245 ns link delay (including 230 ns FEC delay), 600 ns switch delay, 64 MB buffer size, and 4096 Bytes MTU size. For Poseidon, we set the parameters in Eq. 3 and Eq. 4 as $p = 40$, $k = 2$, $m = 0.25$ based on §5.6.2. For Swift, we follow the best parameters in [35] and set the base delay to 25 μs, the max flow scaling to 100 μs, and the hop-based scaling to 1 μs per hop. We verified the fidelity of the simulator by comparing it to the result of the testbed. Note that RTT here is calculated based on NIC timestamp and doesn't include the delay in the networking stack at the host. For HPCC parameters, we use the values from the paper [40]. To be fair in our comparisons, we enable pacing only when $cwnd < 1$, similar to Swift[4].

5.1 Implementation in Testbed

For the host networking stack, we change Swift implementation in a transport stack similar to Pony Express [41] to 1) at the sender, add a 2-bytes INT header for max-hop-delay right after L4; 2) at the receiver, reflect back the max-hop-delay in another 2-bytes in the payload; 3) at the sender, update the congestion window based on Poseidon algorithm in §3.3.

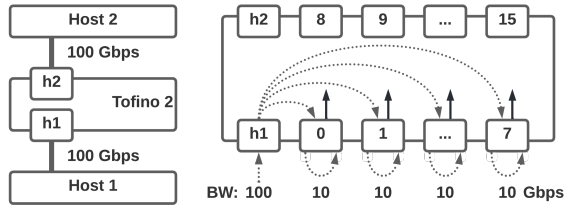
For network switches, we extract the queuing delay at the egress pipeline and update the max-hop-delay in the INT header. We implemented the P4 program with only 2 lines of P4 code (Listing 1) and 16 lines of parser/deparser code in a Tofino switch. Moreover, we verified that packets with or without INT headers can both be routed.

Our testbed only has two hosts and one switch (Figure 11(a)). To simulate congestion from multiple hosts, we create 8 virtual interfaces in hosts with 100G links and route the

[4] HPCC always paces packets, but that is costly in software and hardware.

```
bit<16> queuing_delay =
    (bit<16>) (eg_intr_md.deq_timedelta >> 8);
hdr.telemetry.max_hop_delay =
    max(hdr.telemetry.max_hop_delay, queuing_delay);
```

Listing 1: Core P4 code for telemetry in Poseidon.



(a) Testbed topology (b) Create virtual hosts

Figure 11: Testbed with 8 virtual sender/receiver ports

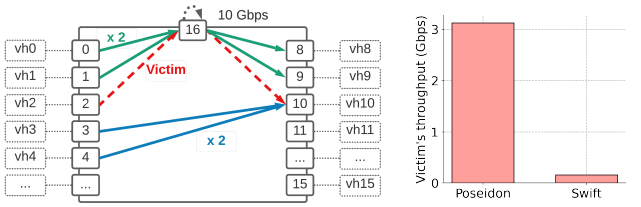
traffic insides the switch based on the virtual IPs into separate loopback ports. Each port, loopback at MAC layer, is configured to 10Gbps and plays the role of a virtual sender/receiver (Figure 11(b)). Ports 0-7 receive traffic from host 1, and ports 8-15 pass the traffic to host 2.

Testbed Results: To create multi-hop and reverse-path congestions from Figure 2 and Figure 3, we route the flows between virtual senders/receivers as Figure 12(a) and Figure 13(a) show. For the multi-hop congestions, Poseidon could fairly share the bandwidth between background flows and the victim flow, while Swift only spares 0.16 Gbps for the victim flow in Figure 12(b). For the reverse-path congestion, Poseidon could achieve line rate for the victim flow, while Swift could only achieve 1.91 Gbps (similar throughput as the flows on the reverse-path) as shown in Figure 13(b).

5.2 Robustness From Max-min Fairness

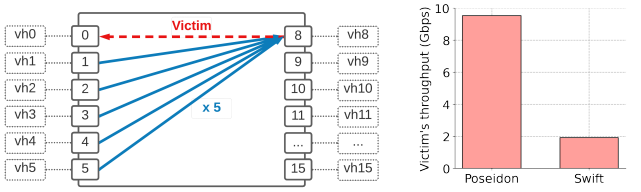
Poseidon achieves fairness in multi-hop congestion. Consider the scenario in Figure 14(a) where we have M green flows at Rack 0 and N blue flows at Rack 10. We add more flows to change M and N to create different multi-hop congestion scenarios. With Swift, the moment we have congestion at multiple hops ($M > 0$ and $N > 0$), the victim flow, red, cannot compete with other flows (Figure 14(b)). The reason is that Swift reacts to the inflated sum of delays (Figure 14(c)). Therefore, the victim reduces its congestion window until its scaled (because of flow-scaling) target delay matches this larger end-to-end delay. HPCC and DCTCP also react to the congestion at *any* hop, thus when $M = N$, the victim does an MD when *either* of the hops gives a congestion signal (ex: ECN) and cannot achieve the fair rate.

Poseidon, however, allows the victim flow to achieve its max-min fair share ($200Gbps/max(M+1, N+1)$) by only reacting to the bottleneck hop where it gets the fair-share. One reflection is that Poseidon's congestion signal, max-hop delay, and target only changes when the bottleneck hop or its congestion changes. For example, they stay the same when



(a) Victim from virtual host 2 (vh2) contends with 2 flows on port 16 and 2 flows on port 10. (b) Victim achieves fair-share rate in Poseidon.

Figure 12: Multi-hop congestion. (Linerate: 10 Gbps)



(a) 4 flows create a congestion on the victim's reverse-path. (b) Victim achieves linerate (10 Gbps) in Poseidon.

Figure 13: Reverse-path congestion. (Linerate: 10 Gbps)

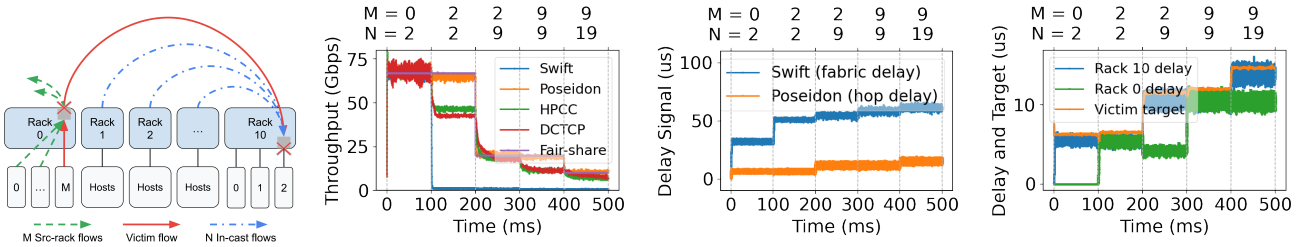
M changes from 0 to 2, but change when N increases from 2 to 9 in Figure 14(c) and Figure 14(d). Although the victim flow experiences higher RTT than other flows, Poseidon uses a higher congestion window to achieve the fair rate. Another interesting point happens when both hops have the same fair-share rate ($M = N$). Although the delay of both hops is close to the target (Figure 14(d)), with a rate-adjusted target, the moment the victim reduces its window, Poseidon raises its target and will not react to the max-hop delay until the rate increases again. §5.6 shows that both max-hop latency and scaling the target are necessary to achieve fairness.

Poseidon utilizes forward path regardless of reverse-path congestion. Reproducing the scenario in Figure 3, Figure 15(a) shows that with Swift, as the number of flows on the reverse-path, N, increases, victim's throughput decreases to the fair-share rate in reverse-path. However, with Poseidon, the victim could maintain 200 Gbps (line rate). The reason is that Poseidon only uses the max-hop delay from the forward path, which is not affected by the reverse traffic (Figure 15(b)). HPCC doesn't have this problem since it only uses INT information on the forward path.

5.3 Fast Convergence and Stability

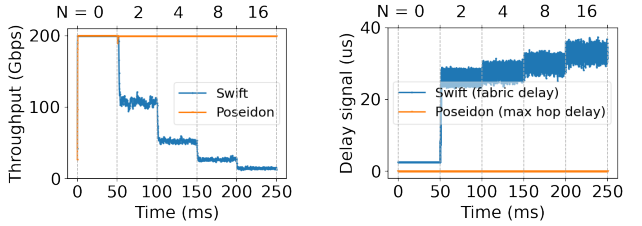
Figure 16 shows the rate of flows in Swift and Poseidon as we add competing flows one by one and then remove them. At a *single* hop, not only does Poseidon achieve the fair-share, similar to Swift, but also lower throughput variation, hence better stability. Next, we evaluate Poseidon's convergence time and throughput stability.

Poseidon converges fast for flows with large windows. Figure 17(a) shows the ramp-up phase of a flow, growing its window to a large value. This flow is competing with another one on a 200G link. First, the ramp-up shows that Poseidon does fewer rate reductions than Swift and HPCC. Second, it shows that Poseidon achieves a super-linear ramp-up at



(a) Multi-hop congestion scenario (b) Victim flow's rate (c) Victim's delay signal (d) Victim's target & per-hop delays

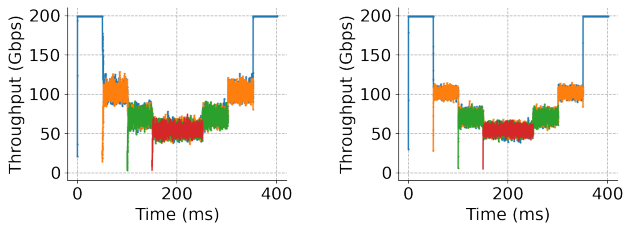
Figure 14: Multi-hop congestion with the same or different fair-share rate on different hops (linerate: 200 Gbps).



(a) Victim's rate is protected by Poseidon (linerate: 200 Gbps) from the congestion on the reverse-path.

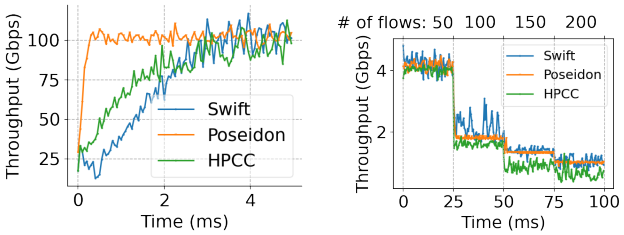
(b) Victim's congestion signal never changes for Poseidon, despite increasing delay on the reverse-path.

Figure 15: Reverse-path congestion: N reverse flows



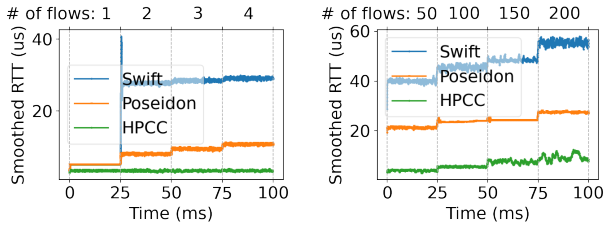
(a) Single hop fairness in Swift (b) Single hop fairness in Poseidon

Figure 16: Fairness on a single hop with step-in&out flows, throughput is measured every 50 μs.



(a) Fast convergence for big windows (b) Stability under high concurrency

Figure 17: Poseidon achieves fast convergence for flows with large windows & stable rate for flows with small windows.



(a) Large window: +1 flow per 25ms (b) Small wnd: +50 flows per 25ms

Figure 18: Poseidon achieves lower RTT than Swift by keeping queues short and stable.

the beginning, and the rate of growth decreases as it reaches the fair-share rate as expected in Figure 7(a). As a result, Poseidon converges around 12× faster than Swift and HPCC.

Poseidon achieves stable throughput for flows with small windows. Figure 17(b) shows the throughput for a flow competing with $N-1$ others on a hop. Poseidon reduces the standard deviation of throughput by 24× for $N = 200$ (70% on average over the four cases). As mentioned in §3.3, the reason is that AIMD in Swift and HPCC becomes more aggressive for smaller congestion windows. By contrast, Poseidon uses an adaptive update ratio to adjust the congestion window.

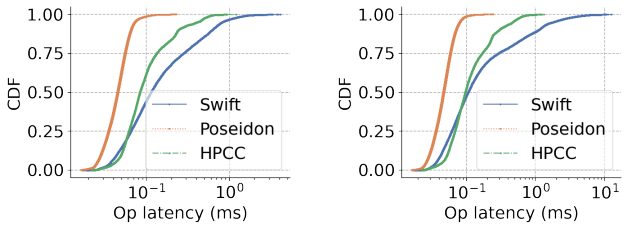
Poseidon keeps link utilization high with low RTT. Poseidon achieves a smaller RTT than Swift for flows across different rates, which means lower latency for small messages. For example, Figure 18 compares the RTT over different numbers of flows in two cases: large window and small window. Swift cannot use a very low target delay because, for high link utilization, it has to accommodate the summation of delays on multiple hops and the variation of delay in a high-degree incast caused by AI (Figure 17(b)). Since the adaptive update ratio can stabilize the rate, Poseidon could afford to use a tighter target and achieve high link utilization and small queues at the same time. HPCC, however, achieves lower RTT than Poseidon as it targets near-zero in-network queues at the cost of op latency (§5.4).

5.4 Application-level Improvements

A key application-level performance metric is op latency, namely, the time from a message was enqueued for sending to its completion [22]. We create two scenarios on two racks (A and B) with 3:1 oversubscription and compare op latency for 128 KB messages:

1) Uniform Random (UR): Rack A sends 960 Gbps to Rack B (60% uplink load), while Rack B sends 480 Gbps to Rack A (30% uplink load). The source and destination hosts are randomly chosen. Poseidon has a 61% lower median and 14.5× lower 99.9p op latency than Swift (PLB [44] enabled), 44% lower median and 5.49× lower 99.9p op latency than HPCC in Figure 19(a). This mostly comes from robustness to reverse-path and multi-hop congestion.

2) Uniform Random with Rotating Incast (UR+RI): A and B communicate similar to UR scenario, but Rack A also suffers from rotating incast from 100 hosts in other racks (not A or B). The incast traffic has 100 flows with 0.5 Gbps load



(a) Uniform random (UR) (b) UR + Rotating Incast (RI)

Figure 19: Poseidon improves op latency (FCT)

(50G in aggregate) and changes its target after sending a message from each host. Poseidon achieves 56% faster median and 41× lower 99.9th op latency for UR traffic than Swift (PLB [44] enabled), 51% faster median and 6.25× lower 99.9p op latency than HPCC in Figure 19(b). Besides being robust to reverse-path congestion, Poseidon allows UR flows to ramp up faster than Swift when the rotating incast targets another victim.

5.5 Brownfield Evaluation

With brownfield deployments, Poseidon achieves partial performance gains over Swift. We repeat the UR scenario explained in §5.4 over 4 racks. There are 24 hosts in each rack connected through 6 hops to hosts in other racks. Only ToRs have 3:1 oversubscription. Figure 20(a) compares the op latency of Poseidon with INT at all switches vs. Poseidon in the brownfield where only 2 or 4 ToRs support INT. It shows that Poseidon can achieve most of the gains compared to Swift in the brownfield.

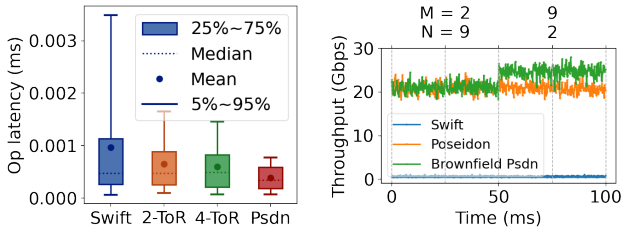
To evaluate the fairness scenarios in §4.1, we use the topology in Figure 14(a) and enable INT only on Rack 10. We make sure all blue flows send to the same host and all green flows send to another to create congestion on multiple hops for them. In Figure 20(b), when $M = 2, N = 9$, the bottleneck of red and blue flows is on Rack 10 that has INT, thus victim and other blue flows reach the fair rate (20 Gbps). When $M = 9, N = 2$, the bottleneck of red and green flows is on Rack 0, which doesn't have INT, and brownfield Poseidon gives a little more bandwidth to the red flow because the green flows have to react to the end-to-end delay. The unfairness is bounded because green flows increase their target until it covers the sum of delay on their congested hops.

5.6 Sensitivity Analysis

5.6.1 Ablation Study

To show the importance of each major aspect of the Poseidon design, we use the same algorithm and parameters as Poseidon, but remove one design aspect each time: 1) **maximum per-hop delay (MPD)** information instead of RTT; 2) **rate-adjust** target for per-hop delay; and 3) **adaptive increase ratio** algorithm instead of AIMD.

Figure 21(a) compares the throughput in the multi-hop scenario (Figure 14(a)). It shows that network-wide fairness is only achieved when using *both* rate-based target scaling



(a) Message op latency shrinks when more switches are equipped with INT capability. (b) Unfairness is bounded when the bottleneck is not on the INT-capable switches.

Figure 20: Partial gains and fairness in brownfield Poseidon

and max hop delay. However, removing the adaptive update ratio will not harm the max-min fairness, and Poseidon can achieve fairness even using AIMD. Figure 17 has already shown that AIMD slows down ramp-up and causes wider throughput variations in the presence of many flows.

5.6.2 Robustness of Parameters

Though Poseidon could achieve the design targets with a wide range of functions and parameters, it is worthwhile to understand the trade-off of each parameter. In this section, we vary the three parameters in Eq. 3 and Eq. 4 and show why we choose: $p = 40$, $k = 2$, and $m = 0.25$.

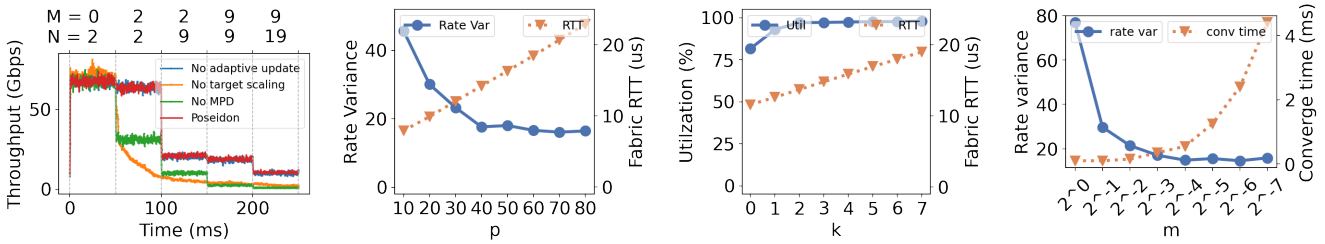
p controls the range of target scaling, affecting round-trip time and rate variation. Figure 21(b) shows that when we have congestion from hundreds of flows, a higher p allows reacting to rate unfairness faster and reduces rate variations. However, that means enduring larger RTT in the network.

k avoids under-utilizing the link bandwidth. Figure 21(c) shows the utilization % when we have a few flows on the bottleneck (where the rate is close to max_rate thus the target is close to k) vs. the fabric RTT when we have hundreds of flows. For small k values, the fluctuation of the queuing delay may lead to link under-utilization as the target is low and flows reduce the congestion window conservatively. However, if the value of k is too large, the RTT will increase.

m determines the trade-off between the variance of flow rates and the convergence speed. Figure 21(d) compares the convergence time in Figure 17(a) experiment and rate variation in Figure 17(b) for different values of m. Larger m values improve stability as they dampen the effect of the target in Eq. 4 but also slow down convergence.

6 Related Work

Delay-based: Swift [35], the basis of Poseidon, is a state-of-the-art delay-based algorithm that relies on hardware timestamps from NICs. Swift has some elements of Poseidon, although for different purposes. 1) It separates fabric delay from engine delay and tracks a separate congestion window for each. However, this separation was because of fundamental differences in congestion at fabric hops vs. hosts, and doesn't address fabric issues explained in §2.1. 2) Swift uses a larger target delay for flows with smaller congestion windows to address synchronized packet arrival from many flows on a



(a) Ablation study (b) p: maximum target scaling (c) k: base target (d) m: update intensity

Figure 21: Ablation study in multi-hop congestion scenario and sensitivity analysis over three parameters.

bottleneck link (flow scaling for windows <10), which forces flows to converge to the same window. Appendix §C shows even if we combine flow-scaling with max-hop latency, Swift still faces unfairness. 3) For multiplicative decrease, Swift decrease depending on how far the delay is from the (almost fixed) target, but still uses a fixed step for additive increase. At a higher level, Swift only looks at the end-to-end delay while Poseidon uses max-per hop latency and rate-based target scaling to respond to the bottleneck hop. Copa [12] adjusts the target rate based on *end-to-end delay* to achieve short queues and fair allocation, but Poseidon compares the max-hop delay to a rate-adjust target to reach *network-wide max-min fairness*.

ECN-based: ECN can be seen as a one-bit INT signal from switches that is set based on a configurable threshold inside switches. It is successfully deployed in datacenters and used by end-to-end CC algorithms such as DCTCP [8] because ECN was non-intrusive (two bits in the IP header) and those algorithms were deployable in brownfield environments. Still, ECN-based algorithms do not recognize the bottleneck hop and *all* flows react to any hop in their path that marks packets.

Richer signals from switches: XCP [34] and RCP [22] get help from switches to enable flows to react to congestion and achieve the fair-share allocation. In particular, XCP introduced the idea of decoupling utilization from fairness. However, both proved difficult to deploy in datacenters because of the lack of a brownfield solution and the overhead in high-speed switches. Poseidon achieves fairness using target scaling and introduces adaptive update ratios to reach better stability. It is deployable in brownfield and requires minimal changes in hardware to support max-hop delay in INT.

HPCC [40] uses queue length, timestamp, and tx-bytes of each hop to estimate in-flight bytes on each link and update a congestion window in an AIMD fashion in order to achieve very low queuing in the network. However, HPCC doesn't recognize the bottleneck hop: high utilization on *any* hop along the path should not force a flow that didn't get the link's *fair-share* to reduce rate. In addition, HPCC assumes all flows experience the same base RTT, relies on the additive increase to achieve fairness, doesn't address brownfield deployment, and requires bandwidth and CPU overhead from three INT metadata *per hop*. In contrast, by using a novel target-scaling solution, Poseidon achieves fairness without relying on AIMD, supports brownfield deployment, and only needs *a single max hop delay per packet*. PowerTCP [7] argues that CC should react to both absolute CC signal and its change rate to avoid slow reaction or overreaction to queue build-up. Poseidon's adaptive update ratio in Figure 7(a) addresses this issue. PowerTCP similar to HPCC still looks at congestion at *any* hop and uses per-hop INT metadata.

Receiver-driven: NDP [29] and HOMA [42] face challenges in oversubscribed networks where may have congestion in the core. However, Poseidon is insensitive to over-subscription, and we expect similar gains on op latency by applying Poseidon's idea to receiver-driven schemes.

Combined with schedulers: HOMA [42] combines a CC algorithm with a scheduling policy that prioritizes the shortest remaining flows to achieve shorter flow completion time. While Poseidon is currently a pure CC algorithm, we believe it has the potential to be integrated with similar scheduling policies and preserve the benefits of fast convergence and robust performance.

7 Conclusion

We proposed *Poseidon*, a congestion control algorithm that reduces op latency through fast convergence and lower latency and is robust in multi-hop and reverse-path congestion by leveraging in-band network telemetry (INT) in a novel way. Poseidon only needs a single max-hop delay per packet from INT, which makes it easily deployable with low overhead. We showed how INT packets can be deployed in brownfield and how Poseidon can still gain from an incremental deployment. In the future, we plan to implement Poseidon in NIC offloading protocols (e.g., RDMA), leverage INT to break down the delay at end-host networking stacks, use INT to hint path changes to avoid hash collisions [33], and apply the target scaling idea to other congestion signals, such as in-flight bytes [40], to achieve lower in-network delay.

This work doesn't raise any ethical issues.

Acknowledgment

We would like to thank our shepherd Paolo Costa and the anonymous NSDI reviewers for providing valuable feedback. We thank the production, serving, and support teams at Google for their contributions to the work and the platform. T. S. Eugene Ng is partially supported by the NSF under CNS-2214272 and CNS-1815525.

References

[1] How Distributed Shuffle improves scalability and performance in Cloud Dataflow pipelines, 2018. https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines.

[2] In-band Network Telemetry (INT) Dataplane Specification, 2020. https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[3] Amazon EC2: Linux accelerated computing instances: Networking performance, 2021. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/accelerated-computing-instances.html#gpu-network-performance.

[4] Encryption in Transit in Google Cloud , 2021. https://cloud.google.com/security/encryption-in-transit.

[5] Tomahawk4 / bcm56990 series, 2021. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series.

[6] PSP Architecture Specification, 2022. https://github.com/google/psp.

[7] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *NSDI*, 2022.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[9] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[10] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *NSDI*, 2020.

[11] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. NanoTransport: A low-latency, programmable transport layer for NICs. In *SOSR*, 2021.

[12] Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based congestion control for the internet. In *NSDI*, 2018.

[13] Sanjeewa Athuraliya, Victor H Li, Steven H Low, and Qinghe Yin. REM: Active queue management. In *Teletraffic Science and Engineering*, volume 4, pages 817–828. 2001.

[14] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[15] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *SIGCOMM*, 2020.

[16] Dimitri Bertsekas and Robert Gallager. *Data networks*. Athena Scientific, 2021.

[17] Neal Cardwell, Yuchung Cheng, et al. BBR Update:1: BBR.Swift; 2: Scalable Loss Handling. IETF 109. https://datatracker.ietf.org/meeting/109/materials/slides-109-iccrg-update-on-bbrv2-00, Nov 2020.

[18] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[19] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[20] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.

[22] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, 2006.

[23] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI*, 2016.

[24] Vishal Fadia and Philip Wells. Turbo boost your compute engine workloads with new 100 gbps networking, 2021. https://cloud.google.com/blog/products/networking/increasing-bandwidth-to-c2-and-n2-vms.

[25] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[26] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *NSDI*, 2022.

[27] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. Backpressure flow control. In *NSDI*, 2022.

[28] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[29] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, 2017.

[30] Christian Hopps. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November, 2000.

[31] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient remote storage access with i10. In *NSDI*, 2020.

[32] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329, 1988.

[33] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. FlowBender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *CoNEXT*, 2014.

[34] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, 2002.

[35] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Mike Ryan, David J. Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[36] J. Kumar, S. Anubolu, J. Lemon, R. Manur, H. Holbrook, A. Ghanwani, D. Cai, H. ou, and Y. Li X. Wang. Inband flow analyzer, 2021. https://datatracker.ietf.org/doc/html/draft-kumar-ippm-ifa.

[37] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *NSDI*, 2021.

[38] Jean-Yves Le Boudec. Rate adaptation, congestion control and fairness: A tutorial. *on line*, 2000.

[39] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *OSDI*, 2020.

[40] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *SIGCOMM*. ACM, 2019.

[41] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[43] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[44] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. PLB: Congestion signals are simple and effective for network load balancing. In *SIGCOMM*, 2022.

[45] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *NSDI*, 2021.

[46] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A decade of clos topologies and centralized control in Google's datacenter network. In *SIGCOMM*, 2015.

[47] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *NSDI*, 2019.

[48] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *EuroSys*, 2014.

A Poseidon Algorithm

Algorithms 2 shows the *RetransmitTimeout* and *FastRecovery* functions called in Algorithm 1 for completeness.

Algorithm 2: Poseidon's CWND Update Algorithms

```
1  Function RetransmitTimeout():
2      retransmit_count ← retransmit_count + 1
3      if retransmit_count ≥
         RETX_RESET_THRESHOLD then
4          cwnd′ ← min_cwnd
5      else
6          if now − t_last_decrease > rtt then
7              cwnd′ ← cwnd ∗ min_md
8      return cwnd′
9  Function FastRecovery():
10     retransmit_count ← 0
11     if now − t_last_decrease > rtt then
12         cwnd′ ← cwnd ∗ min_md
13     return cwnd′
```

B A Valid Cluster of Functions

We prove that the cluster of functions in Eq. 3 and Eq. 4 satisfy Eq. 1 and Eq. 2.

The target functions are:

$$T(x) = p * \frac{ln(max_rate) - ln(x)}{ln(max_rate) - ln(min_rate)} + k \quad (6)$$
$$min_rate < x < max_rate, p > 0, k > 0$$

Then we give a cluster of update functions, which is specifically designed for the above target functions:

$$U(x,D) = \exp\left[\frac{T(x)-D}{p} \cdot \alpha \cdot m\right] \quad (7)$$
$$\text{where } \alpha = ln(max_rate) - ln(min_rate)$$

B.1 Proof for Target Functions

When delay $D \le T(x)$:

$$Update(x,D) = \exp\left[\frac{T(x)-D}{p} \cdot \alpha \cdot m\right] \ge 1 \quad (8)$$

When delay $D > T(x)$:

$$Update(x,D) = \exp\left[\frac{T(x)-D}{p} \cdot \alpha \cdot m\right] < 1 \quad (9)$$

Thus, Eq. 3 satisfies Eq. 1.

B.2 Proof for Update Functions

Without loss of generality, assume two flows' rates $a < b$, delay is D.

For the rhs, since T(a) > T(b):

$$\begin{aligned}\frac{U(b,D)}{U(a,D)} &= \frac{\exp\left[\frac{T(b)-D}{p} \cdot \alpha \cdot m\right]}{\exp\left[\frac{T(a)-D}{p} \cdot \alpha \cdot m\right]} \\ &= \exp\left[\frac{T(b)-T(a)}{p} \cdot \alpha \cdot m\right] \\ &< 1\end{aligned} \quad (10)$$

For the lhs:

$$\begin{aligned}\frac{U(b,D)}{U(a,D)} &= \frac{\exp\left[\frac{T(b)-D}{p} \cdot \alpha \cdot m\right]}{\exp\left[\frac{T(a)-D}{p} \cdot \alpha \cdot m\right]} \\ &= \exp\left[\frac{T(b)-T(a)}{p} \cdot \alpha \cdot m\right] \\ &= \exp\left[p \cdot \frac{ln(a)-ln(b)}{\alpha} \cdot \frac{1}{p} \cdot \alpha \cdot m\right] \\ &= \exp\left[m \cdot (ln(a)-ln(b))\right] \\ &= \exp\left[m \cdot ln(\frac{a}{b})\right] \\ &= \left(\frac{a}{b}\right)^m\end{aligned} \quad (11)$$

So as long as $m < 2$, we can have

$$\frac{U(b,D)}{U(a,D)} = \left(\frac{a}{b}\right)^m > \frac{a^2}{b^2} \quad (12)$$

Thus, Eq. 4 satisfies Eq. 2.

B.3 Updating Based on Ratio vs. Distance

A valid update function with the same target function as in Eq. 3 is to use the ratio of target and max-hop delay. This can be seen as an extension of the Swift's MD function.

$$U(T(rate), delay) = \frac{T(rate)+m}{delay+m}, m \ge 0 \quad (13)$$

A problem with Eq. 13 is that it scales its update ratio depending on the value of delay. As an example, suppose that m is negligible, and we went 1 μs above the target. If the target is 4, the update ratio will be 0.8, but if the target is 30 (high concurrency scenario), the update ratio will be 0.968.

This means that for high concurrency scenarios where fair-share rate is low, and the target is high, the convergence will be slow. For example, Figure 22 compares the op latency in UR+RI scenario introduced in §5.4 for the update function based on the distance in Eq. 4 vs. the function based on the ratio in Eq. 13. The update function based on the distance clearly has an advantage at the tail.

C Flow Scaling in Swift

Swift uses flow scaling to inflate target delay to compensate for synchronized packet arrivals. The authors in [35] noticed

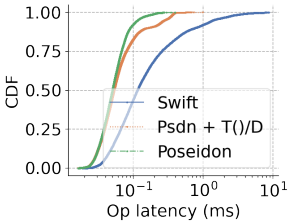
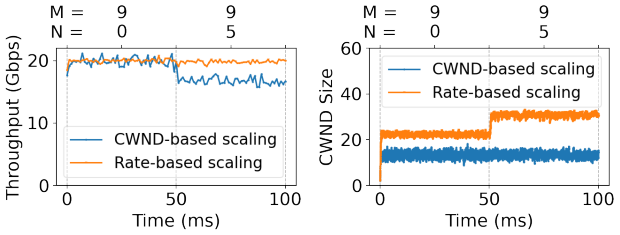


Figure 22: Poseidon can achieve lower op latency using the update function based on the distance of targe and max-hop-delay



(a) Rate of the victim flow (b) CWND of the victim flow

Figure 23: Compare scaling the target using the rate or congestion window in Poseidon.

that the average queue length grows as $O(\sqrt{N})$ where N is the number of flows on a link. Swift adjusts the target in proportional to $1/\sqrt{cwnd}$ because it argues that the *cwnd* trend is inversely proportional to the number of flows when Swift converged to its fair-share. The flow scaling in Swift also helps fairness as it speeds slow flows with a larger target, and slows fast flows with a smaller target. However, the flow scaling of Swift is not applicable to max-hop delay to find the bottleneck hop because of two reasons: 1) Its effect is nominal for windows > 10 (See Figure 5 in [35]) and more importantly 2) The formulation that reached $1/\sqrt{cwnd}$ assumes, at fair-share, flows see the same RTT (≈target delay) and pushes them to have the same *cwnd*. However, to solve the scenarios in §2.1 and get the fair-share, we only need to react to the congestion at the bottleneck hop. This means that flows get different RTTs and pushing flows to get the same *cwnd* cannot achieve the fair share rate ($rate = \frac{cwnd}{RTT}$). For example, in the fair-share allocation of multi-hop congestion scenario, the victim flow will have higher RTT and needs higher *cwnd* to achieve the fair-share.

We show this shortcoming in the following equations. Suppose that the link capacity is C, and the congestion window and RTT for flow i are $cwnd_i$ and RTT_i. The target delay is calculated as follows, where t_{base} is the base delay in Swift and A is just a constant.

$$t = t_{base} + A \cdot \frac{\sqrt{N}}{C} \tag{14}$$

$$\text{Fair share for flow i} = \frac{cwnd_i}{RTT_i} = \frac{C}{N} \tag{15}$$

If we **assume** that at the steady state RTT_i is equal to t (target delay) for all flows and thus *cwnds* are equal to w in order to get the same throughput, we can estimate $\sqrt{N}$ from Eq. 15 as follows

$$\sqrt{N} = \frac{A + \sqrt{A^2 + 4 \cdot C \cdot w \cdot t_{base}}}{2 \cdot w} \tag{16}$$

Therefore, $\sqrt{N}$ can be estimated by $\frac{\alpha}{\sqrt{w}} + \beta$.

However, as explained above, to achieve fair-share rate, flows get different RTTs thus converging to the same congestion window is not fair. Now, we show that if we follow the formulation of how Swift reached $1/\sqrt{cwnd}$ for cases that flows have different RTTs at fair share, we end up with a valid Poseidon rate-based scaling. We repeat Eq. 14 here as Eq. 17 after changing t to t_{hop} to emphasize that for Poseidon we have a target for per-hop delay.

$$t_{hop} = t_{hop_base} + A \cdot \frac{\sqrt{N}}{C} \tag{17}$$

If we combine Eq. 17 and Eq. 15 we get

$$t_{hop} = t_{hop_base} + \frac{A}{\sqrt{C}} \cdot \sqrt{\frac{RTT_i}{cwnd_i}} = \frac{\alpha}{\sqrt{\frac{cwnd_i}{RTT_i}}} + \beta \tag{18}$$

$\frac{cwnd_i}{RTT_i}$ in Eq. 18 is the rate of flow i. Therefore, for the flow scaling of Swift to work in a fair-share setting where flows can have different RTTs, the target should increase in reverse relation to rate not just *cwnd*. Figure 23 compares the throughput and congestion window of the victim flow in Poseidon if it uses the above target function using rate vs *cwnd* in the multi-hop congestion scenario (Figure 14(a)). The victim and $N = 9$ flows start at time 0. Then at 50 ms, $M = 5$ flows start to create congestion at the source rack. Figure 23(b) shows that after 50ms, target scaling based on the rate converged to a higher *cwnd* to keep the throughput the same for the victim flow.

Eq. 18 is a special case of $T(b) = p \cdot b^q + k$, a valid cluster of functions that satisfy Eq. 1 and Eq. 2 for $-2 \leq q < 0$, with $q = -0.5$. However, we believe Poseidon's function in Eq. 3 is a better function as explained in §3.3.

D Proof of Lemma 1

We first repeat the Lemma here: When achieving network-wide max-min fairness, each flow will have the largest rate among all flows on its bottleneck hop and not on any other saturated hop. Formally, for the "max-min fair" allocation $\vec{x}$, for any flow s, denote the flows shared the same bottleneck with s as $\{b_1, b_2, ..., b_k\}$. For any flow b_i, $x_s \geq x_{b_i}$. Denote the flow's share on the saturated non-bottleneck hop of s as $\{c_1, c_2, ..., c_k\}$, then there must exist some c_j such that $x_{c_j} > x_s$.

Proof: Assume there exists a flow s that has reached its fair-share rate r, and there is another flow s' on its bottleneck hop with an even larger rate r'. But this state is not max-min fair

because flow s could get some bandwidth from flow s' and let them have the same rate $\frac{r+r'}{2}$. By contradiction, the flow s has the largest rate on its bottleneck hop.

On the other hand, assume there exists a flow s, which is the fastest flow, with rate r, on one of the non-bottleneck hops. However, given that this link is congested, its fair-share flows with rate r' could obtain bandwidth from flow s and increase their fair-share rate to at least $\frac{r+n\cdot r'}{n+1}$, where n is the number of fair-share flows. By contradiction, the flow s cannot be the largest flow on its non-bottleneck hop.

E Proof of Convergence to Max-min Fairness

Problem description:

For any network topology, any traffic pattern (flows' source and destination, routing), given an initial flow rate allocation, Poseidon converges to the max-min fair allocation.

Notations:

Denote all the link bandwidth as B;

Denote the target for flow with rate r as $T(r)$;

Denote a flow rate allocation as $\vec{y}$;

In allocation $\vec{y}$, denote the rate of flow f as y_f;

In allocation $\vec{y}$, denote the maximum flow rate on a saturated queue q as R_q^y;

In allocation $\vec{y}$, denote the delay on a queue q as D_q^y;

Denote the max-min fair rate allocation as $\vec{x}$;

In max-min fair allocation $\vec{x}$, denote the maximum flow rate on a saturated queue q as R_q^x, which is also the fair-share rate of that port.

Designs of Poseidon and observations:

Design 1: Poseidon reacts to the maximum hop delay along the path.

Design 2: In Poseidon, the target of a flow increases when the flow rate decreases. And Poseidon decreases the flow rate when the delay is higher than the flow's target; increases the rate when the delay is lower than the target.

Observation 1: The queuing delay on a saturated port is no larger than the target of flows with the fastest rate on that port.

Observation 1 holds true because of design 2: if the delay exceeds the target of a flow, that flow will decrease its rate immediately. However, in Poseidon, the decrease operation only happens once per RTT, so the reaction of decreasing rate may happen at most one RTT later. But this will not affect the overall trend of queuing.

Observation 2: the queuing delay on an unsaturated port is always 0.

Observation 2 holds true when senders send packets without bursts. However, the synchronized arrival of many flows may create a transient queue. But the queue will disappear within 1 RTT, because the average data sent within one RTT is less than the line rate.

Proof:

We will use induction to prove any allocation $\vec{y}$ will converge to max-min fair allocation $\vec{x}$.

In allocation $\vec{x}$, if we sort the saturated queues based on their maximum flow rate (fair-share rate), we can get:

$$R_{q_1}^x \leq R_{q_2}^x \leq ... \leq R_{q_k}^x \tag{19}$$

$$T(R_{q_1}^x) \leq T(R_{q_2}^x) \leq ... \leq T(R_{q_k}^x) \tag{20}$$

(1) prove queue q_1 will converge to the max-min fair allocation:

For any allocation $\vec{y}$, for queue q_1, its fastest flow's rate is $R_{q_1}^y$. Note that this q_1 is still the same q_1 sorted by allocation $\vec{x}$.

Because the queue q_1 is saturated in $\vec{y}$, it has to satisfy:

$$\sum_{f \in Flows(q_1)} y_f \geq B \tag{21}$$

And we already know:

$$\sum_{f \in Flows(q_1)} x_f = B \tag{22}$$

Because in allocation $\vec{x}$, all the flows on queue q_1 has the same rate, which is $R_{q_1}^x$. Any other allocation $\vec{y}$'s largest rate cannot be as small as $R_{q_1}^x$ because their rates are not all equal, so we have:

$$R_{q_1}^y \geq R_{q_1}^x \tag{23}$$

$$D_{q_1}^y = T(R_{q_1}^y) \leq T(R_{q_1}^x) = D_{q_1}^x \tag{24}$$

Because of the same reason, we also have:

$$D_{q_i}^y = T(R_{q_i}^y) \leq T(R_{q_i}^x) = D_{q_i}^x, \forall i \in [2,k] \tag{25}$$

So for flows whose rates are smaller than $R_{q_1}^x$, their target is higher than delay on queue q_1 and also delay on any other queue q_i:

$$T(y_f) > T(R_{q_1}^x) \geq R_{q_1}^y, \forall y_f < R_{q_1}^x \tag{26}$$

$$T(y_f) > T(R_{q_1}^x) \geq T(R_{q_i}^x) \geq R_{q_i}^y, \forall y_f < R_{q_1}^x, \forall i \in [2,k] \tag{27}$$

Thus, those flows with smaller rate will keep increasing and flows with larger rate than $R_{q_1}^x$ will decrease because of the delay on queue q_1 or on other queues. Eventually, all of them will converge to the same target:

$$T(y_f) = T(R_{q_1}^x), \forall f \in q1 \tag{28}$$

So that:

$$T(R_{q_1}^y) = T(R_{q_1}^x) \tag{29}$$

Thus, we show that the queue q_1 will converge to the max-min fair allocation x

(2) Assume queue q_1 to q_m have already converged, prove queue q_{m+1} will converge:

Assume queue q_1 to q_m have already converged to max-min fair allocation $\vec{x}$, so we have:

$$T(R^y_{q_i}) = T(R^x_{q_i}), \forall i \in [1,m] \tag{30}$$

For queue q_{m+1}, the flows whose bottleneck is q_{m+1} will not travel queue q_1 to q_m. Because if they travel to one of those ports, those ports will have a higher fair-share rate, which contradicts the max-min fair allocation's conclusion.

Thus, with a similar analysis as the proof for step 1, we have:

$$T(y_f) > T(R^x_{q_{m+1}}) \geq R^y_{q_{m+1}}, \forall y_f < R^x_{q_{m+1}} \tag{31}$$

$$T(y_f) > T(R^x_{q_1}) \geq T(R^x_{q_i}) \geq R^y_{q_i}, \forall y_f < R^x_{q_1}, \forall i \in [m+2,k] \tag{32}$$

So that, the flows with smaller rate than $R^x_{q_{m+1}}$ will increase their rate, while flows with larger rate will decrease their rate, until:

$$T(y_f) = T(R^x_{q_{m+1}}), \forall f \in q1 \tag{33}$$

So that:

$$T(R^y_{q_{m+1}}) = T(R^x_{q_{m+1}}) \tag{34}$$

So we proved that queue q_{m+1} will also converge to max-min fair allocation $\vec{x}$.

In conclusion, all the ports in allocation $\vec{y}$ will eventually converge to the max-min fair allocation $\vec{x}$.

Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery

Taehyun Kim
KAIST

Deondre Martin Ng
KAIST

Junzhi Gong
Harvard University

Youngjin Kwon
KAIST

Minlan Yu
Harvard University

KyoungSoo Park
KAIST

Abstract

The recent advancement of high-bandwidth I/O devices enables scalable delivery of online content. Unfortunately, the traditional programming model for content servers has a tight dependency on the CPU, which severely limits the overall performance. Our experiments reveal that over 70% of CPU cycles are spent on simple tasks such as disk and network I/O operations in online content delivery.

In this work, we present IO-TCP, a split TCP stack design that drastically reduces the burden on CPU for online content delivery. IO-TCP offloads disk I/O and TCP packet transfer to SmartNIC while the rest of the operations are executed on the CPU side. This division of labor realizes the separation of control and data planes of a TCP stack where the CPU side assumes the full control of the stack operation while only the data plane operations are offloaded to SmartNIC for high performance. Our evaluation shows that IO-TCP-ported lighttpd with a single CPU core outperforms the Atlas server and lighttpd on Linux TCP for TLS file transfer by 1.8x and 2.1x, respectively, even if they use all 10 CPU cores.

1 Introduction

The demand for online content delivery is booming in recent years [1, 5]. Especially, the popularity of high-quality video streaming is growing rapidly [9, 56].For cost-effective streaming service, it is highly important for online video service providers [3, 4, 11, 19, 26, 44] to optimize their content delivery systems.

However, improving the content delivery performance is increasingly challenging as the growth of CPU capacity stagnates [50]. While modern innovation in I/O devices such as high-bandwidth NICs and NVMe disks has alleviated the I/O bottleneck, the lack of CPU cycles often fail to translate the high I/O performance into content delivery throughput. The root cause lies in the inefficiency of the modern OS abstraction which requires all disk data to be brought to main memory before being delivered to remote clients. For this reason, CPU (or more precisely, the memory subsystem) easily becomes the performance bottleneck for I/O-intensive applications like video content delivery as over 70% of its entire cycles are spent on simple I/O operations. To effectively harness the recent advancement in the I/O devices, the OS abstraction must reduce the dependency on CPU and its memory system for I/O operations.

Our approach to breaking the CPU dependency is to employ peripheral processors to handle the I/O operations. We observe that the recent programmable I/O devices such as SmartNICs [7, 24, 27, 33] or Computational SSDs [28, 40] may make up for the insufficient compute cycles in CPU. As the PCIe standard allows peer-to-peer DMA (P2PDMA) without the intervention of CPU [35], one can conceive a server system whose NIC offloads disk I/O operations completely from the CPU. In fact, recent works like DCS [46] and DCS-Ctrl [63] have demonstrated that an FPGA-based coordinator can perform all disk I/O operations via P2PDMA for a content delivery server. The main drawback of these systems is that they do not support TCP-based delivery commonly adopted by today's video streaming [3, 11, 26].

However, supporting TCP for an I/O-offloaded server raises an interesting question of function placement – if disk I/Os are offloaded to SmartNIC, where do we run the TCP stack? Running the TCP stack on CPU is impossible as the data for packet payload is unavailable. So, the obvious alternative is to run it on the NIC side. While it is non-trivial to implement a full TCP stack on an FPGA [1], it is possible to run it on SmartNIC. Actually, recent SmartNIC platforms support Arm-based embedded processors that run Linux with a full TCP stack [7, 33]. However, running the full TCP stack on NIC typically requires its application to co-execute on the same platform, with limited resources. In fact, we observe that the throughput of `nginx` on SmartNIC with 8 Arm cores is smaller than that with even a single CPU core.

We tackle this question with I/O Offloading TCP (IO-TCP), a split TCP stack design for I/O-intensive applications. The

[1] There are a few TCP/IP stacks on the FPGA [87, 88], but they simplify the key features with assumptions on the data center environment.

key idea of IO-TCP is to run only the "control plane" operations on CPU while delegating all "data plane" operations to SmartNIC that can access disks via P2PDMA. Figure 1 shows the overview of our design. The control plane includes all core functionalities of the TCP protocol – connection management, reliable data transfer, and congestion/flow control. On the other hand, the data plane operations refer to all aspects of data packet creation and transmission including content fetching from disks. This design ensures that the CPU side assumes full responsibility of controlling all operations while actual disk and network I/O operations are offloaded to SmartNIC under the hood. This design enables dedicating CPU cycles to complex control operations while exempting them from simple but repetitive I/O operations. The rationale for the design is that the split stack avoids CPU cache pollution from intensive disk IO [96] that slows down the control path operations, stretches the RTT, and lowers the throughput. In addition, the control path would benefit from advanced hardware features of modern CPU as it is compute-intensive with frequent random accesses and branches. In contrast, the data path depends more on memory bandwidth than computation, and it is easily parallelizable and can be even built into hardware.

While IO-TCP presents a great potential for saving CPU cycles, it brings a few new challenges. First, IO-TCP must handle TCP packet retransmission on SmartNIC without the timeout or packet loss information, which may issue redundant disk I/Os. To avoid the inefficiency, IO-TCP employs an internal ACK protocol to notify the SmartNIC of the data delivery so that it can safely throw it out from memory. Second, the RTT measurement in the host TCP stack could be inaccurate due to disk-induced delay on SmartNIC before transmission. In IO-TCP, actual data packet transfer is delayed until the packet content is fetched from the disk. However, disk I/O could add significant delay to packet transfer even without any congestion in the network path. IO-TCP addresses this challenge by carefully removing the disk-induced delay from RTT measurement. It employs an echo packet that allows the host stack to keep track of the packet departure time accurately. Third, IO-TCP must provide a well-defined API for an application to flexibly construct file or non-file content for data transfer. For this, IO-TCP extends the Berkeley socket API with a few "offload" functions that open a file and send the file content from the NIC. The "offload" functions are implemented as a form of API remoting, and the results are seamlessly delivered to the application on the CPU side.

We implement IO-TCP with the Mellanox BlueField-2 SmartNIC [31] that can directly access NVMe disks with P2PDMA. For the host stack, we extend an existing user-level TCP stack [58] to support I/O offloading while we implement the NIC stack with the DPDK library [12]. It requires 1,793 lines of code modification for the host stack and 1,853 lines of C code for the NIC stack. To evaluate the effectiveness with real-world applications, we also port lighttpd [23] to

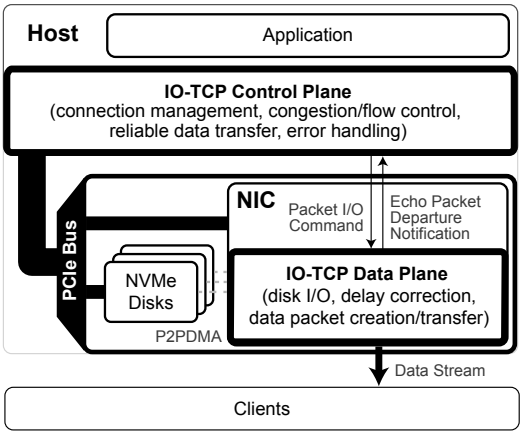


Figure 1: Overview of IO-TCP stacks

using IO-TCP with only about 10 lines of code modification.

Our evaluation demonstrates that IO-TCP-ported lighttpd achieves 77.4 Gbps of TLS video content delivery with a single CPU core, nearly saturating the full bandwidth of four NVMe disks. In contrast, the Atlas server [74] on FreeBSD and lighttpd on Linux reach only 44.2 and 37.4 Gbps, respectively, even with all 10 CPU cores. We observe that the current bottleneck of IO-TCP lies in the low memory bandwidth of the BlueField-2 NIC, but we believe the future version will achieve better performance. The main contributions of this work are summarized as follows. (1) We analyze the impact of CPU usage and cache interference by disk I/O on the performance of modern content delivery systems. (2) We present the design and implementation of IO-TCP, a split TCP stack design that fully leverages recent I/O advances in SmartNICs by separating TCP control and data planes. (3) We demonstrate how IO-TCP can surpass the limitations of the CPU bottleneck to achieve I/O bandwidths far greater than what the CPU could have normally performed.

2 Background & Motivation

We provide a brief background on content delivery systems in terms of recent trends in computing hardware.

2.1 Inefficiencies in Content Delivery System Stacks

Modern content delivery systems [2, 14, 26] consist of a large number of geographically distributed content delivery Web or reverse proxy servers. These systems serve as the basis for many applications such as video streaming and Web page accesses. Among them, the video traffic takes up about 60% of the entire Internet traffic and the overall volume has increased due to the recent pandemic [39, 56]. Average Web object sizes range from 0.01 to 1 MB while average video chunk sizes are between 0.2 to 1.5 MB [89].

For high performance, the server design has traditionally focused on optimizing disk access and CPU utilization because hard disk I/O is many orders of magnitude slower

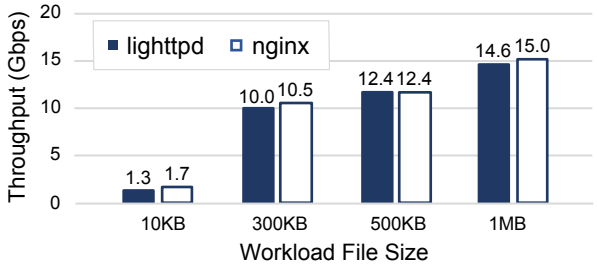


Figure 2: Throughputs of lighttpd and nginx on a single CPU core.

Function	% CPU
Read data from disk to a kernel buffer	33.53%
Memory management	21.93%
Move data to TCP send buffer (no copy)	10.30%
Open files and get stat	6.00%
Control Plane	*28.24%*
Total	**100%**

Table 1: CPU usage per data plane function in nginx when serving disk-bound workload with `sendfile()`. The breakdown for lighttpd is also similar.

than modern storage. For fetching small objects such as Web page content, the server is optimized to minimize the disk seeks while maintaining a small memory footprint for indexing [47]. For large-object access like video download, the server exploits sequential disk reads to maximize the disk throughput. Also, it typically employs `sendfile()` to avoid redundant memory copy and context switching between user and kernel spaces. To improve CPU utilization, the server typically takes the event-driven architecture [48, 67, 77, 81].

Traditional disk-based optimizations have become largely obsolete due to the advent of inexpensive large RAM and flash-based disks (e.g., NVMe SSDs) that removed the seek-induced limitations. Since the major disk bottleneck is lifted, the memory subsystem becomes the next bottleneck in today's server [74]. The problem is exacerbated by multiple memory copies due to disk and network I/O as well as content scanning for encryption and decryption. While a recent work [74] optimizes the disk access layer and exploits Intel Data Direct I/O (DDIO) [21] to arrange all such operations to perform with the data in CPU cache, it does not dissipate the workload from CPU. Also, it may be hard to expect a similar benefit if the workload exceeds the CPU cache size.

To better understand the performance of Web-based content delivery, we run experiments with two popular Web servers, lighttpd (v1.4.32) [23] and nginx (v1.16.1) [29] for disk-bound workload, which simulates a typical setting for HTTP-adaptive video streaming. The server setup is the same as in §5.1, and we use various file sizes that represent Web objects and video chunks of different quality. We configure the servers to use `sendfile()` for good performance.

Figure 2 shows the results with a single CPU core (refer to Figure 7 for performance trend over multiple CPU cores). The performances of both servers are similar, and they generally improve with larger file sizes. As our NVMe disk achieves

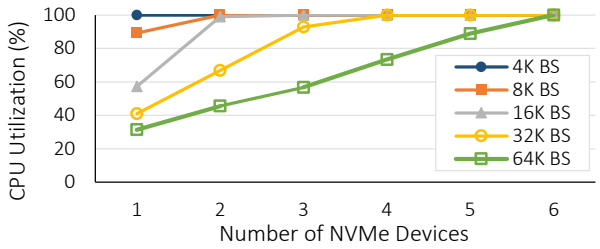


Figure 3: CPU utilization of `fio` with for varying number of NVMe disks. BS refers to block size.

around 2.5 GB/s (or 20 Gbps) per disk for random file reading, the single CPU core utilizes around half the bandwidth (10 Gbps) of a single NVMe disk for 300KB files. Considering that a server-class machine can carry 8 to 10 NVMe disks per CPU, CPU is a major source of resource bottleneck.

We analyze the CPU overhead in popular Web servers for content delivery. Table 1 shows the CPU cycle breakdown of nginx reported by `perf`[36]. `sendfile()` and `open()` take up the majority of the CPU cycles, which amounts to 71.76% of the consumed cycles. `sendfile()` reads the data on disk to kernel buffers, and serve it to clients without memory copy (33.35%). This clearly shows where the most of CPU cycles are spent in a content delivery server – disk and network I/O. Offloading these operations from the CPU would have a great potential for improving the performance.

2.2 Mismatch between I/O Device Advances and CPU Capacity

The capacity growth with recent I/O devices is impressive. Two decades ago, the fastest hard disk could achieve only about 200 random I/O operations per second (IOPS), but the recent NVMe disk can perform over 1 million IOPS [41, 97], a speedup of almost four orders of magnitude. For the same period, the bandwidth of an Ethernet NIC has improved by 400 times (from 1 Gbps in 1997 to 400 Gbps in 2021) while 800 Gbps / 1.6 Tbps Ethernet is expected to be standardized in a few years [55]. In contrast, CPU capacity improvement has been largely hampered by the end of Moore's law and breakdown of Dennard scaling [2] [54]. The first general-purpose multicore CPU appeared in 2005 [6], but the number of cores of Intel CPU has increased by only 28 times for 16 years [22].

Figure 3 shows the utilization of a single CPU core when saturating the NVMe disks with `fio` [15]. We use Intel Xeon Silver 4210 (2.20GHz) for CPU and Intel Optane 900P for NVMe devices. The figure indicates that it is relatively easy to handle large block sizes but a single core cannot saturate even 2 NVMe disks with a block size of 4KB. For 16KB blocks, it can handle up to 3 NVMe disks in parallel. Even when serving large files, disk I/O could still spend a significant portion of the CPU cycles as metadata access in filesystem would require frequent random accesses for small blocks.

[2] Dennard scaling dictates that the power density stays constant as the transistors become smaller. It is said to stop in 2006 and CPU capacity could only scale out since then.

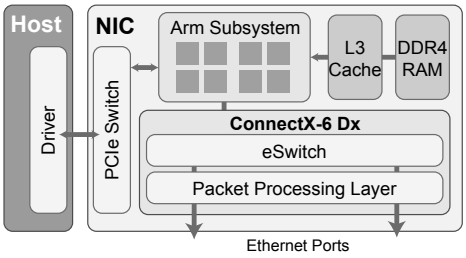


Figure 4: Architecture of the BlueField SmartNIC

In addition to NVMe, the use of persistent memory (PM) sees similar CPU bottlenecks. A recent PM performance study [94] on Intel Optane DC memory shows that 16 cores are required to fully utilize PM read bandwidth even with large I/O sizes like 64KB or 256KB. Due to the CPU bottleneck, many PM-based storage systems fall back to a lightweight storage stack design that misses features [51, 52, 53, 59, 64, 93, 98].

The performance disparity between CPU and I/O devices calls for revisiting the current OS abstraction for I/O operations, especially for serving large files with the growing trend of high-throughput content delivery. Existing OS requires CPU intervention for performing I/O operations such as reading disk content and transferring it via NIC. This is because the programming model on the current OS requires the content of the I/O device to be brought to main memory before performing any operation on the content. This memory-centric execution model wastes CPU cycles for frequent memory access stalls due to memory operations.

2.3 Opportunities with SmartNIC

The key idea of our work is to offload data I/O from CPU to a programmable I/O device while supporting TCP-based content delivery. Any programmable device that can perform direct disk I/O and network packet I/O can meet our goal, but we use SmartNIC as it serves as a convenient place to interact with remote clients. For example, recent SoC-based SmartNICs [7, 33] offer an Arm-based embedded system on top of a NIC data processing unit. These systems support direct access to NVMe disks on the same domain without intervention of CPU or main memory. More specifically, the Mellanox BlueField NIC supports P2PDMA via NVMe over Fabrics (NVMe-oF) target offload [18] through which the Arm processors can read directly from local NVMe disks. These disks are directly mounted on the Linux environment running on the Arm processors, and they run on the same file system as seen by the host OS.

Figure 4 shows the architecture of the Mellanox BlueField-2 NIC (BF-2) [31] that we use for our platform. It is equipped with 8 Armv8 cores and 16 GB of DDR4 memory that runs on Linux [3]. The Arm subsystem allows running DPDK applications to perform fast packet I/O either with remote machines or with the local host. In addition, applications can offload TCP/IP checksum calculation as well as TCP segmentation (i.e., TSO) to its ConnectX-6 Dx NIC hardware. The BlueField-2 also supports hardware acceleration for cryptographic operations that we use for supporting TLS.

[3] We run CentOS 7.6, but one can run embedded Linux like Poky [37].

lighttpd setup	Throughput (Gbps)
Linux TCP on BF-2 only	11.98
Linux TCP on BF-2 and 1 CPU core	22.02
IO-TCP-on BF-2 and 1 CPU core	44.13

Table 2: Performance of lighttpd with Linux TCP vs. IO-TCP for serving 300KB files over 1600 connections on BlueField-2 (BF-2) and a single CPU core. We use four Intel Optane 900P in all experiments.

With the Linux-operated SmartNIC, one might be tempted to use it as an extra server system [90]. However, running a server directly on SmartNIC does not efficiently use the resources. Table 2 compares the performances of lighttpd on only BF-2 (w/ all 8 cores), lighttpd on BF-2 and the host's single CPU core combined (by evenly dividing the request load), and IO-TCP-ported lighttpd on the same setup. Naïve scaling of processing power with SmartNIC ends up with only half the throughput of our solution (§4).

The experiments clearly show the current limitation with the SmartNIC – the processors and their memory are not so powerful as the host system. In fact, the Arm processors on BF-2 have 2.2x and 4.2x smaller L3 cache and memory bandwidth than those of our host CPU, which limits the overall performance. While this is not an inherent limitation as the next version [32] is reported to have 3.5x larger memory bandwidth, one should carefully design the offload functionality to effectively exploit the architectural difference.

3 Design

In this section, we present the design of IO-TCP that enables content delivery systems to leverage recent SmartNIC I/O advances. The key design choice of IO-TCP is to *separate the control and data planes* of the TCP stack such that the CPU stack takes the full control of every operation (control path) while individual I/O operations (data path) are offloaded to the SmartNIC stack. The core rationale for this is to save the majority of CPU cycles for performing I/O operations while keeping the SmartNIC stack simple to implement. Simplicity is the key to achieve the performance scalability.

There are three design goals for IO-TCP: (1) IO-TCP must conform to the TCP protocol and should be able to support various congestion control implementations. For example, handling disk I/O in the NIC stack should not compromise the congestion control logic in the host stack due to imprecise RTT measurements induced by disk access latency (§3.5). (2) The modification of existing applications should be minimal for migrating to IO-TCP – it should use the same socket API except for offloading file I/O (§3.2). (3) The IO-TCP host stack needs to communicate with the NIC stack for I/O offloading, and its overhead should be made small. In addition, the host stack should be notified of any failures in the NIC stack to

`int offload_open(const char *filename, int mode)` – opens a file in the NIC and returns a unique file ID (fid).
`int offload_close(int fid)` – closes the file for fid in the NIC.
`int offload_fstat(int fid, struct stat* buf)` – retrieve the metadata for an opened file, fid.
`size_t offload_write(int socket, int fid, off_t offset, size_t length)` – sends the data of the given length starting at the offset value read from the file, fid, and returns the number of bytes virtually copied to the send buffer.

Table 3: IO-TCP offload API functions

handle them in time (§3.3 and §3.6).

3.1 Separating TCP control and data planes

To save host CPU cycles, we need to determine which operations would benefit the most from offloading based on the capabilities of SmartNICs and CPU. The embedded processors on either SoC or ASIC-based SmartNIC are better fit for simpler data plane operations while x86 CPUs with advanced features [4] can handle complex control plane operations faster. To better reflect the architectural difference into the design, we divide the TCP stack into control and data plane operations.

The control plane functions refer to the key TCP protocol features such as connection management, reliable data transfer, congestion/flow control, and error control. These typically require complex state management as the behavior depends on the response from the other end. For example, reliable data delivery on the receiver side requires tracking all received data ranges that are disjoint for proper in-order delivery and ACK generation. It is also tightly coupled with congestion control as loss detection and packet retransmission for reliable delivery in turn re-adjust the send window size. Similarly, flow control needs to run with congestion control as they collectively determine the window size. Error control cannot run alone, either, as it requires tracking detailed flow states to infer any erroneous behavior. Theoretically, each individual operation can be offloaded, but it would be more efficient to offload them together. However, offloading them all could overload the SmartNIC as seen in experiments in §2.3.

The data plane operations refer to all operations that involve data packet preparation and transfer, which supports the implementation of control plane functions. These include managing data buffers, segmenting data into packets, calculating TCP/IP checksums, etc. IO-TCP offloads only the operations in the send path because they are simple, stateless, and easily parallelizable. In addition, IO-TCP offloads the file/disk I/O and combines it into TCP data plane operations. The rationale for offloading is that these operations would interfere with control plane operations on CPU as recent innovation like Intel DDIO would pollute the CPU cache by huge disk data [96]. Offloading them to SmartNIC would allow the control path to execute on CPU much faster, which in turn improves the data path performance. Also, SmartNICs tend to have hardware-based crypto accelerators [24, 27, 31], which enables TLS data encryption at line rate. Section 5.5 analyzes the source of performance improvement in depth.

[4] Such as larger CPU cache and vectorized instructions like AVX/AVX2.

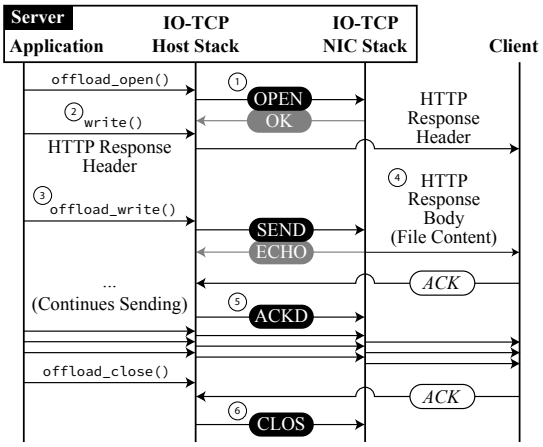


Figure 5: Content delivery from a Web server on IO-TCP

3.2 IO-TCP Offload API Functions

Ideally, porting an application to IO-TCP should require little modification of its core logic, yet it should flexibly express the application needs. For example, an IO-TCP application should be able to compose any data to transfer regardless of whether it is file content or not. Towards this goal, we extend the existing socket API by adding only four functions (see Table 3) for offloaded file and network I/O.

`offload_open()` asks the NIC stack to open a file and to report the result (either success or any error). It returns a file ID (instead of a file descriptor) that identifies the opened file in the NIC stack for later operations. `offload_open()` is an asynchronous function whose result should be checked with `epoll()` or subsequent function calls as file opening can fail for various reasons. After all file operations, the application can call `offload_close()` to close the file on the NIC stack. In addition, IO-TCP supports `offload_fstat()` that retrieves the metadata for a file (e.g., file size and permission).

With the opened file ID, the application can call `offload_write()` to send the file content on a TCP connection. Essentially, `offload_write()` carries out the same operation as `sendfile()` in Linux with the file opened at the NIC embedded system. The application can still call an existing socket API like `write()` to send out any custom data (e.g., HTTP response headers), or it can send the content from multiple files opened by the NIC stack. Figure 5 illustrates a subset of these operations with the API functions in the context of an HTTP server.

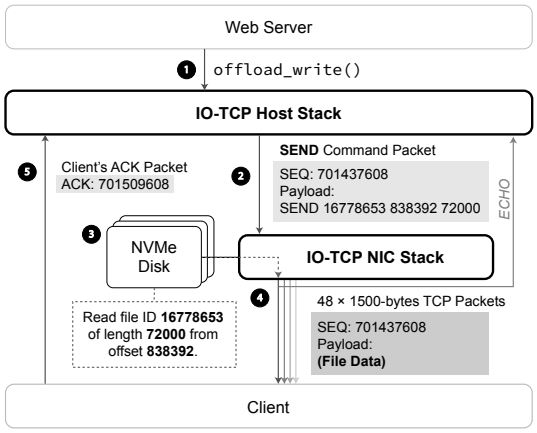


Figure 6: Generation of data packets with `offload_write()`

3.3 IO-TCP Host Stack

The role of IO-TCP host stack is to provide the full TCP functionality to applications while it interacts with the NIC stack to offload the data plane operations. The key challenge in the host stack design is how to create data packets with "missing" file data. Similarly, it should handle TCP packet retransmission without actual file data in the host side.

IO-TCP addresses the challenge by *virtually* performing data plane operations on the host stack. The host stack keeps track of which data in the sequence number space is "virtual" and performs only the bookkeeping operations while it delegates the real I/O operations to the NIC stack. For example, an application can call a mix of `write()` and `offload_write()`, and the host stack writes the immediate content directly into the send buffer while it virtually fills out the buffer range for `offload_write()` by metadata update. `offload_write()` returns immediately with the number of "virtual" bytes that can be written to the send buffer.

Then, the host stack determines the send window size with its congestion and flow control parameters, and posts a "SEND" command to the NIC stack to transfer the virtual data (Refer to ③ and ④ in Figure 5). Note that any data packets with real content (written by `write()`) in the host stack are sent out directly bypassing the NIC stack. [5] The "SEND" command is carried on a TCP packet destined to the NIC stack (with an internal MAC address of the NIC). The TCP/IP headers of the command packet contain the full connection information (i.e., four connection tuples, sequence and ACK numbers for the next data packet, etc.) while its payload contains the "SEND" command that is eventually replaced by the real content before it is sent out to the client. The "SEND" command specifies a file ID, the start offset to read, and the length of the data. With this information, the NIC stack reads the file content and creates and sends real data packets with the header information. Depending on the file content size, one "SEND" command can be translated into multiple MTU-sized data packets. Figure 6 illustrates how a "SEND" command packet is processed.

The host stack handles packet retransmission in the same manner – sending a "SEND" command with the file content information for retransmission. The rationale for this design is to make the NIC stack as simple as possible. An obvious alternative is to have the NIC stack handle retransmission so that it ensures reliable delivery of whatever data is transmitted due to the "SEND" command. Then, the NIC stack must keep track of all ACKs from the client and run the congestion control logic to determine when to retransmit packets. This would make the NIC stack stateful and more complex, which would be challenging to implement efficiently on some other SmartNIC platforms (e.g., FPGA-based ones).

For all other operations, IO-TCP behaves similarly to the normal TCP stack. All complex operations such as per-connection state and buffer management on the receive path, timer management, reliable data transfer, congestion/flow control, and error control are executed on the host stack. In addition, for the control packets or packets whose data is available on the host stack, the host stack creates and sends them directly to the client bypassing the NIC stack. All incoming packets from the client get delivered directly to the host stack as well. (See ② and client-sent ACKs in Figure 5) This is not only because going through the embedded system on the NIC incurs extra latency, but it also places an unnecessary burden on the NIC stack. This packet steering can be easily enforced in the separated mode of the Mellanox BlueField-2 NIC where an embedded system on NIC has different IP and MAC addresses.

3.4 IO-TCP NIC Stack

The IO-TCP NIC stack is responsible for performing all real data plane operations for the host stack – it handles offloaded file I/O and network I/O for data packet transfer. It operates by handling custom commands from the host stack where each command is carried on a special packet destined to the NIC stack. Currently, four commands are defined: "OPEN", "CLOS", "SEND", and "ACKD". "OPEN" and "CLOS" are for fille opening or closing. "SEND" is the main command for sending the file content to the client. "ACKD" is used to efficiently handle retransmission without redundant disk access.

The "SEND" command is the key driver for I/O operations. Conceptually, it extends TCP segmentation offload (TSO) with the metadata that describes how to fill in the packet payload. Given the "SEND" command, the NIC stack checks if the target file is opened, and reads the file content into a fixed-sized memory buffer. The file read offset and its length are aligned to the NVMe disk page boundary (e.g., 4KB), and the actual file I/O is executed asynchronously to prevent blocking of the main thread. When the file content becomes available on the memory buffer, the NIC stack creates a TSO packet with the TCP/IP headers in the "SEND" command packet, and sends it out to the NIC hardware data plane. The NIC

[5] If real data has to be sent after virtual data, the host stack delays transmission until the arrival of an echo packet (§3.5) to keep the order.

hardware data plane takes care of TCP packet segmentation and TCP/IP checksum calculation.

3.5 Challenges with Integrated I/O

Combining file I/O into the network I/O in the NIC stack brings a few unique challenges in the correctness of the TCP stack operation.

Retransmission timer and RTT measurement. TCP relies on delay measurement for setting up retransmission timers. However, the delays induced by disk I/O could confuse the RTT measurement. Even with fast NVMe disks, the disk access delay for reading a few KBs of data is in the order of microseconds, and it can be up to milliseconds if the I/O requests for the same disk are backlogged. We observe that our early implementation of IO-TCP often retransmits the packets even if the original packets have not been sent out to the client.

To address this problem, we have the NIC stack send back an *echo* packet to the host stack just before transferring data packets for the corresponding "SEND" command. The host stack starts the retransmission timer only when it receives an echo packet for the SEND command. For accuracy, the host stack adds a one-way delay of the echo packet (~3 microseconds on our platform) from the NIC stack to itself to the timeout value. The CPU overhead for the echo packet is small as it is sent per "SEND" command and a typical "SEND" command is translated to tens to even hundreds of MTU-sized packets for large-file delivery.

Also, for precise RTT measurement, the NIC stack reflects the real "packet processing" delay into the TCP timestamp option value, i.e., the delay between the arrival of a "SEND" command to the NIC stack and the departure of the corresponding data packets from the NIC stack. That is, the "SEND" command packet carries the TCP timestamp option filled by the host stack, and the NIC stack updates the value before sending out the packets. As the timestamp option value is in the millisecond granularity [38] and the time feed in the host stack is on the order of microseconds, the host stack sends the extra time information in the microsecond granularity to the NIC stack. Then, the NIC stack can round up the timestamp value if necessary.

Handling retransmission. Since retransmission of I/O-offloaded packets is also implemented with the "SEND" command, a naïve implementation that re-reads the file content would waste the disk and memory bandwidth. To avoid the inefficiency, the NIC stack keeps the original data content in memory until the host stack confirms the delivery to the client. When the host stack sees the ACKs for the I/O-offloaded sequence space range, it periodically informs the NIC stack of the delivered portion with the "ACKD" command packet. Then, the NIC stack can recycle the memory buffers holding the delivered data. To minimize the overhead, the host stack informs the NIC stack whenever it sees a threshold amount of data (e.g., we use 32KB for now) acknowledged by the client from the last time. Note that this buffer memory essentially serves as the socket send buffer in the normal TCP stack, and the required memory in practice roughly corresponds to the bandwidth-delay product. A 100 Gbps NIC with 30ms of average RTT for the connections would require 375MB of the buffer memory in aggregate.

3.6 Handling Errors

In IO-TCP, the host stack is responsible for handling all TCP-level errors such as handling packet losses, malformed packets, or abrupt connection failures. Since the NIC stack only sends packets on behalf of the host stack and all incoming packets bypass the NIC stack, the host stack can reason about any TCP-level errors as other TCP stacks do.

In contrast, the NIC stack must report errors in file I/O to the host stack. For an "OPEN" command, the NIC stack responds to the host stack whether opening a file was successful or not. Then, the host stack raises an event to the corresponding file ID so that the application learns the result. Since the host stack caches the metadata for an offloaded file (see §4), it can return an error if `offload_write()` is passed wrong parameter values. In case a file read operation itself fails, it is reported to the host stack with an "Error" command packet with the file ID and the error code. Then, `offload_write()` would return −1 with the error code at `errno` next time the application calls it.

3.7 Support for TLS and QUIC

TLS is widely used in the modern Internet as QUIC [65] and HTTP/2 [20] adopt it by default. IO-TCP can support TLS similarly to kTLS [92] except that it offloads the encryption to the SmartNIC. This is feasible as many SmartNICs (including Bluefield-2) [24, 27, 31] already support AES and SHA in hardware. So, the CPU side runs the TLS handshake and sets up the encryption and hashing keys with the SmartNIC. All data in the receive path should be decrypted by the CPU stack similarly to other receive-path processing in IO-TCP. One complication lies in how to encrypt the non-offloaded data, but one can forward such packets to SmartNIC for encryption or encrypt them with CPU's AES-NI instructions. Support for TLS is still in progress as we have implemented content encryption with AES-GCM in NIC hardware and plan to support TLS handshake and TLS record structures.

The key idea of IO-TCP can be easily applied to other transport layer protocols like QUIC – Appendix A briefly explains the architecture of IO-QUIC. We plan to elaborate on the detailed design in the follow-up work.

4 Implementation

IO-TCP host stack. We implement the IO-TCP host stack by modifying mTCP [58], a high-performance user-level TCP stack. We choose mTCP as its socket API is similar to the Berkeley socket API and it supports event-driven programming with `epoll`. The host stack extends the mTCP API

functions with four offload functions (as shown in Table 3). Each offload function is implemented by exchanging special command packets with the NIC stack. The NIC stack detects a command packet by checking the special value in the ToS field in the IP packet. The "SEND" command packet has valid TCP/IP headers with the full connection information so that only the payload (as well as checksums) needs to be replaced with the real file content before being sent to the client.

For `offload_open()`, the host stack generates a unique file ID for the file path and returns it to the user. Under the hood, it attaches the file ID to the "OPEN" command to refer to the opened file on the NIC stack. As part of response, the NIC stack provides the metadata of the opened file (e.g., output of `fstat()`) so that the host stack can handle `offload_stat()` locally on its own. This should cut back the round trip to the NIC stack. For file operations, both the host and NIC stacks share the same file system – the host OS mounts the file systems on the NVMe disks as read/writable while the NIC stack mounts them as read-only. One problem is that any update on the host-side file system does not automatically propagate to the NIC stack as they run on a separate operating system. While we currently assume that the files do not change during the content delivery service, one should add support for dynamic synchronization of the two file systems in the future.

IO-TCP NIC stack. The NIC stack is implemented as a DPDK application. It operates by handling command packets from the host stack. Each Arm core runs one main thread and a few disk reading threads that are pinned to the core. The command packets are distributed to the main threads by receive-side scaling (RSS) on the NIC hardware, which ensures in-order packet delivery in the same connection.

For efficient memory buffer management, the NIC stack pre-allocates all buffers for file content at startup. Each main thread owns $1/n$ of them to avoid any lock contention, and a simple user-level memory manager allocates and frees the buffers at low cost. We implement zero-copy DMA of file data and packet header with DPDK (i.e., scatter-gather DMA), which improves the large-file delivery throughput (in the experiments for Figure 8) by 63%.

File reading, even with faster NVMe disks, is slower than memory operations, so each main thread employs a few disk reading threads to prevent blocking of the main thread. Disk reading threads use direct I/O to bypass the inefficiency in the file system cache [74], and communicate with the main thread through shared memory. An alternative is to use a user-level disk I/O library like Intel SPDK [42]. In fact, we observe that SPDK reaches the peak disk read performance with half the Arm processor cycles used by direct I/O, but we stick to a regular file system here (i.e., ext4 on Linux) as SPDK's support for file system is not mature yet.

IO-TCP TLS implementation. We modify the DPDK NIC driver to offload TLS symmetric key encryption with the BlueField-2 NIC. Our TLS module initializes the NIC with TLS offloading feature enabled, and it registers a chosen ciphersuite with the NIC and returns an ID for it. When the TLS module marks the packets with the ID, the packets are encrypted with the corresponding ciphersuite. We implement AES-GCM with 256bit keys with ConnectX-6 Dx, which supports the encryption almost at line rate.

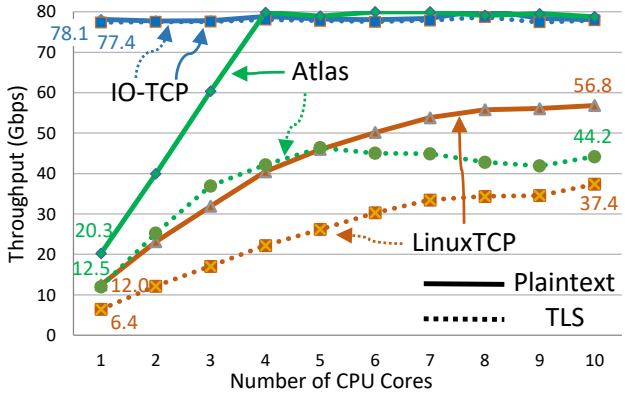


Figure 7: Comparison of throughputs of lighttpd on Linux TCP and IO-TCP, and those of the Atlas server [74] over varying number of CPU cores serving 500KB files. Dotted lines are for TLS traffic.

Porting lighttpd to IO-TCP. To evaluate the effectiveness of IO-TCP in the real-world applications, we port lighttpd v1.4.32 to IO-TCP. We obtain the mTCP-ported lighttpd code in Github [16], and have modified it to support offloaded I/O operations. Porting it to IO-TCP was straightforward as we needed to modify only about 10 out of 41,871 lines of the lighttpd code.

5 Evaluation

We evaluate IO-TCP with the following questions in mind: (1) how much performance improvement does IO-TCP bring over Linux or custom TCP stacks for content delivery systems? (2) does it result in significant CPU cycle saving? and (3) do our design choices (Retransmission timer and RTT measurement correction) serve their purposes well? Before running the experiments, we first verified the correctness (integrity of transferred files) of the IO-TCP stack with the IO-TCP-ported lighttpd server even in the case of many packet losses and multiple concurrent connections.

5.1 Experiment Setup

Our experiment setup consists of one server and two client nodes. The server machine has two Intel Xeon Silver 4210 CPU @ 2.20 GHz (20 cores) [6], with two 100G Mellanox BlueField-2 SmartNICs and four Intel Optane 900P NVMe SSDs. We attach one SmartNIC with two NVMe disks using NVMe-oF target offload so that the NIC can access the two NVMe disks directly. The host CPU runs on Linux 4.14 while the SmartNIC runs on Linux 4.20. The client machines are each equipped with an Intel E5-2683v4 CPU @ 2.10GHz (16

[6] We use only one CPU (i.e., 10 cores) for experiments.

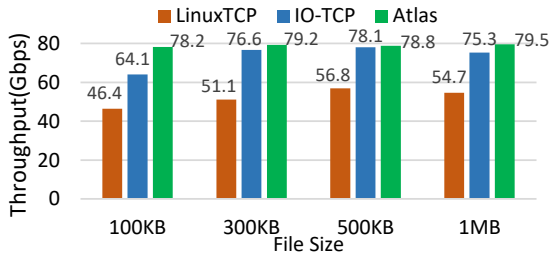
cores) and a 100G Mellanox ConnectX-5 NIC. All clients run on Linux 4.20, and we confirm that the clients are not the bottleneck for the experiments. All NICs are connected to a 100 Gbps Dell EMC Networking Z9100-ON switch.

We populate the NVMe disks with 100KB, 300KB, 500KB and 1MB files, which represent the video chunks of different quality [74, 89]. We make sure that the workload is disk-bound so that the working set size exceeds the main memory size. Each disk has an advertised read throughput of 2500 MB/sec, which would imply that we have a theoretical limit of 80 Gbps when reading from our four disks.

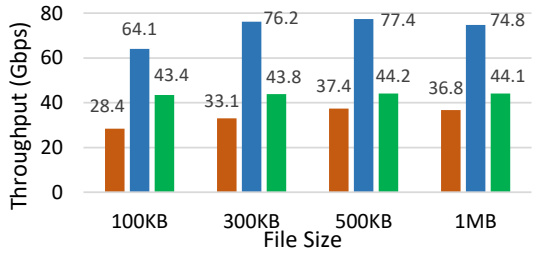
5.2 IO-TCP Throughput

We evaluate the effectiveness of IO-TCP in the large-file content delivery. We compare the throughput of IO-TCP-ported lighttpd and that of the stock version with `sendfile()` over varying numbers of CPU cores. We also compare against the Atlas server of Disk|Crypt|Net [74] that runs on FreeBSD 1.10. The Atlas server integrates raw disk reading into large-file transfer over a user-level TCP stack. For Atlas, we use a dual-port Chelsio 100Gbps NIC (T-62100) as FreeBSD 1.10 does not support the netmap [86] driver for BF-2. We have added support for TSO to the NIC driver. We note that it is not an apples-to-apples comparison as the current implementation of the Atlas server deviates from the correct operation of a typical Web server – the current version does not support regular file systems, so it simply returns a random content whose HTTP response headers are also hard-coded into NVMe disks. While implementing a proper custom file system should fix the problem, the current version benefits from avoiding the overhead. Nevertheless, comparing with Atlas would give us the rough idea of how well an IO-TCP-ported server fares over the state-of-the-art CPU-based server. Clients run `wrk` [43] to concurrently request on 1600 persistent connections. For testing Atlas, we reduce the number of concurrent connections to 800 for plaintext transfer as its custom TCP stack becomes unstable at high concurrency.

Comparison with Linux TCP and Disk|Crypt|Net. Figure 7 shows the results for serving 500KB files. lighttpd on IO-TCP achieves 78.1 Gbps with a single CPU core on the host side for plaintext transfer, which demonstrates that a single CPU core is sufficient to handle the control plane operations for all 1600 clients. IO-TCP saturates the full bandwidth of the four NVMe disks, and each NIC reaches 39 Gbps, indicating that the performance scales to the number of NICs. In contrast, Linux TCP does not go beyond 57 Gbps even with 10 CPU cores. Even when we use both CPUs (i.e., 20 cores), we do not see performance improvement (56.2 Gbps). This shows that the memory bandwidth is inefficiently utilized [74] despite the usage of a zero-copy API like `sendfile()`. When `lighttpd` on each CPU runs with a distinct port and serves a disjoint set of files, the performance goes up slightly (59 Gbps) as it benefits from local memory bandwidth. However, the improvement is limited because the content often has to



(a) Plaintext



(b) TLS

Figure 8: Comparison of maximum performance of lighttpd on LinuxTCP, and IO-TCP and Atlas for varying file sizes.

cross the NUMA domain to a NIC and the placement of kernel objects are not NUMA-aware. On the other hand, Atlas performs much better, reaching the same performance of IO-TCP at four CPU cores. When we add more NVMe disks (up to 8) and use two CPUs with both NIC ports, the performance of Atlas peaks at 107 Gbps where the memory bandwidth becomes a bottleneck. The performance of IO-TCP goes up to 95.2 Gbps if IO-TCP serves a random content with raw disk access like in Atlas, but the bottleneck lies in the memory bandwidth of the BF-2 NIC. This shows that efficient use of the host memory bandwidth is highly effective in achieving a very good throughput with only CPU. However, the performance advantage disappears when serving TLS traffic where the memory bandwidth becomes a bottleneck much earlier (discussed in the next paragraph). Figure 8a compares the performances with different file sizes. All performances of Atlas are similar as it avoids calling filesystem APIs. The performance of IO-TCP is comparable to those of Atlas from 300 KB files. IO-TCP outperforms Linux TCP by 38% to 51% and it uses 2x to 10x smaller number of CPU cores to reach the peak performance.

TLS performance. IO-TCP excels at serving TLS traffic. We enable packet encryption with AES-GCM with 256bit keys on the NIC crypto hardware for IO-TCP. For stock lighttpd, we use OpenSSL 1.0.2k [34] with TLSv1.2, and use the same algorithm for symmetric key encryption. Both Atlas and the IO-TCP-ported lighttpd do not implement the TLS handshake, but the overhead for the handshake with stock lighttpd is negligible as we use persistent connections. Figure 7 and Figure 8b show that IO-TCP experiences little performance degradation with TLS due to the dedicated crypto hardware on NIC. In contrast, Atlas achieves only 44.2 Gbps even with

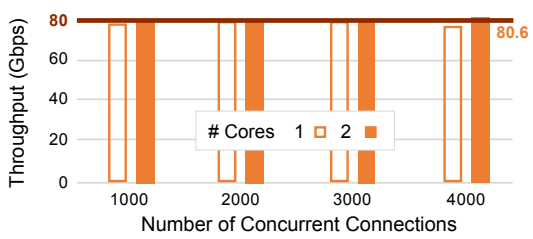
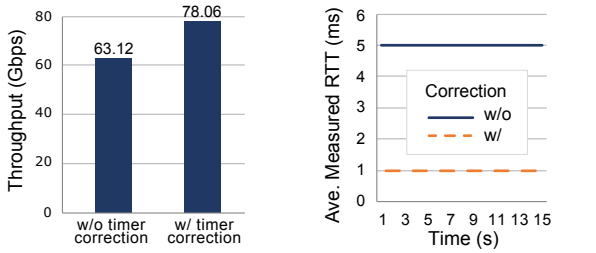


Figure 9: Comparison of TLS performance of IO-TCP over different number of connections using 1 and 2 CPU cores.



(a) Correcting retransmission timers.

(b) Correcting timestamps for more accurate RTT estimates.

Figure 10: Time measurement correction in IO-TCP

10 CPU cores as the main memory bandwidth becomes the bottleneck. The performance goes up to 54.6 Gbps with two CPUs, but overall, the TLS performance is 51% to 56% of the plaintext throughput. The similar trend is seen with Linux TCP - the TLS performance drops by 48% to 63%. We confirm that both Atlas and Linux TCP benefit from the AES-NI instructions of the CPU, but their TLS performances are poor due to content scanning for encryption. We note that the working set size of Atlas exceeds the CPU cache, so their TLS performance is bottlenecked by the memory bandwidth much faster. So, the claim that Disk|Crypt|Net manages the workload within the CPU cache depends on the hardware.

Connection scaling. Figure 9 shows the performance of IO-TCP serving 1000 to 4000 concurrent TLS connections. We observe that the performance is more or less stable over different number of connections. For 4000 connections, IO-TCP loses about 5% of performance with a single CPU core, but it reaches 80 Gbps again with two CPU cores. We check that the plaintext performance exhibits comparable trends. At 4000 connections, each connection would get around 20 Mbps, a comfortable bandwidth to stream 4K videos.

Comparison with user-level TCP stacks. One might be tempted to compare the performance with recent user-level TCP stacks like mTCP [58], IX [49], TAS [62], and F-Stack [13] as they use the CPU cycles efficiently. However, we find that these stacks are not optimized for large-file content delivery as most of them do not implement `sendfile()` nor benefit from TSO. In fact, we measure the performance of TAS, mTCP, and F-Stack on the same platform with all 10 cores, but they showe 8, 21.4, and 36 Gbps, respectively, for 500KB file delivery. Even if they implement a zero-copy API, we doubt that it would substantially outperform Linux TCP because the primary goal of the kernel-bypass networking stacks is to avoid the overhead of frequent system calls and kernel data structures for small-message transactions. However, transferring large messages would rarely impose the system call overhead nor suffer from the overhead of kernel structures. Instead, insufficient memory bandwidth (or CPU cycles) is the main cause for poor performance in large-file content delivery, which kernel-bypass TCP stacks do not help.

TCP fairness. We also evaluate if IO-TCP provides bandwidth fairness among the competing connections. Jain's Fairness Index of IO-TCP ranges from 0.91 to 0.97 for different numbers of concurrent connections. We see a similar range (0.90 to 0.97) with Linux TCP for the same experiments.

5.3 Evaluation of IO-TCP Design Choices

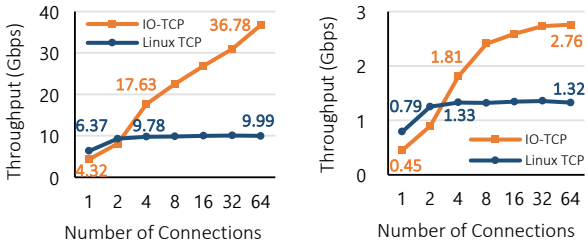
Retransmission timer correction. We evaluate the impact of echo packets that adjust the retransmission timer – when the real data packets are sent out. Figure 10a compares the throughput of lighttpd on IO-TCP with and without timer correction. Without timer correction, the IO-TCP host stack stays at 63.12 Gbps. With timer correction, IO-TCP improves the throughput by 22.6%. This is because IO-TCP without timer correction experiences highly variable RTTs and produces 600x more timeouts than that with timer correction. Such performance drop due to premature timeouts can be more severe in the wide-area-networks (WANs) where the end-to-end RTTs are larger.

RTT measurement correction. We compare the impact of fixing the TCP timestamps on the NIC stack. We measure the average RTT values recorded by the TCP stack every second with 200 concurrent connections. Figure 10b shows that the average RTT is 1 ms with timestamp correction. When we disable the TCP timestamp correction, the average RTT reaches 5 ms, a blowup by a factor of 5. This is because the RTT includes disk access delay that adds a few milliseconds. More accurate latency measurement is critical to trigger the timeout in time when there is a packet loss.

5.4 Overhead Evaluation

The split architecture of IO-TCP may suffer from the communication overhead between host and NIC stacks as well as lower computing capacity of the Arm-based subsystem in the NIC. For this reason, the CPU-only approach on Linux TCP would perform better than IO-TCP for a small number of concurrent connections as CPU can comfortably handle the connections without the overhead. However, this trend will change as the number of connections increases.

Figure 11a shows the throughputs over different numbers of concurrent connections requesting 300KB files. With a single persistent connection, Linux TCP outperforms IO-TCP by over 1.5 times. However, it reaches the peak performance with as few as four connections and the performance stays the same beyond that. In contrast, the throughput of IO-TCP



(a) Throughputs over varying # of connections w/ 300KB files **(b)** Throughputs over varying # of connections w/ 10KB files

Figure 11: Overhead Evaluation.

slowly increases due to the overhead, but it outperforms the Linux TCP at four connections.

The performance trend continues to hold with smaller file sizes. Figure 11b shows the throughput for serving 10KB files over different numbers of connections. Like in the previous case, Linux TCP and IO-TCP reach the peak performance at 4 and 64 connections, respectively, but their performance is much lower than in Figure 11a due to the increased overhead of file operations. Nevertheless, IO-TCP outperforms Linux TCP at four or more concurrent connections.

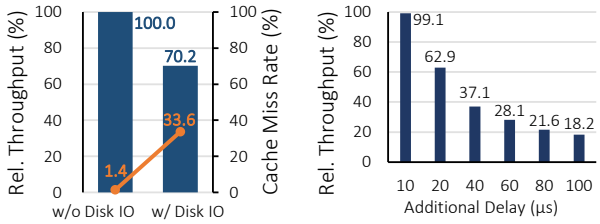
To evaluate the latency overhead, we compare the latency of single-file downloading with IO-TCP vs. Linux TCP. IO-TCP shows 50 to 80us of an extra delay for 10KB files, but the extra overhead goes up to 150 to 200us for 300KB files. This latency overhead at low congestion is inevitable as the host CPU is much faster than the Arm processor in the NIC, but it is negligible for content delivery in wide-area networks.

Memory bandwidth limitation. The performance bottleneck of our current prototype lies in the low memory bandwidth of the BlueField-2 NIC when it accesses more than two NVMe disks. We confirm this by running the same test as in Figure 8a without disk I/O – we observe that the performance reaches 80 Gbps per NIC. Note that disk I/O is the only memory copy for packet payload in our NIC stack as we employ scatter-gather DMA. Nevertheless, we still think our design is promising in the future. First, Arm-based SoC can be designed with much higher memory bandwidth and future SmartNICs would benefit from it. For example, Bluefield-3 [32] is reported to have 3.5x better memory bandwidth (~90 GB/s) than Bluefield-2, and we expect over 100 Gbps per NIC [7] for the same workload as in Figure 8a. Cavium ThunderX2 [8], an Armv8-based SoC server, has 166 GB/s of peak memory bandwidth, even larger than that of our CPU. Second, improving the memory bandwidth of the SmartNIC is more cost-effective as Arm SoCs are less expensive than server-class x86 CPUs [70] and one can easily scale the overall performance by employing multiple NICs. We note that the current SmartNIC price is very high, but it will go down with wider adoption as evidenced in the GPU prices.

[7] 40 Gbps (per NIC) × 90 GB/s (BF-3 BW) / 25.6 GB/s (BF-2 BW) = 141 Gbps

Functions	Instr. per cycle	
	Before	After
APP: Parsing HTTP request	0.82	1.57
APP: Writing response header	0.92	1.89
TCP: Check RTO expire	2.65	3.26
TCP: Process ACK	0.13	1.24
Overall IPC	0.93	1.47

Table 4: Comparison of the IPC of lighttpd and control plane functions before and after data plane offloading with IO-TCP.



(a) Relative throughputs and cache **(b)** Relative throughputs over extra miss rates with and without disk IO. delays by the control plane.

Figure 12: Analysis on the source of performance improvement.

5.5 Source of Performance Improvement

We analyze the source of performance improvement with IO-TCP. First, we observe that the control plane functions in the IO-TCP stack run faster after the separation of the data plane. Table 4 indicates that the instructions per cycle (IPC) of the control path in the IO-TCP stack improves by 58% with the division of labor. Especially, ACK processing benefits the most from the split – note that it is the key function that initiates complex operations frequently such as looking up the TCB in the connection table, determining packet loss/duplicate ACKs, computing the new send window size, etc. After the split, the IPC of this function improves by 9.53x. The performance gain mainly comes from reduced cache/memory contention as we find that the cache miss rate of last-level cache (LLC) improves by 27% with the separation. Then, how come the cache miss rate is reduced? This is because DDIO of NVMe disk IO evicts the data in the CPU cache if both planes run together [96]. To confirm this, we measure the TCB lookup performance with and without NVMe disk reading (`fio`). Figure 12a shows that the cache miss rate of the TCB lookup goes up by a factor of 24 if we co-run disk IO, which in turn reduces the lookup performance by 30%. Finally, we observe that the faster execution of the control plane actually improves the content delivery throughput. To show this, we add redundant code into the ACK processing function so that we can delay its execution by as much as we want. Figure 12b shows that the throughput degrades significantly as the extra delay increases. This implies that the faster control plane reduces the end-to-end RTT and increases the send window size quickly, which ends up improving the overall performance.

6 Related Work

PCIe P2P communications. Enabling PCIe P2P communication between external devices could reduce CPU overhead significantly when transmitting data among them. NVIDIA GPUDirect RDMA [17], GPUDirect Async [45], and AMD DirectGMA [10] techniques, provide a way for other devices to directly access data from GPU by exposing GPU memory directly to PCIe memory space. EXTOLL [79] proposes enabling direct communication between Intel Xeon Phi coprocessors (accelerators) and the NICs, so accelerators can communicate with each other over the network without CPU involvement. Morpheus [91] enables communications between NVMe devices and other PCIe devices. DCS [46] and DCS-ctrl [63] propose a hardware-based framework to enable P2P communication among various types of external PCIe devices. However, all these P2P solutions only consider data communication on hardware, without considering the kernel stacks. As a result, those solutions still suffer from kernel stack overhead when running content delivery applications.

Accelerated networking stacks. There are several existing works that attempt to improve the performance of networking stacks. Some works try to improve the performance of existing kernel stacks. Fastsocket [71] improves the TCP stack performance by achieving table-level connection partition, increasing the connection locality, and eliminating the lock contention. StackMap [95] dedicates network interfaces to applications and offer a zero-copy, low-overhead network interface for applications. Megapipe [57] leverages partitioned, lightweighted sockets, and batches system calls to improve the performance. Another approach is bypassing the heavyweight kernel stack and running the whole stack in the user level. mTCP [58], IX [49], Sandstorm [73], F-Stack [13], and PonyExpress/Snap [75] leverage user-level packet I/O libraries, and leverage multiple CPU cores to process incoming flows simultaneously, in order to increase the processing throughput and reduce latency from kernel calls. ZygOS [85], Shinjuku [60], and Shenango [80] further improve the tail latency of packet processing by improving the load balancing of the tasks among CPU cores. Arrakis [82] and IO-TCP share the same vision of separation of data and control planes, but Arrakis is focused on bypassing the kernel involvement on the data path while IO-TCP harnesses extra processors for work division of the TCP stack operations. TAS [62] builds a TCP fast path as a separated OS service, which targets to improve the performance of RPC calls in the data center. Disk|Crypt|Net [74] builds a scalable video streaming stack, containing a novel kernel-bypass storage stack and an existing kernel-bypass network stack, which achieves lower latency and higher throughput for video streaming applications. However, all these solutions still require huge CPU involvements in packet processing, which still consumes a lot of CPU power on transmitting data among external devices. A recent work called AccelTCP [78] offloads TCP connection management as well as connection relaying into SmartNIC, which relieves a part of packet processing computation from host CPU cores. However, it focuses only on improving the throughput for short-lived connections and L7 proxies.

NIC offload. Traditionally, there have been a spectrum of NIC offload schemes. Stateless schemes like TCP/IP checksum offload, TCP segmentation offload (TSO) and large receive offload (LRO) have become ubiquitous in modern NICs while stateful schemes like TCP Engine Offload (TOE) and Microsoft Chimney Offload [25] have largely been deprecated due to security and maintenance concerns coming from its complexity. IO-TCP is essentially TSO with file reading, and we believe it can be easily implemented into commodity NIC hardware due to its simplicity.

More recently, several works have focused on offloading various tasks to SmartNICs to improve the performance for specific applications. KV-Direct [68] leverages FPGA-based SmartNIC to improve the performance of in-memory key-value stores. Floem [83], ClickNP [69], and UNO [66] leverage SmartNICs to accelerate general packet processing for network applications. Metron [61] offloads packet tagging into the NICs to reduce the latency of packet processing for network functions. iPipe [72] builds a general framework for offloading distributed applications into SmartNICs. Lynx [90] uses the SmartNIC as part of an accelerator-centric architecture where the SmartNIC allows direct networking with the accelerators. Gimbal [76] uses SmartNIC as the traffic orchestrator for disk IO, and realizes efficient multi-tenancy using congestion control algorithms and fair scheduling. LeapIO [70] offloads disk IOs to SmartNIC and provides the seamless address space for cloud tenants while [84] handles NVMe-oF on NIC for remote storage access. However, neither supports TCP operations to clients from NIC. To the best of our knowledge, our IO-TCP is the first work that leverages SmartNICs to accelerate disk and packet I/O for content delivery systems.

7 Conclusion

In this paper, we have presented IO-TCP, a split TCP stack design that offloads I/O operations from CPU for scalable content delivery. IO-TCP provides a new abstraction that leverages SmartNIC processors to perform I/O operations, which significantly relieves the pressure on CPU and its main memory system. Also, our proposal maintains the simplicity in the NIC stack design so that it can be easily implemented with low-powered processors on I/O devices.

Our evaluation shows IO-TCP significantly saves CPU cycles while it delivers the benefit even for small-file transfer when it serves enough connections. Along with the benefit, we also discuss the limitations of the current prototype, and we hope that SmartNIC vendors will consider higher memory bandwidth for the embedded system when designing the next version of their SmartNIC. The source code of IO-TCP is available at https://iotcp.kaist.edu/

Acknowledgements

We appreciate the insightful feedback and suggestions from USENIX NSDI 2022 reviewers on revising our original submission. We thank Ilias Marinos for sharing the source of Disk|Crypt|Net and for helping us with setting up the Atlas server. This work is in part supported by the ICT Research and Development Program of MSIP/IITP, Korea, under [2018-0-00693, Development of an ultra low-latency user-level transfer protocol]. Junzhi Gong and Minlan Yu are supported in part by the NSF CNS-1955422 and CNS-1955487.

References

[1] Akamai braces for huge streaming audiences in 2021. https://www.fiercevideo.com/tech/akamai-braces-for-huge-streaming-audiences-2021. Last Accessed: 2021-09-15.

[2] Akamai Technologies, Inc. https://www.akamai.com/. Last Accessed: 2021-09-15.

[3] Amazon Prime Video. https://www.primevideo.com/. Last Accessed: 2021-09-15.

[4] Apple TV+. https://tv.apple.com/. Last Accessed: 2022-08-23.

[5] As Covid pushes more people online, companies that help the web stay speedy are having a moment. https://www.cnbc.com/2020/12/13/cdn-providers-cloudflare-fastly-benefit-from-covid-web-traffic-boost.html. Last Accessed: 2021-09-15.

[6] Athlon 64 X2. https://en.wikipedia.org/wiki/Athlon_64_X2. Last Accessed: 2021-09-15.

[7] Broadcom Stringray SmartNIC. https://www.broadcom.com/products/ethernet-connectivity/smartnic. Last Accessed: 2021-09-15.

[8] Cavium ThunderX2 Arm-based Processors. https://www.marvell.com/products/server-processors/thunderx2-arm-processors.html. Last Accessed: 2021-09-15.

[9] Cisco Visual Networking Index 2021. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf. Last Accessed: 2021-09-15.

[10] DirectGMA on AMD's FirePro GPUs. https://www.amd.com/Documents/SDI-techbrief.pdf.

[11] Disney+. https://www.disneyplus.com/. Last Accessed: 2021-09-15.

[12] DPDK. https://www.dpdk.org. Last Accessed: 2021-09-15.

[13] F-Stack | High Performance Network Framework Based on DPDK. https://github.com/F-Stack/f-stack. Last Accessed: 2021-09-15.

[14] Fastly, Inc. https://www.fastly.com/. Last Accessed: 2021-09-15.

[15] fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html. Last Accessed: 2021-09-15.

[16] GitHub - mtcp-stack/mtcp. https://github.com/mtcp-stack/mtcp. Last Accessed: 2021-09-15.

[17] GPUDirect. https://developer.nvidia.com/gpudirect. Last Accessed: 2021-09-15.

[18] HowTo Configure NVMe over Fabrics (NVMe-oF) Target Offload. https://community.mellanox.com/s/article/howto-configure-nvme-over-fabrics--nvme-of--target-offload. Last Accessed: 2021-09-15.

[19] Hulu: Stream TV and Movies Live and Online. https://www.hulu.com/. Last Accessed: 2021-09-15.

[20] IETF RFC 7540. https://tools.ietf.org/html/rfc7540. Last Accessed: 2021-09-15.

[21] Intel Direct Data I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. Last Accessed: 2021-09-15.

[22] Intel® Xeon® Platinum 9282 Processor. https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html. Last Accessed: 2021-09-15.

[23] Lighttpd - fly light. https://www.lighttpd.net/. Last Accessed: 2021-09-15.

[24] Marvell LiquidIOII Smart NICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics.html. Last Accessed: 2021-09-15.

[25] Microsoft Windows Scalable Networking Initiative. http://download.microsoft.com/download/5/b/5/5b5bec17-ea71-4653-9539-204a672f11cf/scale.doc. Last Accessed: 2021-09-15.

[26] Netflix - Unlimited movies, TV shows, and more. https://www.netflix.com/. Last Accessed: 2021-09-15.

[27] Netronome Agilio LX SmartNICs. https://www.netronome.com/products/agilio-lx/. Last Accessed: 2021-09-15.

[28] NGD Newport NVMe Computational Storage Drive. https://www.ngdsystems.com. Last Accessed: 2021-09-15.

[29] nginx. http://nginx.org/. Last Accessed: 2021-09-15.

[30] NGINX and Netflix Contribute New sendfile(2) to FreeBSD. https://www.nginx.com/blog/nginx-and-netflix-contribute-new-sendfile2-to-freebsd/. Last Accessed: 2021-09-15.

[31] NVIDIA BlueField-2 Programmable SmartNIC. https://www.mellanox.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf. Last Accessed: 2021-09-15.

[32] NVIDIA BlueField-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf. Last Accessed: 2021-09-15.

[33] NVIDIA BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf. Last Accessed: 2021-09-15.

[34] OpenSSL. https://www.openssl.org/. Last Accessed: 2021-09-15.

[35] PCI Express Base Specification. https://pcisig.com/specifications. Last Accessed: 2021-09-15.

[36] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Last Accessed: 2021-09-15.

[37] Poky – Yocto Project. https://www.yoctoproject.org/software-item/poky/. Last Accessed: 2021-09-15.

[38] RFC 7323. https://tools.ietf.org/html/rfc7323. Last Accessed: 2021-09-15.

[39] Sandvine Global Internet Phenomena Report COVID-19 Spotlight. https://www.sandvine.com/phenomena. Last Accessed: 2021-09-15.

[40] SmartSSD Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd/index.html. Last Accessed: 2021-09-15.

[41] StoPool Distributed Storage. https://storpool.com/blog/7-million-iops-and-0-15-ms-latency-for-an-nvme-powered-vdi-cloud. Last Accessed: 2021-09-15.

[42] Storage Performance Developement Kit. https://spdk.io/. Last Accessed: 2021-09-15.

[43] wg/wrk - Modern HTTP benchmarking tool. https://github.com/wg/wrk. Last Accessed: 2021-09-15.

[44] YouTube TV - Watch and DVR Live Sports, Shows & News. https://tv.youtube.com/. Last Accessed: 2021-09-15.

[45] Elena Agostini, Davide Rossetti, and Sreeram Potluri. Offloading communication control logic in GPU accelerated applications. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[46] J. Ahn, D. Kwon, Y. Kim, M. Ajdari, J. Lee, and J. Kim. DCS: A Fast and Scalable Device-centric Server Architecture. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[47] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[48] G. Banga and J.C. Mogul. Scalable Kernel Performance for Internet Servers under Realistic Loads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1998.

[49] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[50] Steve Blank. What the GlobalFoundries' Retreat Really Means. https://spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means, 2018. Last Accessed: 2021-09-15.

[51] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[52] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS

User-Space NVM File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[53] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[54] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[55] Ethernet Alliance. The 2020 Ethernet Roadmap. https://ethernetalliance.org/technology/2020-roadmap/, 2020. Last Accessed: 2021-09-15.

[56] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. A Year in Lockdown: How the Waves of COVID-19 Impact Internet Traffic. *Communications of the ACM (CACM)*, 64(7):101–108, 2021.

[57] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[58] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[59] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[60] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[61] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[62] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[63] D. Kwon, J. Ahn, D. Chae, M. Ajdari, J. Lee, S. Bae, Y. Kim, and J. Kim. DCS-ctrl: A Fast and Flexible Device-Control Mechanism for Device-Centric Server Architecture. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[64] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[65] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, and J. Iyengar. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[66] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2017.

[67] Jonathan Lemon. KQueue–A Generic and Scalable Event Notification Facility. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2001.

[68] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[69] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[70] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan Ports, Irene Zhang,

Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[71] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. *ACM SIGARCH Computer Architecture News*, 44(2):339–352, 2016.

[72] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019.

[73] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[74] Ilias Marinos, Robert NM Watson, Mark Handley, and Randall R Stewart. Disk|Crypt|Net: rethinking the stack for high-performance video streaming. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[75] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[76] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 106–122, 2021.

[77] G. Banga J.C. Mogul and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1999.

[78] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[79] Sarah Neuwirth, Dirk Frey, Mondrian Nuessle, and Ulrich Bruening. Scalable communication architecture for network-attached accelerators. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[80] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[81] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1999.

[82] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2013.

[83] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for NIC-accelerated network applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[84] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC Offloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[85] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[86] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[87] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015.

[88] David Sidler, Zsolt Istvan, and Gustavo Alonso. Low-Latency TCP/IP Stack for Data Center Applications. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2016.

[89] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[90] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[91] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. *ACM SIGARCH Computer Architecture News*, 44(3):53–65, 2016.

[92] Dave Watson. KTLS: Linux Kernel Transport Layer Security. *Proposal by Facebook Engineer*, 2016.

[93] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[94] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[95] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[96] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *Proceedings of the 48th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2021.

[97] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable Parallel Flash Firmware for Many-core Architectures. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[98] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

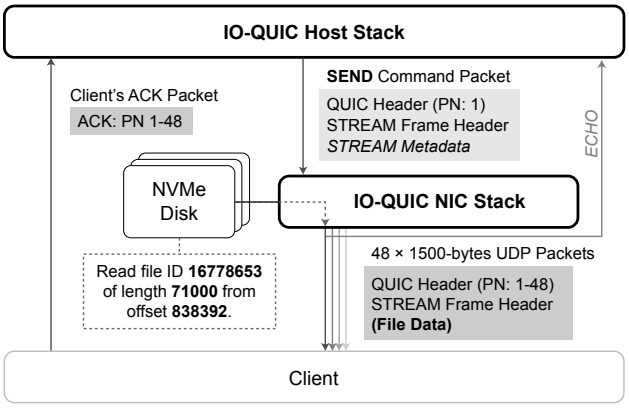


Figure 13: An adaptation of IO-TCP to QUIC.

Appendix

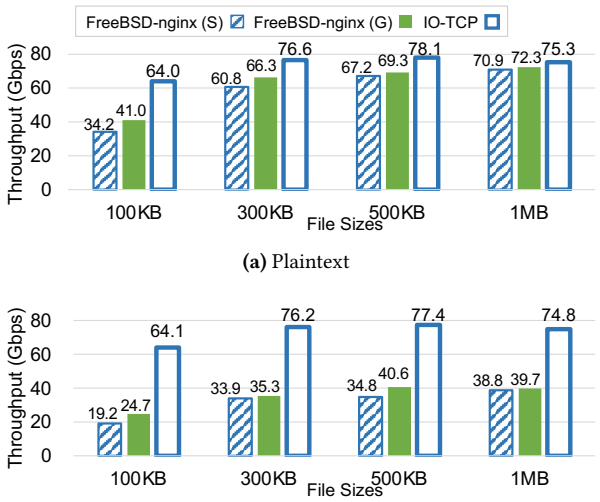
A Support for the QUIC protocol

Unlike TCP/IP headers, the QUIC header is variable-sized, so the host stack should carefully estimate the header size before offloading. Since the data length can exceed an MTU size, the SmartNIC should perform QUIC packet segmentation with generation of QUIC headers as well as UDP/IP headers. In addition to file IO offloading, UDP packet segmentation with large content on SmartNIC could improve the performance further as QUIC on the Linux UDP stack suffers from frequent context switchings for invoking a system call for each UDP packet. For reliable transfer, the host stack should keep track of STREAM frame packet numbers and data offsets that are sent out. Retransmissions can be handled similarly to IO-TCP as the IO-TCP NIC stack manages the file content buffers independently of the particular transport layer protocol. Likewise, the "ACKD" command can free the file content buffers that are confirmed to be delivered to the QUIC client.

We have finished implementing "IO-QUIC" in the plaintext version, and we will add support for TLS in the future. Unlike the IO-TCP implementation, the host stack of our IO-QUIC implementation uses unmodified Linux kernel as it communicates with the NIC stack with a special UDP packet.

B Performance Comparison with Asynchronous `sendfile()` on FreeBSD

Recent FreeBSD supports asynchronous `sendfile()` that does not block on disk reading [30], so we compare the performance of nginx (v1.20.1) on FreeBSD (v13.0) that utilizes this feature. Since FreeBSD does not allow disabling individual CPU cores, we use all 20 CPU cores for FreeBSD experiments (FreeBSD-nginx(S)). Also, to gauge the impact of higher-capacity CPU, we employ a different server with two Intel Xeon Gold 6142 CPUs @ 2.60 GHz, a 100G Mellanox ConnectX-5 NIC and 32GB memory of DRAM (FreeBSD-nginx(G)). Again, we use all 32 cores in the two CPUs for the experiments.



(a) Plaintext

(b) TLS

Figure 14: Comparison of maximum performance of nginx that uses asynchronous `sendfile()` on FreeBSD vs. IO-TCP. IO-TCP uses the same number as in Figure 8a and Figure 8b.

Figure 14a shows the results over different file sizes. Overall, FreeBSD achieves better performance than Linux for plaintext transfer, but it does not reach the performance reported in [74] (~70 Gbps with 8 cores). This is because the stock FreeBSD version does not support other features in [74] except asynchronous `sendfile()`. In contrast, IO-TCP outperforms all other setups despite that FreeBSD uses 19 to 31 more CPU cores. In terms of the TLS performance, Figure 14b shows that FreeBSD suffers from the same issue as Linux.

Hydra: Serialization-Free Network Ordering for Strongly Consistent Distributed Applications

Inho Choi[1], Ellis Michael[2], Yunfan Li[1], Dan R. K. Ports[3], and Jialin Li[1]

[1]*National University of Singapore,* [2]*University of Washington,* [3]*Microsoft Research*

Abstract

Many distributed systems, e.g., state machine replication and distributed databases, rely on establishing a consistent order of operations on groups of nodes in the system. Traditionally, this ordering has been established by application-level protocols like Paxos or two-phase locking. Recent work has shown significant performance improvements are attainable by making ordering a network service, but current network sequencing implementations require routing all requests through a single sequencer – leading to scalability, fault tolerance, and load balancing limitations.

Our work, Hydra, overcomes these limitations by using a *distributed* set of network sequencers to provide network ordering. Hydra leverages loosely synchronized clocks on network sequencers to establish message ordering across them, per-sequencer sequence numbers to detect message drops, and periodic timestamp messages to enforce progress when some sequencers are idle. To demonstrate the benefit of Hydra, we co-designed a state machine replication protocol and a distributed transactional system using the Hydra network primitive. Compared to serialization-based network ordering systems, Hydra shows equivalent performance improvement over traditional approaches in both applications, but with significantly higher scalability, shorter sequencer failover time, and better network-level load balancing.

1 Introduction

Replication is ubiquitous in data center applications. Consensus protocols like Paxos, Viewstamped Replication, and Raft are used to maintain multiple copies of data, providing the illusion of a single correct service that remains available even as individual replicas fail and recover. However, these protocols impose substantial latency and throughput overhead.

A recent line of work demonstrated that in-network processing can alleviate this cost [42, 43, 56]. This *network sequencing* approach routes requests through a sequencer – implemented in a programmable switch or middlebox – which assigns a monotonically increasing sequence number to each request. By pre-establishing a total order of all requests, they enable lighter weight consensus protocols, ultimately yielding impressive performance gains: Network-Ordered Paxos achieves throughput within 2% and latency within 10% of an unreplicated, non-fault-tolerant system [43].

However, employing this approach in practice is not easy. Fundamentally, the difficulty stems from the fact that *network sequencing requires serialization*: all traffic for a replicated service must pass through a single sequencer. This poses three major challenges in production networks. First, the single sequencer must process all request traffic, posing a scalability bottleneck. Second, it imposes a new routing requirement for specific application traffic, which network operators are loath to accept. Restricting path diversity interferes with existing policies, carefully engineered for load balancing and fault tolerance. Finally, it introduces an undesirable coupling between network and application-level recovery. Replacing a failed or unreachable sequencer requires coordinating a simultaneous update to the network routes and recovery of the sequencer state (via a consensus protocol). This adds deployment complexity and increases system downtime during the recovery process. All three are serious barriers to adoption, based on our experiences with large-scale production data centers.

This paper asks whether network sequencing can be achieved *without* serialization. We answer that question in the affirmative by presenting the design of Hydra,[1] a new protocol for network sequencing that allows packets to be sequenced by multiple active sequencers. Hydra's sequencers themselves run a lightweight coordination protocol, in which each sequencer independently assigns sequence numbers to requests that can be merged to establish a total order of operations. Specifically, Hydra leverages a combination of per-sequencer sequence numbers and loosely synchronized physical clocks across sequencers to assign a global ordering while still efficiently detecting dropped messages.

Hydra is a practical protocol; we have built both a software implementation that runs on end hosts and one in P4 [11] that runs on an Intel Tofino programmable switch; the latter uses only a small fraction of switch resources, demonstrating its practicality for modern network devices. Hydra's sequencing functionality allows it to run the existing NOPaxos [43] and Eris [42] replication and transaction processing protocols with minimal modification, while making them more resilient to sequencer faults with marginal performance cost.

Our evaluation results demonstrate that Hydra achieves a 378% increase in throughput and 42% reduction in latency compared to an atomic multicast baseline, while scaling to high numbers of receivers, multicast groups, and sequencers. Comparing to systems that use a network serialization approach, Hydra significantly improves network-level load bal-

[1]Hydra is named after the Lernean Hydra of Greek mythology, a multi-headed serpent that could regrow a new head if one was chopped off [31].

ancing and reduces system downtime by 5×. Moreover, Hydra achieves these benefits without sacrificing performance: our Hydra-based state machine replication system gets latency within 5 μs and throughput within 17% of NOPaxos, and our transactional system attains 47% higher throughput than Eris.

2 Background

Establishing a consistent order of operations is fundamental to many distributed systems: state machine replication [43,52,53,61] requires all correct replicas to execute a totally ordered set of client operations; distributed transactional systems [5, 14, 17, 19, 26, 39] mandate that all shards of the data store process transactions in a serializable order; distributed caches [50, 55] require consistent updates to ensure coherence.

Traditionally, guaranteeing strong consistency necessitates running complex application-level distributed protocols which involve coordination among servers. For instance, many state machine replication protocols [52, 53, 61] designate a single leader to assign an order to operations, and require it to communicate with replicas before returning a result to the client, and existing distributed databases execute concurrency control, atomic commitment, and consensus protocols for each client transaction. This expensive coordination is at odds with the demanding throughput, latency, and scalability requirements of modern data center applications.

2.1 Request Ordering in the Network

The need for these protocols stems from the fundamental assumption of a fully *asynchronous* network which can arbitrarily drop, reorder, or delay messages. A classic line of work in distributed computing proposes stronger communication primitives to simplify distributed applications, including virtual synchrony [8, 9], atomic broadcast [10, 34], and atomic multicast [28]. These provide broadcast or multicast operations that ensure all correct receivers will deliver the *same set* of messages in the *same order*. Such guarantees can obviate the need for consensus protocols – but implementing them is a problem equivalent to consensus [13], so applications do not enjoy a performance benefit.

Network ordering without reliability guarantees. A recent line of work [42, 43, 56] proposes a new network model that balances guarantees and implementation efficiency. This new model moves the responsibility of consistent message ordering into the network, but leaves reliable delivery of messages to application-level protocols. By providing ordering guarantees in the network, this network/protocol co-design approach allows faster replication protocols than traditional designs; by *not* enforcing reliability, the network model is simple enough to implement efficiently.

A key mechanism employed by these systems to implement network ordering is *in-network serialization*. For instance, Speculative Paxos [56] routes all client requests first to a designated switch in the network before multicasting to the replica servers. The single switch serves as a serialization point and ensures that, with high probability, all replicas receive client requests in the same order. NOPaxos [43] extends this serialization approach by using programmable switch ASICs to provide *guarantees* of request ordering. The designated switch stamps a sequence number into each client request. Receivers then ensure consistent ordering by processing requests only in sequence number order. Additionally, sequence numbers allow replicas to identify dropped messages (by detecting gaps in the sequence).

Eris [42] further generalizes the sequencing approach to support requests that are sent to multiple replication groups (e.g., to implement fault-tolerant distributed transactions). The sequencer switch maintains a counter for each group, and on each client request, *atomically* increments the counter value for all destination groups. These counter vectors ensure a consistent ordering of all multi-group operations, while still allowing receivers to independently detect dropped messages.

2.2 Limitations of In-Network Serialization

Prior work [42, 43, 56] has demonstrated the performance benefits of the network ordering approach and its applicability to several classes of distributed systems. However, the in-network serialization approach employed by existing network ordering solutions has important limitations.

Scalability bottleneck. A key requirement in the serialization approach is that all client requests need to go through a *single sequencer device*. This device can become a performance bottleneck. While in-switch sequencers can sustain a far higher sequencing rate than the server-based replicas that actually execute operations, sequencer capacity can still be a scalability limit for sharded database systems like Eris [42] where one sequencer serves many replica groups. Moreover, if the sequencer is implemented on an end host, as can be more practical for many deployments, poor CPU-based packet processing performance is at odds with the horizontal scaling capability of the system.

Prolonged system downtime. Being a serialization point of all client requests, a failure in the sequencer will result in unavailability of the entire distributed system. Sequencer failover is more complicated than traditional recovery (e.g., changing leaders in a Paxos deployment) because it couples network rerouting with application-level recovery. To resume operation, the network control plane must first detect the failure and carry out network-wide routing changes to redirect client traffic to a new sequencer, and afterwards begin a view change procedure to ensure system state is consistent; only then can replicas process requests from the new sequencer. Rerouting in a large data center network is expensive: previous studies [38, 42, 43] show that updating forwarding tables in a single switch alone can take more than 200 ms. Before this lengthy rerouting procedure is complete, the system will remain unavailable.

Worsened data center network properties. Data center networks are carefully engineered to provide high reliability, performance, and cost efficiency [2, 27, 49]. By adding redundant paths to the network and using protocols like ECMP, these networks can effectively load balance network traffic, tolerate link and switch failures, and sustain high bisection bandwidth. Serializing traffic through a single switch, however, reduces the number of available paths and can easily nullify these desirable network properties. For example, it can lead to link congestion at the sequencer switch.

Incompatible with multi-pipeline switches. Many switch ASICs scale out processing capacity by using multiple (e.g., 2–8 [18]) separate pipelines, with few or no shared resources. Existing sequencing approaches, however, update and atomically read a single copy of the sequence number. This requirement restricts deploying network sequencing logic to a single switch pipeline [36]. This not only limits the maximum throughput of the network sequencer to a fraction of the switch capacity, it complicates cabling and routing because specific physical ports are bound to each pipeline.

3 Sequencing with Multiple Sequencers

Hydra allows multiple active sequencers to work concurrently, preventing a single sequencer from becoming a scalability bottleneck or a single point of failure. This allows Hydra to support new deployment models for network sequencers.

3.1 Deployment Options

Hydra supports a spectrum of deployment models.

Root switches. Prior work envisioned using programmable switches at the root of a tree topology as sequencers, leveraging their centrality in the data center network. Such switches can handle high request load, making scalability beyond a single switch's capacity a less urgent concern, but they do so through the use of multiple ASICs and forwarding pipelines which prior sequencer designs do not support. Hydra can also improve availability by decoupling sequencer failover from reroute latency of the underlying network (§7.5). In addition, using multiple Hydra sequencers rather than routing all sequenced traffic through one switch provides path diversity, which allows better link-level load balancing and resilience to link failures (§7.4).

ToR switches. Many existing data center architectures cannot deploy programmable switches at the network core: they use large, multi-ASIC chassis switches at the root layer [15, 27], and programmable switches are not available in this configuration. For example, Tofino-based switches are only available in smaller 32/64-port configurations. For many scenarios, using top-of-rack switches as sequencers is thus a more practical alternative. In such deployments, scalability and fault tolerance are acute concerns: ToR switches fail more commonly [25] and frequently experience congestion on their uplinks. Hydra can avoid both problems by employing multiple sequencers (§7.4).

Sequencer appliances. In our experience, incrementally deploying new functionality in existing switches, ToR or otherwise, can be a challenge: coordinating updates with existing switch functionality and validating the correctness of a custom data plane are both obstacles. An appealing alternate approach is to employ a cluster of switches as dedicated "sequencer appliances" attached to the network as edge devices rather than being part of the fabric [57], as proposed for other network function accelerators [37, 64]. Again, fault tolerance of individual sequencers and congestion on their network links (which may not exploit the full bandwidth of the switch) are major concerns, which Hydra can alleviate.

End hosts. A final approach eschews specialized hardware in favor of using end hosts as sequencers [43]. This offers obvious deployment benefits and may be the only practical approach for many environments. However, both scalability and fault tolerance are critical here: Eris's end-host sequencer barely handles the load of a 15-shard database [42], making it an option only for smaller deployments. Hydra's multiple-sequencer approach allows it to go beyond this limit, providing a practical, scalable approach for environments where specialized hardware is unavailable (§7.1.3).

3.2 Addressing and Routing

Regardless of deployment options, Hydra integrates easily with existing data center routing structures. Each Hydra deployment has a unique IP address. Each sequencer in the deployment advertises its IP address via BGP anycast, allowing routes to be dynamically updated as sequencers join or leave the deployment. Messages are routed to individual sequencers using traditional shortest-path routing and load balancing techniques, e.g., ECMP. Alternatively, in an SDN-oriented design with a centralized controller, the network controller can install appropriate anycast routes for the group of sequencers.

Apart from these routing changes, Hydra does not require any changes to any other elements in the network besides the sequencers themselves. This is a key design constraint, and one that differentiates Hydra from other ordering approaches like 1Pipe [41], which exchanges timestamps between every switch, as well as complementary techniques like RDMA, which requires complex in-network flow control [29].

4 Hydra: Serialization-Free Network Ordering

4.1 High-Level Abstraction

The core abstraction provided by Hydra is a group communication protocol. A Hydra deployment consists of receiver *groups*, and each group contains one or more *receivers*. Hydra offers a *groupcast* primitive, where a sender specifies one or multiple groups as the destination, and the message is multicast to the receivers in the destination groups. The Hydra groupcast primitive provides the following properties to the

participants:

- **Partial Ordering.** Hydra groupcast messages are partially ordered (the partial order relation is denoted as $\prec$) – all groupcast messages with overlapping destination groups are comparable. If groupcast message m_1 is ordered before m_2 ($m_1 \prec m_2$) and a receiver receives both m_1 and m_2, then every receiver delivers m_1 before m_2.
- **Unreliable Delivery.** Hydra only offers best effort message delivery. A groupcast message is not guaranteed to be delivered to any of its recipients.
- **Drop Detection.** If a groupcast message is not delivered to all its recipients, the primitive will notify the remaining receivers by delivering a DROP-NOTIFICATION. More formally, let R be the set of receiver groups for message m, then either one of the following two conditions holds: **all** receiver groups in R deliver m or a DROP-NOTIFICATION for m, or **none** of the receiver groups in R delivers m or a DROP-NOTIFICATION for m.

These are the same guarantees provided by the network abstractions in NOPaxos and Eris [42, 43]. However, critically, Hydra allows scalability, fast failure recovery, and load balancing across sequencers, where previous designs fell short.

4.2 Prior Approach: Centralized Sequencer

A recent line of work [6, 7, 42, 43, 63] proposed to use dedicated devices in the network – a programmable switch, a network processor, or an end-host server – as a centralized sequencer to establish message ordering. In particular, Eris [42] builds a *multi-sequenced groupcast* primitive that provides the same set of guarantees as we specified in §4.1.

To implement multi-sequenced groupcast, a centralized sequencer maintains a *sequence number* for each group in the system. Senders of a groupcast message encode all recipient groups in a special packet header, and the packet is first routed to the sequencer. Upon receiving a groupcast packet, the sequencer atomically increments the sequence number for each recipient group, and writes a *multi-stamp* into the packet. The multi-stamp contains a set of $\langle$*group-id*, *sequence-num*$\rangle$, one for each recipient group. The groupcast packet is then forwarded to each receiver in each receiver group.

Groupcast receivers track the next sequence number they expect from the sequencer. When a receiver receives a groupcast packet, it checks the sequence number that corresponds to its group ID in the multi-stamp. The receiver rejects the packet if the sequence number is lower than the expected value (indicating out-of-order or duplicated messages), and delivers a DROP-NOTIFICATION to the application if the sequence number is higher than expected.

Multi-sequencing provides the three properties of §4.1. By incrementing sequence numbers *atomically*, the sequencer ensures that if two groupcast messages have overlapping groups, all receivers in those groups will deliver the two messages in a consistent order. By maintaining per-group sequence numbers, any packet loss from the sequencer to a receiver will result in a gap in the received sequence numbers, and hence a DROP-NOTIFICATION. However, using a centralized sequencer introduces the limitations previously described.

Algorithm 1 SequencerHandlePacket(pkt)

```
   id: sequencer ID
   N: total number of Hydra groups
   clk: switch physical clock
   seq[N]: sequence number for each group
1: pkt.id ← id
2: pkt.c ← clk
3: for grp in pkt.grps do
4:     pkt.seq[grp] ← ++seq[grp]
5: end for
6: Forward pkt
```

4.3 Consistent Ordering with Multiple Sequencers

Naïvely applying multi-sequenced groupcast to a multi-sequencer deployment violates the guarantees listed in §4.1. Suppose each sequencer independently maintains sequence numbers for each receiver group, and groupcast messages can be forwarded to any of the sequencers. Consider two groupcast messages m_1 and m_2, both destined to group G_1, but routed through two sequencers. The two sequencers may write the same sequence number (since they maintain sequence numbers independently) for G_1 into m_1 and m_2. When receivers in G_1 receive m_1 and m_2, they cannot consistently order the messages while providing drop detection. Breaking ordering ties with sequencer ID, for example, would be consistent across receivers, but a receiver that only received the "larger" of m_1 and m_2 would have no way of inferring the existence of the "smaller."

To enforce all the guarantees in §4.1 while scaling to multiple sequencers, we propose a new approach by combining *loosely synchronized clocks* across sequencers and *per-sequencer sequence numbers* to establish consistent ordering and detect drops, and using *periodic flush messages* to ensure receiver progress.

4.3.1 Physical Clocks for Message Ordering

Hydra uses a combination of sequence numbers and physical clocks to order messages. Concretely, each Hydra sequencer possesses a local *physical clock* that is strictly monotonically increasing; each sequencer also maintains a sequence number for each receiver group. Physical clocks are loosely synchronized across sequencers, but this is *not* required for safety; clock skew can only slow progress. Safety of Hydra only depends on physical clocks not drifting *backwards*. This requirement is already common practice: existing clock synchronization protocols such as NTP ensure that clocks can only move forward [51].

Each Hydra groupcast message is routed to one sequencer before being forwarded to all receivers in each destination group. When a sequencer receives a groupcast message, in addition to incrementing the sequence number for each recip-

Algorithm 2 ReceiverHandlePacket(pkt)

M: total number of sequencers
gid: receiver group ID
buf: ordered queue of undelivered messages
$s[M]$: largest sequence number received (per sequencer)
$c[M]$: largest clock value received (per sequencer)

```
1:  if pkt.seq[gid] ≤ s[pkt.id] then
2:      return
3:  end if
4:  c[pkt.id] ← max{c[pkt.id], pkt.c}
5:  Deliver DROP-NOTIFICATION for
        (s[pkt.id] + 1 ... pkt.seq[gid] − 1) (inclusive)
6:  s[pkt.id] ← pkt.seq[gid]
7:  if pkt is not a flush message ∧ pkt ∉ buf then
8:      Add pkt to buf
9:  end if
10: for p in buf do
11:     if p ⪯ min{(c[m], m) : m ∈ (1...M)} then
12:         Dequeue p from buf and deliver p
13:     else
14:         break
15:     end if
16: end for
```

ient group and inserting a multi-stamp, it writes its current clock value into the packet (Algorithm 1 line 2-5). Note that reading the clock value and incrementing sequence numbers must be done in an atomic block. Strict monotonicity of physical clocks and the above atomicity requirement ensure the following: for any two groupcast messages m_1 and m_2 with an overlapping recipient group g sequenced by the same sequencer, $s_1 \neq s_2 \wedge (s_1 < s_2 \iff c_1 < c_2)$, where s_1 and s_2 are the assigned sequence numbers for g, and c_1 and c_2 are the assigned clock values.

With a clock value inserted into each groupcast message, Hydra defines the partial ordering ($\prec$) of groupcast messages in the following way: for groupcast messages m_1 and m_2 with overlapping recipient groups and clock values c_1 and c_2 sequenced by sequencers with IDs i and j, $m_1 \prec m_2$ if $c_1 < c_2 \vee (c_1 = c_2 \wedge i < j)$ [2]. Breaking ties between equal clock values using sequencer IDs is necessary in guaranteeing the partial order property in §4.1.

Hydra groupcast receivers deliver groupcast messages to their users *according to our partial order*. If a receiver receives groupcast messages m_2 before m_1 with $m_1 \prec m_2$, it must either deliver a DROP-NOTIFICATION for m_1 before delivering m_2 or add m_2 to a buffer until it receives m_1. However, delivery based on physical clocks alone is not strong enough to detect message drops.

4.3.2 Combining Physical Clocks and Multi-Stamps for Drop Detection

Attaching sequence numbers to messages offers the useful property that any dropped message can be detected by observing gaps in the number sequence. Unfortunately, this property is lost when using physical clocks to order messages – a receiver seeing a message with a clock value c cannot determine if it missed any message with $c' < c$. To detect message drops, Hydra combines physical clock values and sequence numbers from multiple sequencers. Hydra receivers buffer incoming messages and deliver them in clock value order, but *only* once they have determined – based on sequence numbers – that no message with a lower clock value from another sequencer will be delivered.

Specifically, each Hydra receiver maintains two values for each sequencer i (Algorithm 2): the largest group sequence number $s[i]$ it has received from i, and the largest clock value $c[i]$ among messages it has received from i. Let c_{min} be the minimum value among all $(c[i],\ i)$ tuples, ordered by $\prec$. A Hydra receiver delivers messages using the following rules:

(i) it delivers pending groupcast messages in clock value and sequencer ID order (line 10),
(ii) it will only deliver a single message or DROP-NOTIFICATION for each sequence number from each sequencer (lines 1 and 6),
(iii) it only delivers groupcast messages m if $m \preceq c_{min}$ (line 11), and
(iv) when receiving groupcast message m with sequence number s from sequencer i, if $s > s[i]+1$, it delivers a DROP-NOTIFICATION for each message from $s[i]+1$ to $s-1$ (line 5).

From our discussion in §4.3.1, rules (i) and (ii) ensure the partial ordering property of Hydra groupcast. To show how rules (iii) and (iv) enforce drop detection, we leverage a key invariant: for a receiver r in group g and for any groupcast message m that has g as a recipient group, if $m \preceq c_{min}$, then r has either received m, or has received another groupcast message m' stamped with a higher sequence number from the same sequencer. With this invariant, the drop detection property of Hydra groupcast is guaranteed, since r either delivers m (m is received and rules (i) and (iii)), or delivers a DROP-NOTIFICATION for m (m' is received and rule (iv)). As an optimization, Hydra receivers can delay the delivery of DROP-NOTIFICATIONs until a message is needed to advance c_{min}. Because Hydra is robust to message reordering, this does not affect the correctness of the receiver protocol.

4.3.3 Ensuring Progress with Flush Messages

Our groupcast design so far ensures all the properties listed in §4.1, but has one remaining issue. In order for a receiver to make progress in delivering messages, it needs to receive groupcast messages from *all* sequencers to advance c_{min}. For instance, if a receiver has received message $m \succ c_{min}$, to deliver m, the receiver needs to receive messages from other sequencers to advance c_{min}. Consequently, any single sequencer that stays idle for an extended period of time would impede the progress of all groupcast receivers in the system.

To ensure progress in message delivery, each sequencer

[2] For ease of exposition, we slightly abuse the $\prec$ notation: it applies to both groupcast messages and (*clock value*, *sequencer ID*) tuples.

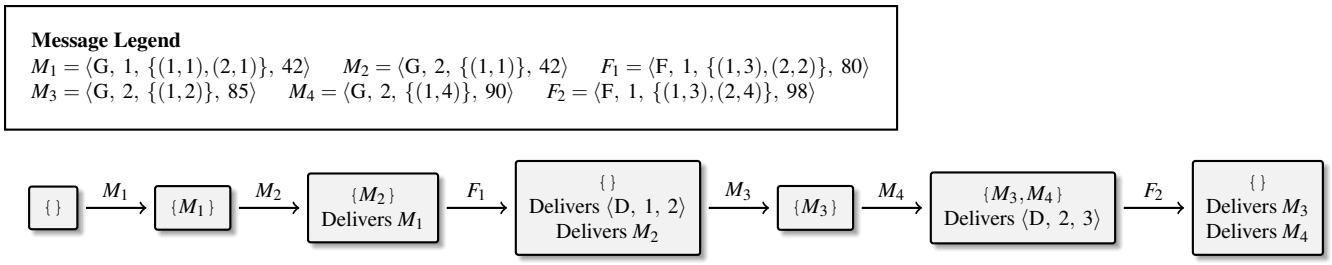


Figure 1: An example execution of the Hydra message delivery protocol. At every step, the state of the Hydra buffer is displayed along with any messages delivered to the application. Hydra groupcasts are written ⟨G, `sequencer`, `multi-stamp`, `timestamp`⟩*, while flush messages are written* ⟨F, `sequencer`, `multi-stamp`, `timestamp`⟩*, and* DROP-NOTIFICATIONS *are written* ⟨D, `sequencer`, `sequence_num`⟩*. A multi-stamp is a set of* (`group`, `sequence_num`) *tuples. This execution follows a receiver in group 1 receiving messages from two sequencers, 1 and 2. Transitions between states of the receiver show the message being received.*

periodically sends a *flush* message to all groupcast receivers containing its current clock value and the latest sequence number that the sequencer has sent to each receiver group (without incrementing). When a receiver receives a flush message from sequencer i, it follows the same procedure (§4.3.2) to advance $c[i]$, c_{min}, and $s[i]$. Applications are unaware of flush messages. Receivers, however, still use the sequence number in flush messages to deliver DROP-NOTIFICATIONS, following rule (iv) (Algorithm 2 line 5). Again, Hydra receivers can delay the delivery of DROP-NOTIFICATIONS until they are needed as an optimization.

The above protocol guarantees that, in the absence of failures (we discuss failure handling below), all received groupcast messages on every receiver will eventually be delivered. Clock divergence on different sequencers can delay message delivery (up to the clock skew), since c_{min} on each receiver depends on the sequencer with the slowest clock value, but cannot violate any of the safety properties.

Figure 1 shows an example execution of Hydra from the point of view of a single receiver receiving groupcast and flush messages from two sequencers. At every step of the execution, the receiver accepts an incoming message and delivers groupcasts and DROP-NOTIFICATIONS and retains pending groupcasts in its local buffer according to the rules defined in §4.3.2.

4.4 Handling Sequencer Failures

If a sequencer fails or link failures occur between a sequencer and some of the groups, some (or all) groupcast receivers will stop delivering messages, since they no longer receive messages from the failed sequencer and are unable to advance c_{min}. We use a reconfiguration protocol to address this issue.

Concretely, each Hydra deployment uses a centralized, fault-tolerant configuration service to manage a sequence of configurations. Each configuration specifies the set of sequencers and groupcast receivers (here we only discuss changes to sequencers across configurations). Groupcast receivers also store the current configuration locally. When a receiver suspects that sequencer j in the current configuration n has failed, e.g., when it has not received messages from sequencer j in a timeout period, it notifies the configuration service. The configuration service creates a new configuration $n+1$ with sequencer j removed, and sends the configuration to all groupcast receivers.

When groupcast receivers receive the new configuration, they run an agreement protocol to agree on the last sequence number each receiver group should deliver from the failed sequencer. To do so, each Hydra receiver additionally stores, for each sequencer, the largest sequence number it has seen in a multi-stamp for each receiver group (not just its own). To continue the sequencer removal process, each receiver sends a message with the largest sequence numbers (for all groups) it has received from the failed sequencer to the configuration service and stops processing messages with higher sequence numbers from that sequencer. Once the configuration service receives a quorum of these messages from each receiver group, it aggregates them to derive the highest sequence number each receiver group has or should have received from the failed sequencer. The configuration service sends a removal message to each group. A groupcast receiver delivers all necessary DROP-NOTIFICATIONS based on this removal message and continues to deliver messages following the rules in §4.3.2; the removal message serves as a final flush from the failed sequencer (with an infinitely large timestamp). Once all pending messages from the failed sequencer have been delivered, the receiver can safely transition to the new configuration. To avoid inconsistencies caused by different configurations, a receiver always attaches its current configuration number when delivering messages to the application.

Discussion. How does Hydra's recovery protocol compare, in terms of availability, to single-sequencer systems like those originally used by NOPaxos and Eris? Like these systems, Hydra experiences an interruption in message delivery caused by the failure of a sequencer. However, Hydra receivers can resume delivering messages from the other sequencers once they run the above protocol, which requires coordination only between the receivers, not the network layer. Thus, its unavailability period depends only on failure detection time and the agreement protocol latency, which can be orders-of-

magnitude shorter than the duration of network rerouting. It can also support more aggressive sequencer removal with shorter timeouts. Even though deploying more sequencers increases sequencer failure probability, by avoiding network rerouting on the critical path, Hydra still achieves overall improvement in system availability.

Adding new sequencers. To add a sequencer k to the system, the configuration service similarly creates a new configuration $n+1$ with sequencer k added, and sends the configuration to all receivers. Once a receiver receives the new configuration, it stops delivering groupcast messages to the application, and waits until it receives a flush message from the new sequencer k that has a higher timestamp than its latest delivered message. It then sends that flush message to the configuration service. When the configuration service receives a quorum of flush messages from each receiver group, it picks the flush message with the highest timestamp, denoted as t_k, and broadcasts that flush to all receivers; t_k effectively serves as the starting time of the new configuration. A receiver then resumes delivering messages for the previous configuration, until the next-to-be-delivered message has timestamp bigger than t_k. At that point, the receiver transitions to the next configuration, sets $s[k]$ to the sequence number in the flush message it receives from the configuration service, and starts delivering messages from the new sequencer. Note that if an old sequencer (removed previously) rejoins in a new configuration, the sequencer's ID is reassigned to a value unique from all other IDs, and its sequence numbers are all reset.

4.5 Correctness

We provide a detailed discussion of the safety of the Hydra protocol in Appendix B. In addition, a TLA$^+$ specification [40] of the Hydra groupcast and sequencer addition/removal protocols (Appendix C) has been model checked against Hydra's safety guarantees.

The liveness of the Hydra protocol is straightforward: as long as (1) receivers continue to receive groupcasts or flush messages from non-failed sequencers, and (2) the configuration service remains available and can communicate with a quorum of each receiver group to remove failed sequencers and complete the addition of new sequencers, Hydra groupcasts will be delivered.

4.6 Optimizations

Flush messages facilitate progress of Hydra receivers. However, generating flush messages at an overly aggressive rate will adversely affect a receiver's performance, as these flush messages consume network, CPU, and I/O resources. To strike a balance between message delivery latency and throughput, we propose two optimizations: receiver-side flush message solicitation and in-network flush message aggregation.

4.6.1 Receiver-Side Flush Message Solicitation

In our basic protocol described in §4.3.3, sequencers periodically send flush messages to all receivers. We can manually tune the flush message generating interval `T` on sequencers to adjust the latency/throughput trade-off: a smaller `T` improves message delivery latency but increases the load on the receivers, while a larger `T` has the opposite effect. However, since flush messages are broadcast to all receivers, this one-value-for-all policy cannot account for the different processing capacities and load levels on different receivers. Moreover, blindly sending flush messages every `T` time unit, particularly when `T` is small, can result in significant amount of unnecessary traffic. To see why this is the case, consider a receiver that currently has no message to deliver. Any flush message sent to this receiver, before the next Hydra message (with a higher clock value) arrives, will have no effect on the receiver's delivery progress and thus are strictly unnecessary.

Our key observation is that *receivers*, not sequencers, have perfect knowledge of when flush messages are required: receivers only need flush messages to make progress *when they possess undelivered messages*. We therefore propose a receiver-centric optimization, in which sequencers do not actively generate flush messages; instead, receivers explicitly request flush messages when needed. This optimization also enables various solicitation policies on the receivers. To optimize for latency, a receiver can immediately request flush messages when it receives groupcast messages that cannot be delivered. To optimize for throughput, it can delay requesting flush messages, equivalent to a batching approach. It can also apply a more sophisticated approach where it determines the solicitation delay based on the current load of the receiver, adaptively optimizing for both latency and throughput.

4.6.2 In-Network Flush Message Aggregation

Our message delivery rules (§4.3.2) require that a receiver delivers a groupcast message if and only if it has received messages with higher clock values from **all** other sequencers. The implication of this rule is that, the number of flush messages required to deliver a groupcast message increases *linearly* with the number of sequencers. To further reduce the processing overhead caused by excessive flush messages, we propose an advanced optimization technique inspired by recent in-network aggregation work. Concretely, we leverage ToR programmable switches connected to Hydra receivers to track each sequencer's clock value and sequence numbers. These numbers are updated when a ToR switch receives a flush message, but it does not immediately forward the flush message to the receiver. Only when the minimum stored clock value becomes large enough, the switch sends a single aggregated flush message containing all the clock values and sequence numbers to the receiver. To accurately determine this threshold, receivers attach the largest clock value among all undelivered messages in its flush message solicitation request. The ToR switch uses this value as the clock threshold,

which guarantees that the aggregated flush message would allow the receiver to deliver all undelivered messages in the buffer (those when the solicitation request was made). By applying our in-network aggregation optimization, the number of flush messages a receiver processes remains constant regardless of the number of sequencers.

5 Hydra Implementation

A Hydra deployment contains a dynamic set of groupcast senders, receivers, and sequencers, managed by a configuration service. We use a centrally-controlled SDN approach for managing groupcast routing: a POX [58]-based SDN controller installs rules that route groupcast messages to a randomly-selected reachable sequencer. When using end-host sequencers, we use a source routing approach: the configuration service tracks addresses of sequencers, which are cached on Hydra senders. When sending groupcast messages, senders randomly pick one of the sequencers and send to it via unicast. No special network routing is required.

Hydra sequencers each maintain minimal state: a unique sequencer ID, a sequence number for each receiver group, and a physical clock that is monotonically increasing. One of our key design principles is *simplicity*. It enables us to implement a Hydra sequencer *efficiently* on different hardware platforms.

In-network sequencing using programmable switches. Implementing Hydra sequencers in the data plane of network switches offer the highest sequencing performance, as current programmable switches can process billions of packets per second, with switching latency consistently under a few hundred nanoseconds. Hydra groupcast is implemented as an application-level protocol atop UDP. We reserve a special UDP port for Hydra groupcast, and append a customized Hydra header after the UDP header. The Hydra header includes a bitmap to specify the destination groups, a vector of sequence numbers (one for each destination group), and a single clock value. The switch implementation uses one switch register array element for each receiver group to store its current sequence number. The switch checks each bit of the bitmap, and for each enabled bit, increments the corresponding sequence number register and writes the sequence number into the Hydra header. Since there is no dependency across groups when processing a groupcast message, bit checking and sequence number updating for all destination groups can be done in parallel. This enables us to significantly reduce the required pipeline stages, allowing us to scale to higher number of groups. Subsequently, the switch stamps the hardware clock time into the header, and uses the replication engine to multicast the packet to receivers.

End-host Sequencers. Implementing Hydra sequencers on end-host servers offers better flexibility and portability, particularly attractive for deployments that cannot deploy specialized hardware. The downside is comparatively lower packet processing performance. Our Hydra protocol, however, enables scaling sequencing performance by adding additional sequencers. As we will show in our evaluation (§7.1.3), throughput of Hydra scales linearly with the number of sequencers.

Sender and receiver libraries. Hydra provides user-space libraries for sending and receiving groupcast messages. In addition to coordinating with the configuration service to track active sequencers and groups, this library also implements receiver-side buffering to deliver messages in the right order, and the flush message solicitation policies of §4.6.1. We have implemented two I/O stacks for the libraries. First, a polling-based DPDK [23] stack for efficient, kernel-bypassed packet processing. Second, a Linux-based transport using sockets and libevent [45] for better compatibility. Our evaluation in §7 uses the DPDK stack.

6 Building Distributed Systems using Hydra

Our Hydra groupcast primitive has a unique set of trade-offs between its guarantees and efficiency of the implementation. Compared to best effort primitives such as unicast and IP multicast, Hydra offers strong message ordering guarantees; compared to atomic broadcast and atomic multicast primitives, Hydra does not guarantee reliable message delivery, but can be implemented efficiently using a single phase protocol. In order to show the benefits of its design, we applied Hydra to two recent distributed systems – NOPaxos and Eris [42, 43] – and built a state machine replication called HydraPaxos and a distributed transaction processing system called HydraTxn.

Hydra's groupcast provides the same guarantees as the network protocols used in NOPaxos and Eris (Ordered Unreliable Multicast and Multi-sequenced Groupcast). Therefore, Hydra is readily composed with these existing protocols. HydraPaxos and HydraTxn use the NOPaxos and Eris protocols to tolerate server faults and handle DROP-NOTIFICATION, while use Hydra to provides message ordering guarantees and allows the adding and removing of sequencers. The only necessary modifications to NOPaxos and Eris are the disabling of their sequencer failure handling protocols, as this is handled by Hydra itself. Both HydraPaxos and HydraTxn can commit operations in a *single round trip* in the normal case.

HydraPaxos. HydraPaxos is a state machine replication system based on NOPaxos that tolerates crash failures of less than half of the replicas (or equivalently, with $2f+1$ replicas, HydraPaxos tolerates f crash failures). It guarantees linearizability [30] as long as the application state machine is deterministic. Each HydraPaxos deployment registers a unique Hydra groupcast address. HydraPaxos clients send state machine operations as a groupcast message with a single destination group. Each replica in a HydraPaxos deployment acts as a single Hydra receiver of the group. HydraPaxos operations are handled in a single round trip in the normal case. Once Hydra delivers an operation to the replicas, the replicas use the NOPaxos protocol to ensure operations are committed durably. Briefly, each replica adds the operation to

its log, and the *leader* replica executes the operation against the current state. Clients wait for consistent replies from a majority of replicas (including the leader) before considering a reply committed. When a DROP-NOTIFICATION is delivered to a replica, replicas need to reach consensus on the fate of the message – either to process or to permanently ignore – to ensure linearizability. The replica first attempts to recover the missing message by contacting other replicas in the group. If replicas fail to recover the dropped message, they coordinate (driven by the leader) to commit the message as a NO-OP.

HydraTxn. HydraTxn, is a fault-tolerant, distributed transaction processing system. HydraTxn partitions the entire data store into multiple shards with each shard replicated on multiple servers. Clients wrap data reads and writes into transactions. HydraTxn guarantees atomic, strict serializable execution of the transactions, and tolerates failures of less than half of the replicas in each shard. Similar to HydraPaxos, each HydraTxn deployment uses a unique Hydra groupcast address. Each shard of the deployment is assigned a unique group, and each replica in the shard registers a Hydra group receiver, delivering Hydra messages and DROP-NOTIFICATIONs. For transactions that qualify as independent transactions – stored procedures that has no dependency across shards and requires no client interactions – clients send the transaction in a single Hydra groupcast message destined to all the involved shards. HydraTxn also supports more general transactions by dividing them into multiple independent transactions and using two-phase locking on the servers to ensure isolation. As in HydraPaxos, independent transactions are handled in a single round trip in the normal case. Replicas in each shard involved in the transaction log the transaction and reply to the client, with the leader of each shard additionally executing the transaction. Clients wait for majority quorums from each shard to reply before considering a transaction committed. Since a transaction groupcast may involve multiple shards, DROP-NOTIFICATION requires all involved shards, not just the local group, to reach consensus on the reception/dropping decision. Similar to Eris [42], we use a logically separate, fault-tolerant failure coordinator service to manage this agreement protocol.

7 Evaluation

Our Hydra implementation includes Hydra host libraries, switch data and control planes, end-host sequencers, and Hydra co-designed replication (HydraPaxos) and transactional (HydraTxn) protocol implementations. The switch data plane is implemented in 1040 lines of P4 [11] code, and the switch control plane is written in 493 lines of Python code. We implemented the end-host sequencer, Hydra host libraries, the HydraPaxos protocol, and the HydraTxn protocol in approximately 8000 lines of C++ code.

Our evaluation testbed consists of 10 nodes connected to an APS BF6064X-T (Barefoot Tofino-based) programmable switch. We ran servers/replicas on nodes with dual 2.90GHz Intel Xeon Gold 6226R processors (32 total cores), 256 GB RAM. We used the remaining nodes to run clients and end-host sequencers. Clients use 2.10GHz Intel Xeon Gold 6230 processors (20 total cores) and 96 GB RAM. All nodes ran Ubuntu Linux 20.04 and use Mellanox ConnectX-5 100 GbE NICs. We statically partitioned resources on the programmable switch to implement multiple switch sequencers.

7.1 Hydra Groupcast Microbenchmarks

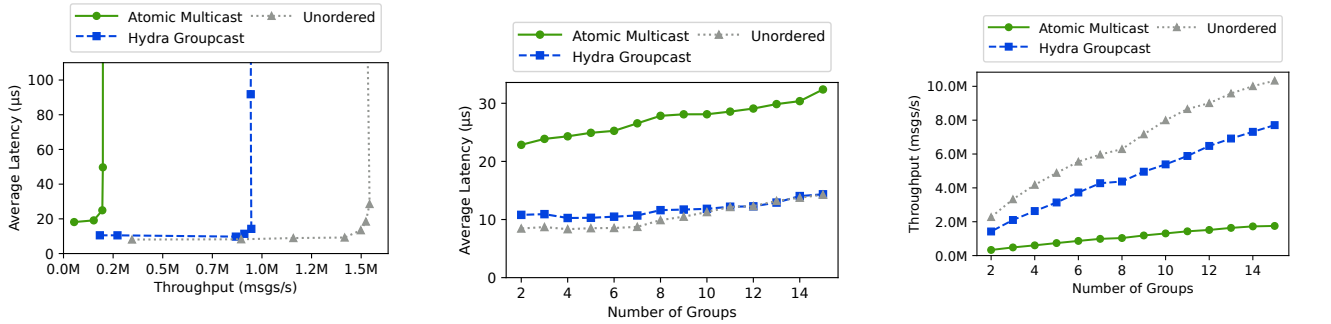
We first used microbenchmarks to evaluate the performance of our Hydra groupcast primitive. We ran closed-loop clients, each sending groupcast messages to a set of receiver groups. When a Hydra receiver *delivers* a groupcast message, it immediately replies to the client. Clients send the next groupcast message when they receives replies from each receiver in all destination groups (we assume no receivers fail).

We compared Hydra to two other groupcast implementations. First, we implemented a version of genuine atomic multicast [28]. To atomic multicast a message, a client first sends the message to all the receivers in each destination group. Receivers in each group run a consensus round to agree on a message timestamp and send the group timestamp to the client. The client picks the highest timestamp as the final message number, and forwards the message number to all involved group receivers. Receivers deliver messages in message number order. Second, we implement an unordered multicast – receivers immediately deliver client messages without any ordering guarantee – as a baseline.

7.1.1 Latency and Throughput

In the first experiment, we used a single group with three receivers to evaluate the base case performance. Two switch sequencers were deployed when evaluating Hydra, and we applied the receiver-side solicitation optimization (§4.6.1). We gradually increased the offered client load, and measured both the latency and the throughput of each system. As shown in Figure 2a, Hydra achieves a 378% increase in throughput and 42% reduction in latency compared to atomic multicast. Running consensus among group receivers for each message adds substantial throughput and latency overheads to atomic multicast. On the contrary, Hydra receivers require no coordination among each other to deliver messages in consistent order. In the worst case, they wait for a half RTT (receiver $\rightarrow$ switch $\rightarrow$ receiver) to receive flush messages in order to deliver a groupcast. This overhead is reflected in Hydra's small latency penalty (3 μs) compared to the baseline. As Hydra receivers can deliver messages without explicit flush messages when sequencers receive enough traffic, Hydra is able to attain throughput within 39% of the baseline.

Increasing group size. Next, we added more receivers to the group. When we increased the group size threefold (from three to nine), throughput of Hydra dropped only by 8%, and its latency remained the same. Hydra scales well to larger group sizes because receivers can independently determine the correct order of messages with no coordination. Perfor-



(a) Latency and throughput for a single group of three receivers

(b) Latency with increasing number of groups

(c) Maximum throughput with increasing number of groups

Figure 2: Latency and throughput of running a micro multicast benchmark. We use two switch sequencers for Hydra, and compare its performance to an atomic multicast and an unordered multicast protocol. For (b) and (c), we use a group size of three.

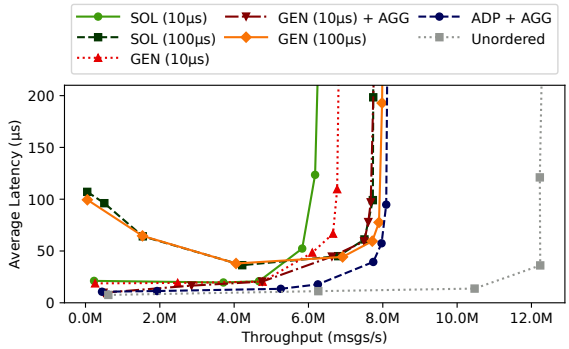


Figure 3: Impact of different flush message policies and parameters on performance of Hydra. The policies are: the receiver solicitation strategy (SOL), sequencer generation strategy (GEN), adaptive solicitation strategy (ADP), ToR switch aggregation (AGG).

mance of atomic multicast, however, degrades proportionally to the number of receivers as the cost of consensus increases. At group size of nine, Hydra outperforms atomic multicast by 567% in throughput and 56% in latency.

Scaling to more groups. To test how well Hydra scales to larger number of groups, we fixed the group size to three receivers, and increased the total number of groups. We used a workload where 80% of the groupcast messages were destined to a single group, and the remaining 20% had two destination groups. Clients chose destination groups following a uniform distribution. As shown in Figure 2b and Figure 2c, Hydra's throughput and latency continue to closely match the baseline. At 15 groups, throughput of Hydra is within 25% of the baseline, and 340% higher than atomic multicast.

7.1.2 Impact of Flush Messages

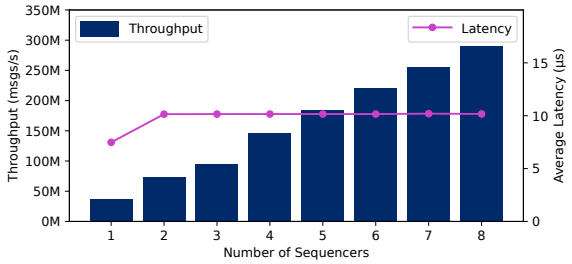
As we discussed in §4.6, policies for generating and handling flush messages can affect Hydra performance. To evaluate their effectiveness, we ran 15 groups each with three receivers, deployed four switch sequencers, and measured the latency and throughput of Hydra with increasing client load. We apply three flush message policies and show their impact in Figure 3: (1) sequencers periodically generate flush messages (GEN), (2) receivers solicit flush messages from sequencers after a delay (SOL), (3) receivers adaptively solicit flush messages based on current load (ADP). We also examine the impact of having ToR switches aggregate flush messages from sequencers (AGG).

When we use a higher delay for generating or soliciting flush messages, receivers on average need to wait longer to deliver messages. This effect is validated by the higher average latency experienced by GEN and SOL when their delay is at 100 μs. By decreasing the delay, both policies enjoy better message delivery latency. Unfortunately, it also degrades maximum throughput, as receivers use more CPU cycles to process flush messages – up to 14% lower throughput for GEN. ToR switch aggregation reduces the impact of frequent flush messages: AGG improves the throughput of GEN by 14%. Finally, by using an adaptive solicitation strategy, ADP achieves both low latency – it immediately requests flush messages when it has spare CPU cycles – and high throughput – it does not receive excessive flush messages at high utilization. As shown in Figure 3, it attains latency within 3μs and throughput within 33% that of the baseline result.

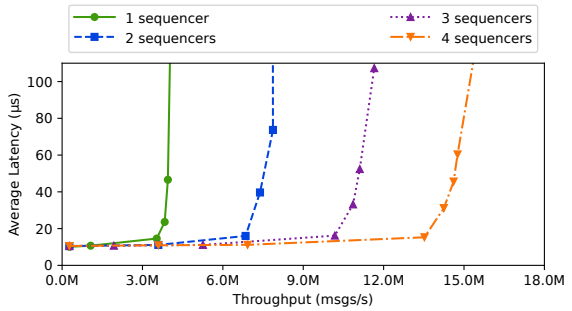
7.1.3 Sequencer Scalability

To evaluate the sequencer scalability of Hydra, we emulated an increasing number of switch sequencers (up to eight) on the same physical switch. For each emulated sequencer, we allocated a dedicated queue in the switch traffic manager that rate limits to 10 Gbps. Due to the limited number of physical servers, we only deployed 15 real Hydra groups, each with three receivers. To saturate the sequencers' capacity, we deployed additional virtual groups, whose request traffic were simply dropped at the switch egress ports. Figure 4a shows that throughput of the system increases linearly with more deployed switch sequencers. With eight sequencers, Hydra can process more than 250 million groupcast per second. The additional switch sequencers also have minimum impact on groupcast latency.

For clusters without programmable switches, sequencers can be deployed on end-host servers, offering an immediately



(a) Latency and sustainable throughput of Hydra with increasing number of switch sequencers



(b) Latency and throughput of Hydra with increasing number of end-host Hydra sequencers

Figure 4: Scalability of Hydra with increasing number of sequencers. We use 15 groups, three receivers per group, and deploy sequencers on switches and end-host servers. We also generate additional group-cast traffic to virtual Hydra groups to saturate the sequencers.

deployable solution. End-host sequencers, however, have limited processing capacity compared to an in-switch implementation. Figure 4b shows that Hydra can avoid this dilemma by adding more end-host sequencers, with near-linear scaling. With enough Hydra traffic, request load were evenly distributed among all sequencers. Since receivers need to wait for at least one message from each sequencer for message delivery, latency of Hydra increases slightly with more sequencers.

7.2 HydraPaxos Evaluations

Next, we evaluate the performance benefits of co-designing state machine replication (SMR) systems with Hydra. We compared our HydraPaxos to three other SMR protocols: Paxos (with the Multi-Paxos optimization), Fast Paxos, and NOPaxos. We also ran an unreplicated system with no fault tolerance as a baseline. All protocols were implemented in the same codebase for a fair comparison. We deployed each SMR system on three replica servers, ran an echo-RPC application, and measured the end-to-end latency and throughput of each system with increasing client load. We used two switch sequencers for HydraPaxos, and one switch sequencer for NOPaxos. Figure 5 shows HydraPaxos achieves significantly higher throughput than Paxos (204%) and Fast Paxos (180%), by avoiding replica coordination in the common case. Figure 5 also shows that our design can attain performance comparable to a network serialization approach: HydraPaxos achieves

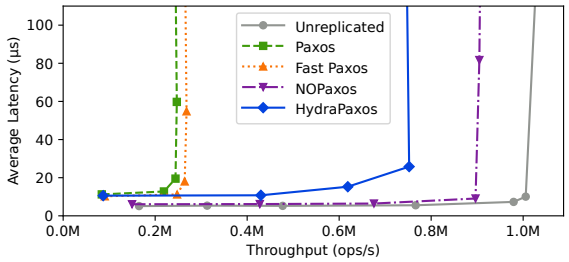


Figure 5: State machine replication system comparison. We measure the latency and throughput of HydraPaxos and other SMR protocols with three replicas. HydraPaxos uses two switch sequencers.

latency within 5 μs and throughput within 17% of NOPaxos. Like NOPaxos, HydraPaxos can sustain its throughput even with a moderate rate of packet drops ($\leq 0.1\%$), because drop recovery uses a lightweight protocol; a full evaluation appears in Appendix A.

7.3 HydraTxn Evaluations

The second distributed application we evaluated was a fault-tolerant, distributed transactional system. We compared HydraTxn with three other systems: Granola [19], Eris [42], and a standard distributed transactional system [17] called Lock-Store that uses two-phase commit, two-phase locking, and Paxos. All systems are implemented in the same code base. We deployed each system on 15 database shards, each replicated on three servers. HydraTxn uses two switch sequencers, while Eris uses only one. Similar to our experiment in §7.1.3, we rate limit each sequencer's bandwidth in the switch traffic manager and generate traffic to virtual Hydra groups.

We use the YCSB+T [16] benchmark that wraps `read` and `write` operations into stored-procedure style transactions. The workload we used consists of single-shard transactions with a read/write ratio of 1:1. Keys are selected using a uniform distribution. As shown in Figure 6, HydraTxn avoids server coordination overhead when processing transactions, leading to a 3.1× and 1.9× throughput, and a 49% and 13% latency reduction compared to Lock-Store and Granola. Performance of Eris is bottlenecked by the single switch sequencer. Excessive client load can even cause sequenced packets to be dropped in the network, leading to throughput collapse due to more frequent drop agreement protocol [42]. Hydra enables HydraTxn to scale beyond the central sequencer bottleneck, achieving a 47% throughput improvement over Eris.

We also tested HydraTxn's resilience to network anomalies by injecting simulated packet drops. As in the SMR experiment, small to moderate levels of packet drops have minimal impact on HydraTxn's performance (Appendix A).

7.4 Network-Level Load Balancing

To evaluate the impact of our approach on network properties, we simulated a data center network with a three-layer FatTree topology in NS3. The network consisted of 2560 servers and 112 switches. All servers generate background traffic follow-

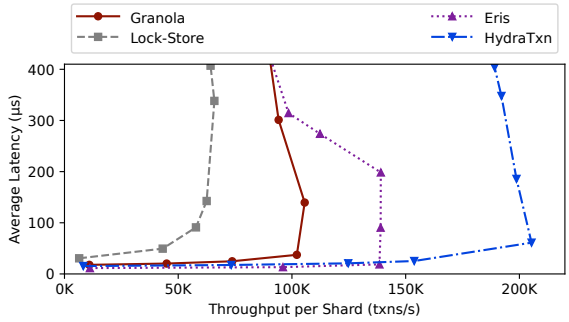
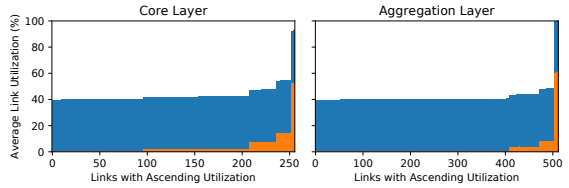
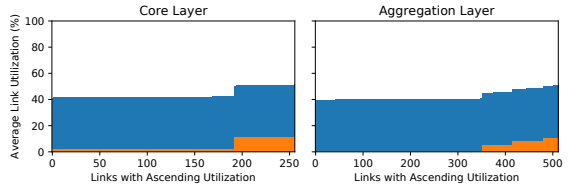


Figure 6: Distributed transactional system comparison. We measure the latency and per-shard throughput of HydraTxn and other transactional systems when running on 15 shards each replicated on three servers. HydraTxn uses two sequencers, while Eris uses one sequencer.



(a) Network link utilization when multicast messages traverse a single ToR switch sequencer



(b) Network link utilization when multicast messages are randomly routed to one of the eight ToR switch sequencers

Figure 7: Average link utilization of a simulated data center network. We simulate a three-layer FatTree topology with 2560 servers and 112 switches. Links between servers and ToR switches are 1 Gbps, and all other links are 10 Gbps.

ing a Poisson distribution. We set up 16 multicast receiver groups in the network, each with three receivers. We selected a few servers across the data center to generate periodic multicast messages, each message destined to a randomly selected group. We compared two approaches: a network serialization approach where all multicast messages are routed through a single ToR switch, and the Hydra approach where eight ToR switches are deployed as sequencers. Figure 7 shows the link utilization of each aggregation and core layer link for each approach. In the network serialization deployment, several aggregation layer links were fully saturated due to concentrated multicast traffic. By distributing multicast traffic across multiple sequencer switches, utilization of all core and aggregation links stayed below 50% in the Hydra deployment, demonstrating the load balancing benefit of our approach.

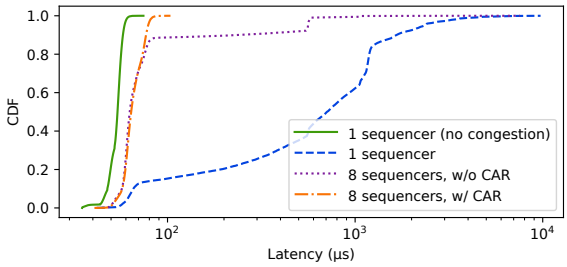
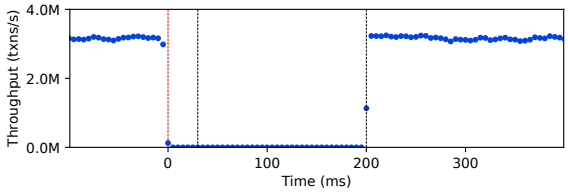
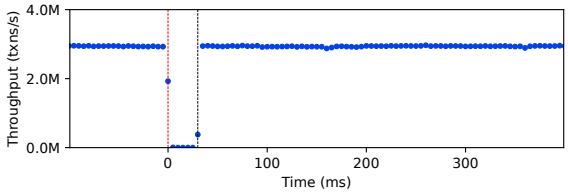


Figure 8: Latency distribution of multicast in the same simulated data center network as Figure 7. We generate bursty traffic to a single sequencer switch that causes congestion.



(a) Throughput of Eris during a sequencer failover



(b) Throughput of HydraTxn during a sequencer failover

Figure 9: Throughput of Eris and HydraTxn during a sequencer failover. For both, we injected a sequencer failure at time 0.

We then studied the impact of network congestion by generating bursty background traffic to one of the sequencer switches. Figure 8 shows that in a network serialization deployment, congestion at the sequencer switch caused median multicast latency to degrade by more than 13×. By distributing multicast traffic to multiple sequencer switches, Hydra reduces the impact of local congestion and improves the median latency by 11×. We also simulated *congestion-aware routing* (CAR) [3] for Hydra. By preferentially routing to non-congested sequencers, Hydra further improves multicast tail latency.

7.5 Sequencer Failover

Lastly, we evaluated the effectiveness of Hydra in handling sequencer failures and compared it to the network serialization approach. To do so, we used the same transactional system setup in §7.3, triggered a sequencer failure during the run, and measured the sustained throughput over time. As shown in Figure 9b, after the Hydra receivers detected the failure of one of the sequencers (we used a 30 ms timeout value), they immediately ran a reconfiguration protocol to remove the failed sequencer. The protocol only took a few hundred microseconds. HydraTxn was able to resume normal operation

and returned to its maximum throughput afterwards, using the remaining sequencers. By contrast, Eris (Figure 9a) relies on the network control plane needed to perform rerouting once a failure is detected, to forward client requests to a new sequencer switch. We simulated a 100 ms rerouting delay which matches results in the literature [38]. Unlike HydraTxn, Eris remained unavailable during network rerouting, demonstrating the benefit of our redundant sequencer approach.

8 Related Work

Ordered group communication primitives such as atomic multicast [28] have a long history, dating back to virtual synchrony [8], and have been implemented and used widely [4, 9, 32, 34, 62]. The classic atomic broadcast model is equivalent to consensus [13]. Our work explicitly adopts the *ordered but unreliable* communication model introduced by NOPaxos [43] and Eris [42], which enables network-accelerated sequencing.

Other distributed systems also use sequencers. CORFU [63] combines an unreliable sequencer with replicated storage on flash to build a shared log that can be used to build distributed data structures [7]. vCorfu [63] extends it to a multi-log abstraction analogous to multi-sequencing. Scalog [22] addresses the blocking reconfiguration and sequencer scalability issues of previous shared log designs by distributing log data to replicated data shards and periodically order log entries using an Paxos-based ordering layer. Hydra does not guarantee message persistence, so it avoids the overhead of intra-shard replication. Hydra also eliminates coordination among the sequencers on the critical path, which reduces message delivery latency and avoids potential bottlenecks of a centralized ordering service. Percolator [54] uses a sequencer for transaction processing, and deterministic databases like Calvin [60], SLOG [60], and Aria [47] combine sequencers with transaction schedulers for concurrency control.

Hydra builds on work on improving the scalability of consensus protocols. Its use of multiple active sequencers and flush messages is analogous to Mencius's rotating leader [48]. Hydra uses loosely synchronized clocks [46] to establish a total order, an idea used in concurrency control protocols like CLOCC [1], Spanner [17], and TAPIR [66]. Protocols like PTP [59] make clock synchronization widely available in data centers, and recent work like Sundial [44] and DPTP [35] demonstrates the precision available. Hydra's approach of using timestamps to order operations is similar to that of TEMPO [24]. However, TEMPO requires at least one and a half RTTs to commit a timestamp. Hydra, using network sequencers, can commit timestamps in half of an RTT even in the presence of concurrent requests. Similar to TEMPO, Hydra also waits for higher timestamps from other sequencers to ensure a timestamp is stable.

Hydra is designed to support programmable devices as sequencers, including PISA/RMT switch ASICs [12]. Recent work shows that these switches can implement complex protocols including consensus [20, 21] and chain replication [33]. Like NOPaxos and Eris, Hydra intentionally implements a limited set of sequencing functionality on the switch, leaving most of the protocol complexity at the end hosts. RedPlane [37] and SwiSh [64, 65] provide abstractions for replicating switch data plane state for reliability and scalability, respectively; sequencing, which requires strong consistency and frequent updates, represents a worst-case performance scenario for both, necessitating a different approach.

A concurrent effort, 1Pipe [41] uses programmable switches in a data center to implement causally and totally ordered communication. Senders in 1Pipe attach local timestamps to messages, and receivers deliver messages strictly in timestamp order. To determine when a timestamp is safe to deliver, switches in 1Pipe track barrier information from all ingress links and write the aggregated barrier timestamp into each packet. Hosts and switches periodically send beacon messages on idle links to ensure progress.

Hydra similarly uses timestamps to order messages. A key difference is that 1Pipe uses sender-generated timestamps, while in Hydra timestamps are generated by the sequencers. Consequently, 1Pipe requires synchronized clocks on *all* nodes in the network and in-network computation at *each* switch, a deployment challenge in heterogeneous networks where not all switches are programmable [57]; Hydra accommodates more practical deployments by only running logic on the sequencers and replicas, and only synchronizing clocks across sequencers. Moreover, in a 1Pipe deployment, any node, link, or switch failure in the network would stall the progress of all receivers; in Hydra, only failures local to the sequencers may impact progress.

9 Conclusion

The deployment of network sequencing approaches has been hindered because they require serializing messages through a single sequencer. Hydra addresses this with a new protocol that allows the concurrent use of multiple sequencers. A Hydra deployment serves as a drop-in replacement for sequencers in systems like NOPaxos and Eris, making their benefits more widely accessible. In particular, it scales beyond the performance of a single sequencer, which allows commodity servers rather than programmable switches; reduces system downtime during sequencer failures; and improves network load balancing by avoiding serialization.

Acknowledgments

We thank our shepherd Shuai Mu and the anonymous reviewers for their valuable feedback. We also thank Xin Zhe Khooi and Raj Joshi for their helpful comments on the P4 implementaion. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship. Jialin Li was supported by an MOE AcRF Tier 1 grant T1 251RES2104, an ODPRT SUG grant, and a Huawei research grant TC20211206645.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, June 1995. ACM.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of ACM SIGCOMM 2014*, 2014.

[4] Y. Amir and J. Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, Baltimore, MD, USA, 1998.

[5] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research*, CIDR '11, Asilomar, California, 2011.

[6] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, San Jose, CA, USA, 2012. USENIX Association.

[7] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, Farminton, Pennsylvania, 2013. Association for Computing Machinery.

[8] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 123–138, New York, NY, USA, 1987. Association for Computing Machinery.

[9] K. P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, page 79–86, New York, NY, USA, 1985. Association for Computing Machinery.

[10] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, jan 1987.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), July 2014.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013. ACM.

[13] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, page 147–158, New York, NY, USA, 1992. Association for Computing Machinery.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, Seattle, Washington, 2006. USENIX Association.

[15] Cisco data center infrastructure design guide 2.5. https://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, Hollywood, CA, USA, 2012. USENIX Association.

[18] I. Corporation. Intel Tofino 3 Intelligent Fabric Processor Brief. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html.

[19] J. Cowling and B. Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC '12, Boston, MA, 2012. USENIX Association.

[20] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. Network hardware-accelerated consensus. Technical Report USI-INF-TR-2016-03, Università della Svizzera italiana, May 2016.

[21] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, N. Zilberman, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. Technical Report USI-INF-TR-2018-01, Università della Svizzera italiana, May 2018.

[22] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, USA, Feb. 2020. USENIX.

[23] Data Plane Development Kit. https://www.dpdk.org/.

[24] V. Enes, C. Baquero, A. Gotsman, and P. Sutra. Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 178–193, New York, NY, USA, 2021. Association for Computing Machinery.

[25] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of ACM SIGCOMM 2011*, Toronto, ON, Canada, Aug. 2011.

[26] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable Consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, Cascais, Portugal, 2011. Association for Computing Machinery.

[27] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.

[28] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1–2):297–316, mar 2001.

[29] C. Guo, H. Wu, Z. Deng, J. Y. Gaurav Soni, J. Padhye, and M. Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of ACM SIGCOMM 2016*, Florianopolis, Brazil, Aug. 2016. ACM.

[30] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[31] Hesiod. Theogony. c. 730 BCE.

[32] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems*, 36(2):1–49, Apr. 2019.

[33] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, Apr. 2018. USENIX.

[34] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, DSN '11, page 245–256, USA, 2011. IEEE Computer Society.

[35] P. G. Kannan, R. Joshi, and M. C. Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 Symposium on SDN Research (SOSR '19)*, Santa Jose, CA, USA, Mar. 2019. ACM.

[36] X. Z. Khooi, L. Csikor, J. Li, and D. M. Divakaran. In-network applications: Beyond single switch pipelines. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 1–8, 2021.

[37] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan. RedPlane: Enabling fault tolerant stateful in-switch applications. In *Proceedings of ACM SIGCOMM 2021*, Virtual Conference, Aug. 2021. ACM.

[38] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 351–364, USA, 2010. USENIX Association.

[39] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, Prague, Czech Republic, 2013. Association for Computing Machinery.

[40] L. Lamport. The TLA+ home page. https://lamport.azurewebsites.net/tla/tla.html.

[41] B. Li, G. Zuo, W. Bai, and L. Zhang. 1pipe: Scalable total order communication in data center networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 78–92. Association for Computing Machinery, 2021.

[42] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, Shanghai, China, 2017. Association for Computing Machinery.

[43] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, Savannah, GA, USA, 2016. USENIX Association.

[44] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkipati, P. Chandra, and A. Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, Banff, AL, Canada, Nov. 2020. USENIX.

[45] libevent – an event notification library. https://libevent.org/.

[46] B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC '91)*, Montreal, QC, Canada, Aug. 1991. ACM.

[47] Y. Lu, X. Yu, L. Cao, and S. Madden. Aria: A fast and practical deterministic oltp database. *Proceedings of the VLDB Endowment*, 13(12):2047–2060, July 2020.

[48] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, San Diego, California, 2008. USENIX Association.

[49] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 39–50, New York, NY, USA, 2009. Association for Computing Machinery.

[50] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, Lombard, IL, 2013. USENIX Association.

[51] NTP clock discipline algorithm. https://www.eecis.udel.edu/~mills/ntp/html/discipline.html.

[52] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, Toronto, Ontario, Canada, 1988. Association for Computing Machinery.

[53] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '14, Philadelphia, PA, 2014. USENIX Association.

[54] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, Oct. 2010. USENIX.

[55] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, Vancouver, BC, Canada, 2010. USENIX Association.

[56] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, Oakland, CA, 2015. USENIX Association.

[57] D. R. K. Ports and J. Nelson. When should the network be the computer? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019. ACM.

[58] POX SDN controller. https://github.com/noxrepo/pox.

[59] IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. https://www.nist.gov/el/intelligent-systems-division-73500/ieee-1588.

[60] K. Ren, D. Li, and D. J. Abadi. SLOG: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11):1747–1761, July 2019.

[61] R. Van Renesse and D. Altinbuken. Paxos Made Moderately Complex. *ACM Computing Survey*, 47(3), Feb. 2015.

[62] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.

[63] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, Boston, MA, USA, 2017. USENIX Association.

[64] L. Zeno, D. R. K. Ports, J. Nelson, D. Kim, S. L. Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, Renton, WA, USA, Apr. 2022. USENIX.

[65] L. Zeno, D. R. K. Ports, J. Nelson, and M. Silberstein. SwiShmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets '20)*, Chicago, IL, USA, Nov. 2020. ACM.

[66] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems*, 35(4):12, Dec. 2018.

A Additional Evaluation

A.1 Clock Skew

Clock skew among sequencers does not affect Hydra correctness, but can delay message delivery progress. To evaluate the impact of clock skews, we deployed eight switch sequencers, 15 groups (no virtual groups), and three receivers in each group. We injected artificial clock skews to different sequencers, and measured both the latency and throughput of Hydra. As shown in Figure 10, clock skew does not impact Hydra throughput. Messages stamped by sequencers with faster clocks are buffered temporarily on the receivers, but the rate of delivering messages remains the same. At small to medium clock skews (1 to 10 μs), Hydra experiences marginal latency penalties (0 to 5 μs). Such clock skews are realistic: modern clock synchronization protocols [59] can maintain clock skews in the sub-microsecond range, and recent work has demonstrated synchronization error under 50 ns between programmable switches [35]. Even a 200 μs clock skew only resulted in less than 100 μs of added latency.

A.2 Message Loss for HydraPaxos and HydraTxn

Handling message drops. When Hydra messages are dropped in the network, HydraPaxos replicas need to coordinate to handle DROP-NOTIFICATIONs. To evaluate HydraPaxos's resilience to network anomalies, we measured its maximum throughput when an increasing percentage of packets were artificially dropped in the network. Figure 11 shows that HydraPaxos is able to sustain its high throughput even with a moderate rate of packet drops ($\leq 0.1\%$). HydraPaxos uses a lightweight protocol to recover from DROP-NOTIFICATIONs, as long as the message is not dropped on *all* replicas. At higher drop rates, throughput of HydraPaxos starts to decline due to more frequent coordination. We observe a similar level of throughput reduction for NOPaxos at these high drop rates.

We conduct the same experiment for HydraTxn. As in the SMR experiment, small to moderate levels of packet drops have minimal impact on HydraTxn's performance (Figure 12): its peak throughput decreased only by 11% even when the network dropped 1% of packets, and remained higher than that of Eris.

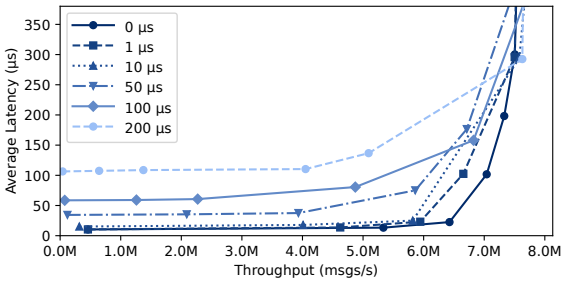


Figure 10: Latency and throughput of Hydra with increasing clock skew among sequencers. We use 15 groups, three receivers per group, and eight switch sequencers. Clock skew shows the maximum skew between any two sequencers.

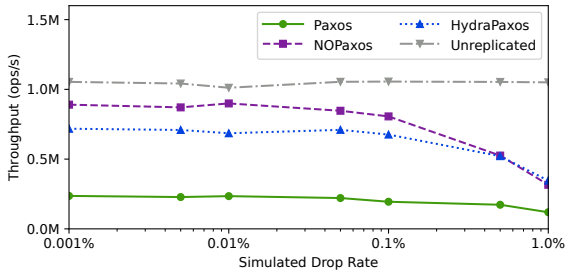


Figure 11: Maximum throughput of SMR systems with increasing packet drop rate. All systems run on three replicas. HydraPaxos uses two switch sequencers.

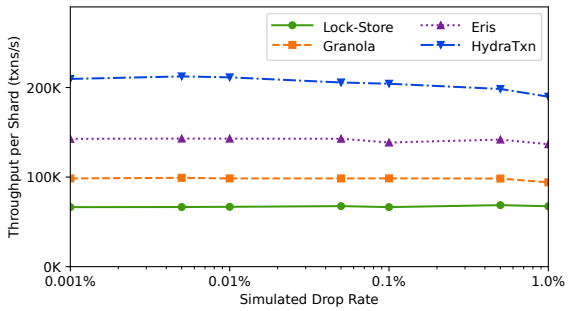


Figure 12: Maximum throughput of transactional systems with increasing packet drop rate. All systems run on 15 shards each replicated three-way. HydraTxn uses two switch sequencers.

B Proof of Safety

As specified in §3.1 Hydra provides the following guarantees to receivers of groupcast messages:

- **Partial Ordering.** All groupcast messages are partially ordered (the partial order relation is denoted as $\prec$) – all groupcast messages with overlapping destination groups are comparable. If groupcast message m_1 is ordered before m_2 ($m_1 \prec m_2$) and a receiver receives both m_1 and m_2, then every receiver delivers m_1 before m_2.
- **Unreliable Delivery.** Hydra only offers best effort message delivery. A groupcast message is not guaranteed to be delivered to any of its recipients.
- **Drop Detection.** If a groupcast message is not delivered to all its recipients, the primitive will notify the remaining receivers by delivering a DROP-NOTIFICATION. More formally, let R be the set of receiver groups for message m, then either one of the following two conditions holds: **all** receiver groups in R deliver m or a DROP-NOTIFICATION for m, or **none** of the receiver groups in R delivers m or a DROP-NOTIFICATION for m.

It is important to note that Hydra receiver groups, like NOPaxos and Eris receiver groups, use quorum-based protocols to decide which messages are delivered to the the group and which are permanently dropped. In order to tolerate the failure of some receivers in a receiver group, the drop detection requirement only considers each receiver group as a whole. Here, a receiver group delivers a message m or DROP-NOTIFICATION for m if every receiver in some quorum delivers m or a DROP-NOTIFICATION for m. Individual receivers in a group can diverge from the quorum when a sequencer is added or removed, and the receiver groups themselves must be able to handle this divergence. (NOPaxos and Eris do handle this case.) If an application using Hydra requires that the drop detection property apply uniformly to all receivers, then the quorum size for each receiver group is the size of the entire group.

Also important to note is that in the drop detection requirement, when we say a receiver delivers a DROP-NOTIFICATION for message m, what we mean is that the receiver delivers a DROP-NOTIFICATION for m *before delivering any message ordered after m in the partial order*. In this way, the drop detection requirement is indeed a safety requirement and not a liveness guarantee.

In the absence of sequencer failures, the correctness of Hydra's groupcast delivery is straightforward. Receivers deliver groupcast messages only when the message's clock value is less than or equal to $c_{\min}$, the minimum among the latest timestamps received from each of the sequencers. In fact, they only deliver messages whose timestamp is exactly $c_{\min}$, as ties in clock value are broken by sequencer ID. Because receivers only deliver messages in sequence number order, once message m is delivered by a receiver, no message with smaller sequence number or clock value from m's sequencer will be delivered by the receiver. Therefore, messages are always delivered in (timestamp, sequencer ID) order, which is a total order; the partial ordering guarantee is satisfied *a fortiori*. Furthermore, because receivers always deliver DROP-NOTIFICATION for smaller undelivered sequence numbers before delivering a message when there would be gaps in the sequence numbers delivered for that sequencer, the drop detection guarantee is satisfied.

In order to show that the sequencer removal process is correct, we first note that it is consistent with the Hydra safety guarantees for a receiver to at any time deliver a DROP-NOTIFICATION for the next sequence number yet to be delivered for some sequencer. The sequencer removal process is functionally equivalent to each receiver delivering infinitely many DROP-NOTIFICATIONs for all non-delivered sequence numbers for that sequencer. The agreement round is only necessary to determine exactly how many DROP-NOTIFICATIONs each receiver must explicitly deliver based on the results from each quorum. If a message m or a DROP-NOTIFICATION for m is delivered by a quorum from group g, and the sequencer that sequenced m is removed, then the configuration service is guaranteed to receive a multi-stamp with a sequence number for g at least as high as m's. Before transitioning to the new configuration (or delivering any message with a timestamp larger than m's), all other receiver groups that m was sent to must deliver a DROP-NOTIFICATION for m. Similarly, if no quorum from any receiver group received m or a message with sequence number larger than m's' before agreeing to

stop processing messages from the removed sequencer, then m will never be delivered by any receiver group (nor will an explicit DROP-NOTIFICATION for m be delivered by any receiver group). Therefore, for every groupcast sequenced by the removed sequencer, either all groups deliver the message or a DROP-NOTIFICATION for it or none do, satisfying the drop detection requirement.

When a sequencer is added, the flush message with timestamp t_k constructed by the configuration service when adding sequencer k is sent to all receiver groups. t_k is necessarily larger than the clock value of any message delivered by a quorum of receivers by construction. No message from sequencer k with clock value less than or equal to t_k will be delivered, nor will any DROP-NOTIFICATION for a message from sequencer k with clock value less than or equal to t_k. t_k was derived from a flush message that included sequence numbers for all groups, and upon entering the new configuration, a receiver immediately sets its sequence number for the added sequencer to the one included in this flush message. Conversely, once the new configuration starts, receivers in the new configuration will deliver messages or DROP-NOTIFICATIONs from the new sequencer with timestamp greater than t_k following the normal protocol for message delivery. Therefore, for any groupcast sequenced by the added sequencer, either all groups deliver the message or a DROP-NOTIFICATION for it or none do, satisfying the drop detection requirement.

C Hydra TLA+ Specification

MODULE *Hydra*

Specifies the *Hydra* protocol.

Receiver groups in this model are treated as single entities. This is done to increase model checking performance and avoid making assumptions about the protocol being run by the receiver groups. This specification focuses on the Hydra protocol and avoids the details of the quorum-based protocol being run by the receivers.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

Constants and Variables

CONSTANTS *numSequencers*, *receivers*, *initialActiveSequencers*

ASSUME $numSequencers \in Nat$
ASSUME $numSequencers > 0$
ASSUME $IsFiniteSet(receivers)$
ASSUME $IsFiniteSet(initialActiveSequencers)$
ASSUME $initialActiveSequencers \in$ SUBSET Nat

$sequencers \triangleq (1 \,..\, numSequencers)$

$mGroupcast \triangleq$ "mGroupcast"
$mFlush \triangleq$ "mFlush"
$mAddSequencer \triangleq$ "mAddSequencer"
$mFinishAdd \triangleq$ "mFinishAdd"
$mRemoveSequencer \triangleq$ "mRemoveSequencer"
$mFinishRemove \triangleq$ "mFinishRemove"

$vGroupcast \triangleq$ "vGroupcast"
$vDropNotification \triangleq$ "vDropNotification"

VARIABLES *messages*, *sequencerState*, *receiverState*, *configState*

$$
\begin{aligned}
Init \triangleq\ & \land messages = \{\} \\
& \land sequencerState = [s \in sequencers \mapsto \\
& \qquad [timestamp \mapsto 0, \\
& \qquad\ sequenceNums \mapsto [v \in receivers \mapsto 0] \\
& \qquad]] \\
& \land receiverState = [v \in receivers \mapsto [\\
& \qquad \text{Undelivered groupcasts} \\
& \qquad buffer \mapsto \{\}, \\
& \qquad \text{Delivered groupcasts and drop notifications} \\
& \qquad delivered \mapsto \langle\rangle, \\
& \qquad \text{Largest timestamps seen} \\
& \qquad timestamps \mapsto [s \in sequencers \mapsto 0], \\
& \qquad \text{Largest } sequenceNums \text{ seen} \\
& \qquad sequenceNums \mapsto [s \in sequencers \mapsto 0], \\
& \qquad \text{Currently active sequencers}
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\quad activeSequencers \;\mapsto initialActiveSequencers, \\
\qquad\qquad\quad \text{Sequencers being added} \\
\qquad\qquad\quad addedSequencers \;\mapsto initialActiveSequencers, \\
\qquad\qquad\quad \text{Sequencers being removed} \\
\qquad\qquad\quad removedSequencers \mapsto \{\} \\
\qquad\qquad]] \\
\wedge\ configState = [addedSequencers \;\mapsto initialActiveSequencers, \\
\qquad\qquad\quad removedSequencers \mapsto \{\}]
\end{array}
$$

Helper and Utility Functions

$Min(S) \triangleq$ CHOOSE $s \in S : \forall\, sp \in S : sp \geq s$
$Max(S) \triangleq$ CHOOSE $s \in S : \forall\, sp \in S : sp \leq s$

$Range(f) \triangleq \{f[x] : x \in$ DOMAIN $f\}$

RECURSIVE $SeqFromSet(_)$
$SeqFromSet(S) \triangleq$
  IF $S = \{\}$ THEN $\langle\rangle$
  ELSE LET $x \triangleq$ CHOOSE $x \in S :$ TRUE
       IN $\langle x \rangle \circ SeqFromSet(S \setminus \{x\})$

Short-hand way of sending a message
$Send(m) \triangleq messages' = messages \cup \{m\}$

Main utility function for delivering groupcasts and drop notifications.

Receiver r adds G to its buffer, increments sequencer s's timestamp to t, increments s's *sequenceNum* to n, and delivers deliverable groupcasts.

It also permanently removes all sequencers in the removed set.

$DeliverAvailable(r,\ G,\ s,\ t,\ n,\ removed) \triangleq$
LET
  $rstate \triangleq receiverState[r]$

  $newRemovedSequencers \triangleq rstate.removedSequencers \cup removed$
  $newActiveSequencers \triangleq rstate.activeSequencers \setminus removed$

  $bg \triangleq rstate.buffer \cup G$
  $oldLog \triangleq rstate.delivered$

  $newTimestamps \triangleq [rstate.timestamps$ EXCEPT $![s] = Max(\{@,\ t\})]$
  $newSequenceNums \triangleq [rstate.sequenceNums$ EXCEPT $![s] = Max(\{@,\ n\})]$

  Groupcasts about to be delivered
  $deliverable \triangleq \{gp \in bg :$
      $Min(\{newTimestamps[sp] :$
        $sp \in newActiveSequencers\}) \geq gp.timestamp\}$

```
    Newly/previously delivered Groupcasts + previous drop notifications
    delivered  ≜  Range(oldLog) ∪ deliverable

    newBuffer  ≜  bg \ deliverable

    Necessary drop notifications
    dropNotifications(sequencer)  ≜  {
        [vtype          ↦ vDropNotification,
         sequencer      ↦ sequencer,
         sequenceNum ↦ k] : k ∈ {l ∈ (1 .. newSequenceNums[sequencer]) :
            ¬∃ gp ∈ Range(oldLog) ∪ bg :
                ∨ ∧ gp.vtype = vDropNotification
                  ∧ gp.sequencer = sequencer
                  ∧ gp.sequenceNum = l
                ∨ ∧ gp.vtype = vGroupcast
                  ∧ gp.sequencer = sequencer
                  ∧ gp.sequenceNums[r] = l}}
    allDropNotifications  ≜  UNION {dropNotifications(s) : sp ∈ sequencers}

    orderedDropNotifications  ≜  SortSeq(
        SeqFromSet(allDropNotifications),
        LAMBDA d1, d2 : d1.sequenceNum < d2.sequenceNum)

    orderedDeliverables  ≜  SortSeq(SeqFromSet(deliverable),
        LAMBDA g1, g2 : ∨ g1.timestamp < g2.timestamp
                        ∨ ∧ g1.timestamp = g2.timestamp
                          ∧ g1.sequencer  < g2.sequencer)

    newLog  ≜  oldLog ∘ orderedDropNotifications ∘ orderedDeliverables
IN
    ∧ receiverState' = [receiverState EXCEPT ![r] =
                        [@ EXCEPT !.buffer = newBuffer,
                                  !.timestamps = newTimestamps,
                                  !.delivered = newLog,
                                  !.sequenceNums = newSequenceNums,
                                  !.activeSequencers = newActiveSequencers,
                                  !.removedSequencers = newRemovedSequencers
                        ]]
    ∧ UNCHANGED ⟨messages, sequencerState, configState⟩
```

Main Spec

```
If two receivers deliver groupcasts, they deliver them in the same order
GlobalOrder  ≜  ∀ r1, r2 ∈ receivers : LET
    d1  ≜  receiverState[r1].delivered
    d2  ≜  receiverState[r2].delivered
```

```
IN
∀ n1_1 ∈ (1 .. Len(d1)), n2_1 ∈ (1 .. Len(d2)) :
   ( ∧ d1[n1_1] = d2[n2_1]
     ∧ d1[n1_1].vtype = vGroupcast
     ∧ d2[n2_1].vtype = vGroupcast) ⇒
∀ n1_2 ∈ (1 .. n1_1), n2_2 ∈ (n2_1 + 1 .. Len(d2)) :
   ( ∧ d1[n1_2].vtype = vGroupcast
     ∧ d2[n2_2].vtype = vGroupcast) ⇒
   d1[n1_2] ≠ d2[n2_2]
```

If any receiver delivers a *Groupcast*, then all receivers deliver that Groupcast or a *DropNotification* before that timestamp

```
Delivery ≜ ∀ r1 ∈ receivers : LET
    d1 ≜ receiverState[r1].delivered
IN
∀ n1 ∈ (1 .. Len(d1)) :
   d1[n1].vtype = vGroupcast ⇒
LET
    g1 ≜ d1[n1]
    t ≜ g1.timestamp
IN
∀ r2 ∈ DOMAIN g1.sequenceNums :
LET
    d2 ≜ receiverState[r2].delivered
IN
∨ ∃ g2 ∈ Range(d2) : g1 = g2
∨ ¬∃ g2 ∈ Range(d2) : ∧ g2.vtype = vGroupcast
                      ∧ g2.timestamp ≥ t
∨ ∃ n2 ∈ (1 .. Min({x ∈ DOMAIN d2 :
                    d2[x].vtype = vGroupcast ∧ d2[x].timestamp ≥ t})) :
   ∧ d2[n2].vtype = vDropNotification
   ∧ d2[n2].sequencer = g1.sequencer
   ∧ d2[n2].sequenceNum = g1.sequenceNums[r2]
```

Groupcasts are always delivered in timestamp and sequence number order

```
LocalOrder ≜ ∀ r ∈ receivers :
LET
    deliveredGroupcasts ≜ SelectSeq(receiverState[r].delivered,
                                    LAMBDA g : g.vtype = vGroupcast)
    deliveredFromSequencer(s) ≜ SelectSeq(deliveredGroupcasts,
                                          LAMBDA g : g.sequencer = s)
    SeqNum(g) ≜ IF g.vtype = vGroupcast
                THEN g.sequenceNums[r]
                ELSE g.sequenceNum
IN
```

$\land \forall\, n \in (1\,..\,Len(deliveredGroupcasts) - 1) :$
$\quad deliveredGroupcasts[n].timestamp \leq deliveredGroupcasts[n+1].timestamp$
$\land \forall\, s \in sequencers : \forall\, n \in (1\,..\,Len(deliveredFromSequencer(s)) - 1) :$
$\quad SeqNum(deliveredFromSequencer(s)[n]) \leq$
$\quad SeqNum(deliveredFromSequencer(s)[n+1])$

$Safety \triangleq GlobalOrder \land Delivery \land LocalOrder$

Actions and Message Handlers

Advance time at sequencer s

$AdvanceTime(s) \triangleq$
$\land\ sequencerState' = [sequencerState \text{ EXCEPT } ![s] =$
$\qquad\qquad [@ \text{ EXCEPT } !.timestamp = @ + 1]]$
$\land$ UNCHANGED $\langle messages,\ receiverState,\ configState\rangle$

Sequencer s sends a groupcast to set of receivers R

$SendGroupcast(s) \triangleq \exists\, R \in \text{SUBSET } receivers :$
$\land\ Cardinality(R) > 0$
$\land\ Send([mtype \mapsto mGroupcast,$
$\qquad sequencer \mapsto s,$
$\qquad timestamp \mapsto sequencerState[s].timestamp + 1,$
$\qquad sequenceNums \mapsto [r \in R \mapsto$
$\qquad\qquad sequencerState[s].sequenceNums[r] + 1]$
$\qquad])$
$\land\ sequencerState' = [sequencerState \text{ EXCEPT } ![s] =$
$\qquad [@ \text{ EXCEPT } !.sequenceNums = [r \in receivers \mapsto$
$\qquad\qquad \text{IF } r \in R \text{ THEN } @[r] + 1 \text{ ELSE } @[r]],$
$\qquad\quad !.timestamp = @ + 1]]$
$\land$ UNCHANGED $\langle receiverState,\ configState\rangle$

Sequencer s sends a flush

$SendFlush(s) \triangleq$
$\land\ Send([mtype \mapsto mFlush,$
$\qquad sequencer \mapsto s,$
$\qquad timestamp \mapsto sequencerState[s].timestamp,$
$\qquad sequenceNums \mapsto sequencerState[s].sequenceNums$
$\qquad])$
$\land$ UNCHANGED $\langle sequencerState,\ receiverState,\ configState\rangle$

Receiver r receives an $mGroupcast$ message m from receiver i

$HandleGroupcast(r,\ m) \triangleq$
LET
$\quad rstate \triangleq receiverState[r]$
$\quad s \triangleq m.sequencer$
$\quad n \triangleq m.sequenceNums[r]$

```
    g ≜ [vtype          ↦ vGroupcast,
         timestamp      ↦ m.timestamp,
         sequencer      ↦ s,
         sequenceNums ↦ m.sequenceNums]
IN
    Don't accept Groupcasts if we're adding a sequencer
    ∧ rstate.addedSequencers ⊆
      (rstate.activeSequencers \ rstate.removedSequencers)
    Sequencer must be active and not being removed
    ∧ s ∈ rstate.activeSequencers
    ∧ s ∉ rstate.removedSequencers
    Don't receive if already handled
    ∧ g.sequenceNums[r] > rstate.sequenceNums[s]
    ∧ DeliverAvailable(r, {g}, s, m.timestamp, n, {})

 Receiver r receives an mFlush message m
HandleFlush(r, m) ≜
LET
    rstate ≜ receiverState[r]
    s ≜ m.sequencer
    t ≜ m.timestamp
    largestDeliveredTimestamp ≜ Max({0} ∪ {
        g.timestamp : g ∈ {gp ∈ Range(rstate.delivered) :
                           gp.vtype = vGroupcast}})
IN
    Don't accept flushes while adding sequencers
    ∨ ∧ rstate.addedSequencers ⊆
        (rstate.activeSequencers \ rstate.removedSequencers)
      ∧ s ∈ rstate.activeSequencers
      ∧ s ∉ rstate.removedSequencers
      ∧ DeliverAvailable(r, {}, s, t, m.sequenceNums[r], {})
    ∨ ∧ s ∈ rstate.addedSequencers \ rstate.activeSequencers
      ∧ s ∉ rstate.removedSequencers
      ∧ t > largestDeliveredTimestamp
      ∧ Send([mtype          ↦ mAddSequencer,
              receiver       ↦ r,
              sequencer      ↦ s,
              timestamp      ↦ t,
              sequenceNums ↦ m.sequenceNums])
      ∧ UNCHANGED ⟨sequencerState, receiverState, configState⟩

 Receiver r begins adding sequencer s
BeginAddSequencer(r, s) ≜
LET
    rstate ≜ receiverState[r]
```

```
IN
    ∧ s ∉ rstate.activeSequencers
    ∧ s ∉ rstate.addedSequencers
    ∧ s ∉ rstate.removedSequencers
    ∧ receiverState' = [receiverState EXCEPT ![r] =
                          [@ EXCEPT !.addedSequencers = @ ∪ {s}]]
    ∧ UNCHANGED ⟨messages, sequencerState, configState⟩

 The config service finalizes the addition of sequencer s
AddSequencer(s) ≜
LET
    adds ≜ {m ∈ messages : m.mtype = mAddSequencer ∧ m.sequencer = s}
    tStart ≜ Max({m.timestamp : m ∈ adds})
    seqsStart ≜ (CHOOSE m ∈ adds : m.timestamp = tStart).sequenceNums
IN
    ∧ s ∉ configState.addedSequencers
    ∧ ∀ r ∈ receivers : ∃ m ∈ adds : m.receiver = r
    ∧ Send([mtype        ↦ mFinishAdd,
            sequencer    ↦ s,
            timestamp    ↦ tStart,
            sequenceNums ↦ seqsStart])
    ∧ configState' = [configState EXCEPT !.addedSequencers = @ ∪ {s}]
    ∧ UNCHANGED ⟨sequencerState, receiverState⟩

 Receiver r receives an mFinishAdd message m
HandleFinishAdd(r, m) ≜
LET
    s ≜ m.sequencer
    rstate ≜ receiverState[r]
IN
    ∧ s ∉ rstate.activeSequencers
    ∧ s ∉ rstate.removedSequencers
    ∧ s ∈ rstate.addedSequencers
    ∧ receiverState = [receiverState EXCEPT ![r] =
      [@ EXCEPT !.activeSequencers = @ ∪ {s},
                !.timestamps = [@ EXCEPT ![s] = m.timestamp],
                !.sequenceNums = [@ EXCEPT ![s] = m.sequenceNums[r]]]]
    ∧ UNCHANGED ⟨messages, sequencerState, configState⟩


BeginRemoveSequencer(r, s) ≜
LET
    rstate ≜ receiverState[r]
    gs ≜ {g ∈ Range(rstate.delivered) ∪ rstate.buffer :
            ∧ g.vtype = vGroupcast
            ∧ g.sequencer = s}
```

```
    seqs ≜ [rp ∈ receivers ↦ Max({0} ∪
              {g.sequenceNums[rp] : g ∈
               {gp ∈ gs : rp ∈ DOMAIN gp.sequenceNums}})]
IN
    ∧ s ∉ rstate.removedSequencers
    ∧ Send([mtype          ↦ mRemoveSequencer,
            receiver       ↦ r,
            sequencer      ↦ s,
            sequenceNums ↦ seqs])
    ∧ receiverState' = [receiverState EXCEPT ![r] =
                          [@ EXCEPT !.removedSequencers = @ ∪ {s}]]
    ∧ UNCHANGED ⟨sequencerState, configState⟩

RemoveSequencer(s) ≜
LET
    removes ≜ {m ∈ messages :
                 m.mtype = mRemoveSequencer ∧ m.sequencer = s}
    lastSeqs ≜ [r ∈ receivers ↦ Max({0} ∪
                 {m.sequenceNums[r] : m ∈ removes})]
IN
    ∧ s ∉ configState.removedSequencers
    ∧ ∀ r ∈ receivers : ∃ m ∈ removes : m.receiver = r
    ∧ Send([mtype          ↦ mFinishRemove,
            sequencer      ↦ s,
            sequenceNums ↦ lastSeqs])
    ∧ configState' = [configState EXCEPT !.removedSequencers = @ ∪ {s}]
    ∧ UNCHANGED ⟨sequencerState, receiverState⟩

Receiver r receives an mFinishRemove message m
HandleFinishRemove(r, m) ≜
LET
    s ≜ m.sequencer
    rstate ≜ receiverState[r]
IN
    ∧ s ∉ rstate.removedSequencers
    ∧ DeliverAvailable(r, {}, s, 0, m.sequenceNums[r], {s})
```

Main Transition Function

```
Next ≜ ∨ ∃ s ∈ sequencers : ∨ AdvanceTime(s)
                            ∨ SendGroupcast(s)
                            ∨ SendFlush(s)
                            ∨ AddSequencer(s)
                            ∨ RemoveSequencer(s)
       ∨ ∃ m ∈ messages :
```

$$
\begin{array}{l}
\quad \lor \ \land \ m.mtype = mGroupcast \\
\qquad \land \exists \, r \in \text{DOMAIN } m.sequenceNums : HandleGroupcast(r,\ m) \\
\quad \lor \ \land \ m.mtype = mFlush \\
\qquad \land \exists \, r \in \text{DOMAIN } m.sequenceNums : HandleFlush(r,\ m) \\
\quad \lor \ \land \ m.mtype = mFinishAdd \\
\qquad \land \exists \, r \in receivers : HandleFinishAdd(r,\ m) \\
\quad \lor \ \land \ m.mtype = mFinishRemove \\
\qquad \land \exists \, r \in receivers : HandleFinishRemove(r,\ m) \\
\lor \exists \, r \in receivers : \exists \, s \in sequencers : \\
\quad \lor \ BeginAddSequencer(r,\ s) \\
\quad \lor \ BeginRemoveSequencer(r,\ s)
\end{array}
$$

The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems

Lei Zhang
Emory University and Princeton University

Zhiqiang Xie
Max Planck Institute for Software Systems

Vaastav Anand
Max Planck Institute for Software Systems

Ymir Vigfusson
Emory University

Jonathan Mace
Max Planck Institute for Software Systems

Abstract

Today's distributed tracing frameworks are ill-equipped to troubleshoot rare edge-case requests. The crux of the problem is a trade-off between specificity and overhead. On the one hand, frameworks can indiscriminately select requests to trace when they enter the system (head sampling), but this is unlikely to capture a relevant edge-case trace because the framework cannot know which requests will be problematic until after-the-fact. On the other hand, frameworks can trace everything and later keep only the interesting edge-case traces (tail sampling), but this has high overheads on the traced application and enormous data ingestion costs.

In this paper we circumvent this trade-off for any edge-case with symptoms that can be programmatically detected, such as high tail latency, errors, and bottlenecked queues. We propose a lightweight and always-on distributed tracing system, Hindsight, which implements a *retroactive sampling* abstraction: instead of eagerly ingesting and processing traces, Hindsight lazily retrieves trace data only *after* symptoms of a problem are detected. Hindsight is analogous to a car dash-cam that, upon detecting a sudden jolt in momentum, persists the last hour of footage. Developers using Hindsight receive the exact edge-case traces they desire without undue overhead or dependence on luck. Our evaluation shows that Hindsight scales to millions of requests per second, adds nanosecond-level overhead to generate trace data, handles GB/s of data per node, transparently integrates with existing distributed tracing systems, and successfully persists full, detailed traces in real-world use cases when edge-case problems are detected.

1 Introduction

Troubleshooting failures and performance problems in large-scale distributed systems is crucial. On one side, tiny performance misbehavior in a production system could be costly [1, 2, 19]. On the other side, exacerbated by growing system complexity, diagnosing problems takes onerous effort from system developers and requires significant engineering resources. Distributed tracing was invented as the solution of troubleshooting distributed systems by recording detailed, end-to-end traces of request executions, and have been proved helpful for a wide range of use cases [59, 62].

Prior distributed tracing works have demonstrated a wide range of use cases. Common-case analysis focuses on aggregated system behaviors, such as monitoring resource usage [46, 59, 60, 62, 70]. In contrast, edge-case troubleshooting (§2.1), the topic of this paper, focuses on rare and outlier system behavior, such as tail latency [18, 38, 48, 69, 74].

Since an edge case is rare by definition, tracing edge cases requires trace coverage of all requests. In typical production environments, tracing *every* request—including transmitting, processing, and storing comprehensive telemetry—requires enormous backend infrastructure and storage that is unacceptable to infrastructure operators. State-of-the-art tracing frameworks manage this overhead by collecting a small sample (0.001%) of traces [9, 34, 37, 64]. Though previous works practically reduce tracing overhead through head sampling [34, 64] and tail sampling [36, 37] techniques, they cannot trace edge cases at scale (§2.3).

In this paper, we resolve the problem of tracing edge-case requests in production environments. To achieve this, we focus our attention on *symptomatic* edge cases, where the performance effects of the problem manifest shortly after its causes and where the impacts can be observed programmatically. We propose **retroactive sampling** to collect telemetry data *back in time* from the present moment of detection from all machines that serviced the request. The key idea is to generate all trace data but only collect useful data through a retrieval mechanism.

To implement retroactive sampling, we built Hindsight—an always-on, lightweight distributed tracing system that is compatible with existing tracing APIs—as a practical tool for edge-case analysis. Under retroactive sampling, all trace data is recorded locally but only reported when a symptom is detected, allowing applications to generate copious trace data in case they are needed without encumbering the tracing system's backend collection infrastructure. Retroactive sampling ultimately reports the same volume of trace data as other sampling methods, but ensures that edge-case traces are not missed. To provide efficient and coherent retroactive sampling, Hindsight's design separates its dataplane, *e.g.* generating trace data into fast local memory, from control logic, *e.g.* for indexing metadata, coordinating among machines, and triggering collection for symptomatic requests on demand.

As demonstration, we apply Hindsight on three use cases corresponding to our running examples. We run experiments on the DeathStar Microservices Benchmark [24], the Hadoop Distributed File System [63], an Alibaba benchmark derived from production traces [42], and on several micro-benchmarks. We have integrated Hindsight with OpenTelemetry [52] and as a replacement collection component for X-Trace [23]. Our experimental results show that Hindsight imposes nanosecond-scale overhead when generating trace data, can scale to 55 GB/s of data per node, rapidly reconstructs traces when triggered, and coherently captures problematic traces (>99%), as well as related lateral traces, within 100 ms of identifying a symptom.

In summary, our paper makes the following contributions.

- We describe the retroactive sampling abstraction for capturing traces of symptomatic edge-cases.
- We present the design of Hindsight, a distributed tracing system that implements retroactive sampling. Hindsight is compatible with existing tracing APIs and can be transparently integrated with existing applications.
- We apply Hindsight on real-world use cases and show that efficiently collecting edge-case requests is practical.
- We evaluate Hindsight on multiple benchmarks and real systems, showing that it can achieve nanosecond-level overhead on trace data generation and handle GB/s data per node while collecting coherent traces.
- We illustrate that Hindsight is compatible and performs better than state-of-the-art tracing systems (X-Trace and Jaeger) with more efficient trace-data generation and lower overhead, while providing edge-case tracing.

2 Motivation

2.1 Edge-Case Troubleshooting

Consider the following three examples of real-world use cases UC1–UC3 of edge-case troubleshooting from prior work.

Error diagnosis (UC1). Hardware failures, component errors, exceptions, and programming mistakes abound in large production systems [73]. Application developers often play the role of detective, to identify root causes of errors. An error might only arise due to a specific, rare combination of factors and code paths exercised; the symptoms of a problem often manifest far from the root causes [22, 41, 45], and the potential root causes are manifold, perhaps combined software or hardware problems on many nodes or network links [35].

Tail-latency troubleshooting (UC2). Distributed systems track a wide range of high-level health metrics, such as API distributions, latency percentiles, resource utilization, and many others [33, 34]. An operator may observe an unusual metric jump, say the 99th percentile latency has spiked for some important API. However, knowing about the spike is not enough; the application developer must identify the specific service, code paths, or conditions that contribute to the peak to address any underlying problems [18, 38, 48].

Temporal provenance (UC3). Many modern distributed systems respond to requests through an architecture of loosely coupled microservices [62]. Application developers need tools for tracking queuing issues when the number of components in a distributed system is large [3–6, 8], since a request *R* exhibiting symptoms (*e.g.* prolonged queueing time) may not be the true culprit for the backlogged queue. Rather, the developer wants to follow the *temporal provenance* of *R* to determine *lateral traces* of other related requests with which *R* interacted through shared components and queues [72].

2.2 Distributed Tracing

Distributed tracing frameworks are in widespread use in both open-source [31, 52, 78] and major internet companies [34, 58, 64] to chronicle *end-to-end requests*. A trace is a recording of one request, and each trace contains spans, events, and annotations, along with timing and ordering, generated from every machine visited by the request. Compared to traditional logs and metrics, the key distinction of distributed tracing is that a trace captures the full end-to-end structural flow of request execution across all components visited.

Advantages. Distributed tracing is thus particularly useful for troubleshooting cross-component problems in large systems, since the request traces explicitly tie together the individual slices of work performed across different machines, enabling an operator to observe how the work done by one machine influences, and is influenced by, work done on others [23, 58, 64, 68]. Prior research on distributed tracing demonstrates a range of use cases, including common-case analyses centered on aggregate system behavior, distributions over data, and relationships between system components [34, 59, 60, 62].

Limitations. Since edge-case troubleshooting concerns rare and outlier system behavior, the symptoms and evidence of a problem might only manifest in a very small fraction of requests. Unfortunately for the operator, this sparsity may yield exceptionally few exemplar traces of edge-case behaviors and symptoms, owing to the design of modern distributed tracing frameworks. Let us look closer at how traces are captured before returning to this problem.

Current designs. Fig. 1 depicts a typical distributed tracing framework [34, 52, 58]. When a new request arrives at the application, the tracing framework assigns it a unique `traceId` (①). Every request is assigned a `traceId`, but not every request is actually traced; the framework sets a per-request `sampled` flag to indicate as such. From this starting point, the application then propagates the `traceId` and `sampled` flag alongside the request at the application level and includes them with all inter-process communication (②).

Any component that handles the request can generate trace data (*e.g.* spans, events) using the tracing framework's client library (*e.g.* OpenTelemetry [52])—trace data is only generated if `sampled` is set. Trace data gets explicitly annotated with the `traceId`, thereby associating the data with the request (③).

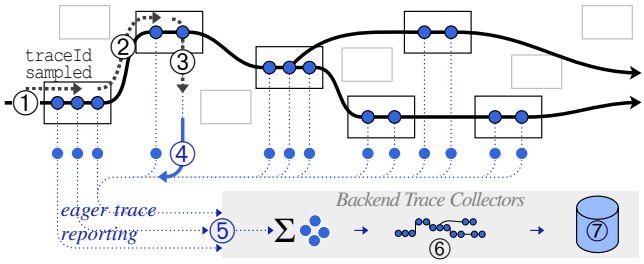


Fig. 1: **Distributed tracing** (§2.2). A request (solid black line) traverses system processes, depositing trace data that is eagerly ingested into the trace collector backends. End-to-end trace objects are constructed from trace data, processed, and stored in a database.

Ultimately there may be many components and machines that handled the request and contribute trace data. At the same time, many requests may execute concurrently (*e.g.* in different server handler threads), generating temporally-interleaved data with different `traceIds`.

The framework's client library eagerly enqueues, serializes, and transmits trace data (④) to its centralized backend collection infrastructure, or *backend* for short (⑤). The backend is distinct from the traced applications and is responsible for continually receiving, processing (⑥), and storing (⑦) trace data generated across all of the application's components. The backend uses the `traceId` to *join* data that was dispersed across many machines but belongs to same request into a single coherent trace object. The backend finally persists that trace object in a database if it decides to retain the trace.

Overhead vs. incompleteness. Traces can be detailed and produced at high volume, risking overheads. Traces at Google, for instance, are typically more detailed than debug-level logging [64]; each traced request at Facebook, similarly, generates several MBs of tracing data and approximately 1 billion traces are captured per day [34]. At high rates, tracing frameworks may encounter several potential bottlenecks: when generating data within the traced application (③); when transmitting trace data over the network (④); and in backend processing and storage (⑤–⑦).

To reduce overheads, the de facto practice is to capture fewer traces. Here, operating at the granularity of an entire trace maintains *trace coherence*: if a request is sampled, then the whole trace is kept including all data across all machines; otherwise nothing is kept. Coherent traces are essential for distributed tracing – a partial or fragmented trace has limited value in diagnosis [23, 29, 30, 64] because it loses the end-to-end visibility that makes the trace useful in the first place [58, 59, 68]. There are two main approaches for foregoing traces coherently: the system may decide to omit a request at ① before tracing and ingestion (*head sampling*) or the traces may be filtered after collection at ⑥ (*tail sampling*).

Head sampling reduces overheads by simply tracing fewer requests in the first place, *i.e.* by setting the `sampled` flag for only a small fraction of requests (①). By leaving `sampled` unset for the majority of requests, trace data will not be recorded for most requests, thus avoiding application overheads to generate data, ingestion overheads to transmit and process data, and storage overheads (③–⑦). Head sampling is widely used in practice; it is enabled by default in Jaeger [31] with a 0.1% sampling probability, and some production systems sample as few as 0.001% [34, 64].

Tail sampling is used to drop traces at the trace backends (⑥). Unlike head sampling, the application will still trace all requests and will incur all expenses of generating and ingesting the trace data (③–⑤). Tail-based sampling primarily allows backends to lower the trace storage costs by selectively dropping traces after combining them into trace objects but before committing them to storage [36, 37, 53].

2.3 Edge-Case Troubleshooting Troubles

Recall that edge-case problem symptoms only manifest in a small fraction of requests, which are undetermined until the problem takes place. We argue that current approaches are ineffectual at getting traces of edge-cases.

Head sampling sacrifices edge-cases. Indiscriminate sampling decisions made at the *beginning* of a request (①), while useful for curbing overhead, *cannot* know a priori whether a request will encounter a rare edge-case problem and should be traced. For edge-case troubleshooting this presents an obstacle: a low head-sampling probability (*e.g.* 0.1%) means a trace of the problem will exist with low probability (*i.e.* 0.1%). The developer may thus have reports that errors took place (UC1) yet the corresponding 'rare' requests were not sampled when those requests began—they lack the detailed cross-machine data necessary for finding the error's root cause. Likewise, the application's high-level metric monitoring may indicate a spike in end-to-end tail-latency (UC2); the developer is thus aware that these high-latency outliers exist, yet without a trace, they cannot localize the problem to a particular component or request class. The situation is even more problematic when investigating bottlenecked queues via temporal provenance (UC3): since each request was sampled independently, the tracing system will have only a vanishing probability that traces of *all* relevant requests in the queue were captured.

Tail sampling sacrifices overheads and scalability. Practitioners have long pointed out a discord between what traces are interesting and what traces get head-sampled [7, 9, 11, 54, 55]. Fortunately, many common edge-case symptoms, including error codes (UC1) and high end-to-end response time (UC2), can be recorded directly within the trace data itself. This enables tail-samplers to explicitly seek out edge-case traces, because at this point (⑥) they can directly inspect the constructed trace object. Today's tail-samplers support filtering traces based on span attributes or metrics, thereby targeting a range of outlier symptoms such as high tail latency, unexpected error codes, uncommon attributes, rare code paths, and undesirable behavior such as RPC retries [7, 10, 32, 40, 49, 53, 58, 67].

Tail-sampling entails enormous costs, however: they must

trace all requests and ingest all trace data in order to make informed decisions. Application latency and throughput can suffer if tracing libraries lack optimization (*e.g.* 2× throughput reduction using OpenTelemetry tail-sampling (§6.4)). Ingesting all traces consumes substantial network bandwidth between applications and collectors, interfering with latency-sensitive application traffic (*e.g.* up to 200 MB/s per node (§6.4)). Tail-sampling demands large backend infrastructure investment, deploying enough collectors to receive and process all incoming traces (*e.g.* even one chatty RPC server can overwhelm an OpenTelemetry collector (§6.4)). Even assuming perfect horizontal scaling, tail-sampling requires *e.g.* 100× the collector capacity of 1% head-sampling. Lastly, tracing frameworks are also not robust to bottlenecks mid-way through ingestion (*e.g.* network backpressure) and quickly lose trace coherence when overloaded.

Practitioners sacrifice edge-cases. The justified pragmatism of avoiding large overheads means that head sampling reigns supreme in real-world distributed tracing deployments [21, 26, 30, 34, 64]. Even tail-sampling features of commercial products have low thresholds on data ingestion (*e.g.* <350 kB/s per host [65], <34 spans/s per host [50], <6 MB/s per collector [39]) after which vendors will automatically enable head-sampling or incoherently drop spans. Ultimately, the operator who wishes to troubleshoot edge cases is left unfulfilled.

3 Approach

Hindsight aims to overcome today's trade-off between overheads and edge cases. Our goal is to enable practitioners to target edge-case traces with the flexible criteria of tail sampling, while retaining overheads similar to that of head-sampling, *i.e.* without high application overheads or substantial additional backend infrastructure. We now describe several insights that lead us to Hindsight's *retroactive sampling* approach.

It is not expensive to generate trace data. We don't know a priori whether a request will be an interesting edge-case; only after symptoms manifest. Paradoxically, once we observe symptoms, it is too late to just enable tracing from that point on, as we have already missed the events that led to the anomaly. The only sure-fire way of obtaining coherent traces for any edge-case is to record trace data from the very beginning of the request, for every request.

Tail-sampling does just that—with high overheads and steep infrastructure costs. However, these costs are primarily because today's tracing frameworks tightly couple trace generation with trace ingestion. Ingesting data is expensive, incurring network and backend infrastructure costs. Generating data into local memory is not—outside of distributed tracing, *e.g.*, we observe new technology like Intel PT can generate 100–200 MB/s of processor telemetry per core at 5–15% runtime overhead [28]; likewise method-tracing techniques for Android applications exhaustively record all function entries and exits with <1 ns per tracepoint and <3% runtime overhead [43]. We believe that comparable overheads should be possible for distributed tracing. With careful client library design, applications should be able to generate detailed trace data locally into memory, in anticipation of that data being useful if a problem occurs.

Retroactive sampling: nodes generate, but do not ingest, all trace data.

Symptoms are locally observable. Although root causes are many, varied, and difficult to predict, the same is not true of *symptoms* of problems. For example, error codes, tail latency, and exceptions are easily-observed indicators of potential problems. Many symptoms are localized, programmatically detectable, and manifest quickly at some point during or shortly after a request was served [27, 53, 58]. For example, tail sampling techniques, by definition, require that some span in the trace was explicitly annotated with the symptom of an anomaly, and typically wait only 10 seconds to accumulate trace data [51, 53]. For these common cases it is not necessary to ingest and construct full trace objects when the symptom is so readily detectable at the source. Moreover, since symptoms can be detected independent of traces in the first place, we do not need the expensive indirection of writing symptoms into trace data only to later extract and filter them. We believe that the key to capturing edge-cases is to decouple detection of symptoms from collection of traces.

Retroactive sampling: applications embed **triggers** that programmatically observe symptoms and signal after-the-fact that a trace is an edge-case.

Triggers are local but trace data is distributed. Prior distributed tracing frameworks ingest traces eagerly. We instead believe that traces should be lazily ingested, only in response to a trigger fired at some point during or soon after a request. However, triggers are local – only one machine might detect a symptom, yet the trace data for the request will be dispersed across memory of all machines that serviced it. To splice together a coherent end-to-end trace, all of these other machines need to learn of the trigger and send their slice of the trace to the backend collectors. To identify and notify all relevant machines of a trigger, we thus need the ability to *back-track* the end-to-end path of a request.

Retroactive sampling: requests propagate and deposit **breadcrumbs** so triggers can be shared with all relevant machines.

Trace data will eventually expire. Applications generate trace data into local memory where it incurs no further processing. We only send trace data to collector backends if a trigger fires. However, we cannot predict *when* a trigger might fire – even if a request has finished executing locally, we cannot easily know that the request isn't still executing on some other machine(s) or that a trigger won't fire remotely. Thus, trace data must remain in memory on each machine indefinitely. Over time this will fill memory and eventually we will need to free up space. The intuitive choice is thus

to expire trace data for the least-recently-seen request. We call the implicit time duration between generating data and overwriting it the ***event horizon***. We believe that retroactive sampling should not require a large event horizon – as low as tens of seconds is reasonable – because triggers are automatic and shared quickly. In the majority of cases a machine should learn of a trigger within a matter of seconds or milliseconds. Thus retroactive sampling should be feasible even with large and detailed traces or constrained memory.

Retroactive sampling: triggers are best effort; we assume we will see triggers quickly if at all.

4 Design

Overview. Hindsight is a distributed tracing framework that implements retroactive sampling. Whereas typical distributed tracing frameworks eagerly ingest trace data, Hindsight lazily ingests data only after a trigger, thus allowing retroactive sampling of edge-case traces without paying the overhead costs of ingesting all trace data. Hindsight remains compatible with existing head-sampling and tail-sampling policies. Hindsight trivially implements head-sampling policies by firing an immediate trigger upon a positive head-sampling decision (or if the `sampled` flag is set). Hindsight is opaque to backend trace collectors and tail-sampling policies, and existing ingestion pipelines require no changes. Likewise, Hindsight is transparently compatible with existing OpenTelemetry APIs and instrumentation [52], and piggybacks breadcrumbs with OpenTelemetry's context propagation.

Walkthrough. Fig. 2 shows a high-level diagram of Hindsight's main components.

① On request arrival (solid black line) Hindsight generates a unique `traceId` and thereafter propagates it alongside the request, as done by existing frameworks (§2.2).

② Applications record trace data (*e.g.* events, spans) using Hindsight's `tracepoint` client API. This leaves the request's trace data scattered across the machines it visited.

③ A Hindsight ***agent*** runs on each machine to manage trace data. Hindsight agents do not inspect, process, or eagerly report trace data to backends – instead, agents index metadata by `traceId` and await further instruction. For most traces nothing further happens, the trace is not reported, and agents eventually evict old trace data.

④ If an application node observes an outlier symptom (*e.g.* erroneous response, high latency, or a bottlenecked queue) it invokes Hindsight's trigger API and passes the request's `traceId`.

⑤ The local Hindsight agent receives the triggered `traceId`. The full trace remains dispersed across many Hindsight agents, so the local agent informs Hindsight's logically centralized *coordinator* service of the `traceId`. Hindsight's coordinator recursively contacts the set of machines that serviced this request, soliciting breadcrumbs deposited by the request at each machine; a breadcrumb is a pointer to another machine involved in the request (*e.g.* to the RPC caller or callee).

⑥ Each agent contacted will set aside its slice of data belonging to the `traceId`, and asynchronously send it to the backend collector.

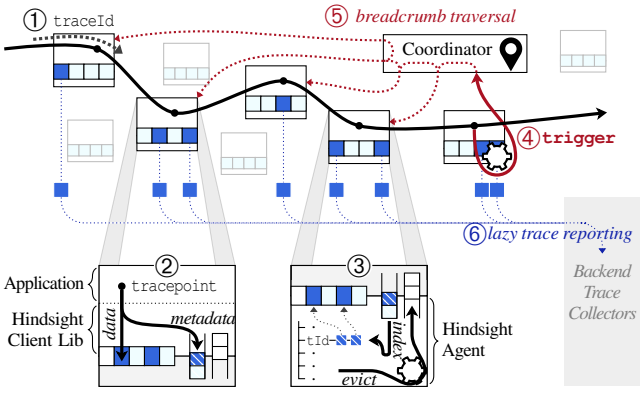


Fig. 2: **The end-to-end lifecycle of a trace in Hindsight** (§4).

Design decisions. Hindsight is most shaped by three key design choices. First, to prioritize trace coherence as a primary objective throughout the architecture. Second, to maintain an efficient data and control plane split to enable tracing 100% of requests. Finally, to support lightweight programmatic trigger mechanisms.

4.1 Trace Coherence

Coherence is a top-level requirement for distributed tracing (§2.2). As soon as any machine drops data for a trace, the trace is incoherent and effectively useless for troubleshooting. Hindsight's design avoids incoherence in several places.

At ③, agents continually evict old trace data to free up space for new data. Agents do this atomically at the granularity of a trace; there is no point in only dropping part of a trace. However, for a single trace, its data is non-contiguous and fragmented in memory. Agents carefully organize and index metadata about where each trace's data resides and do not simply evict old data in a LIFO manner.

At ⑤, the coordinator must contact all agents that handled a request before those agents overwrite their slice of trace data. Breadcrumbs are a lightweight and scalable solution – the coordinator recursively follows breadcrumbs and only contacts the specific agents known to have serviced the request. This approach takes only a few milliseconds in our evaluation. Breadcrumb traversal is independent of reporting the trace data; agents set aside and asynchronously send trace data to the collector backends after learning of a trigger.

At ⑥, agents can potentially experience network congestion or backpressure from the collector backends, such as in response to a trigger-happy application that fires too many triggers and causes a backlog of unreported trace data on many, or all, agents. Eventually even triggered data must be dropped. Hindsight agents do not drop data arbitrarily (*e.g.*,

skipping a full queue) because different agents would then tarnish different victim traces—it only takes one agent dropping its slice of a trace to render the remaining data on other agents practically worthless due to incoherence. Instead, in several places agents use priority queues, with priority determined by consistent hashing of `traceIds`. A given `traceId` will enjoy the same priority across all agents and queues, and the same traces will be dropped by all agents in the face of a bottleneck.

Finally, at ④, applications may fire multiple different triggers for a diverse range of symptoms, using a developer-provided `triggerId` to distinguish different trigger types. Hindsight will prevent a profuse trigger from stifling trace collection of other, low-frequency triggers: agents implement weighted fair sharing for reporting and evicting trace data, with user-defined weights and rate-limits for each `triggerId`.

4.2 Efficient Data Management

Lazy ingestion significantly reduces the volume of trace data sent from agents to the backend trace collection infrastructure. However, within an individual machine, retroactive sampling requires the application generate trace data into local memory for *all* requests (③). The most sensitive performance bottleneck for Hindsight is thus between client applications generating data (`tracepoint`) and the local Hindsight agent that manages trace metadata. Our design establishes a clear split between *control* and *data* activities, which congregates general-purpose data and efficiency in the data plane, and embeds all logic in the control plane.

Data plane. Hindsight's data plane is concerned with efficiently writing trace data from client applications. Using `tracepoint`, applications write trace data to a large shared memory pool subdivided into *buffers*. Different threads write to different buffers; each buffer may only belong to one `traceId` at a time, and threads acquire new buffers when full or when the active `traceId` changes. Consequently, the buffer pool is not consumed sequentially and a single trace may be fragmented across several non-contiguous buffers.

Control plane. Hindsight's agent process encapsulates control plane activities, continually circulating *metadata* about buffers to the application, via two shared memory queues. Applications poll for available buffers and push full buffers; agents poll for full buffers, index metadata of full buffers grouped by `traceId`, and push evicted buffers back to the application. Agents receive triggers and communicate with Hindsight's coordinator, manage breadcrumbs linking the trace data that is strewn across many agents, extract triggered trace data, and report data asynchronously to the backend trace collection infrastructure. Hindsight's control and data distinction yields an efficient agent implementation because agents only touch metadata.

4.3 Triggers

Applications initiate retroactive sampling via Hindsight's trigger API (⑤). In the common case, symptoms are easy to detect and localize: top-level error codes; high latency; increased queue time. Such symptoms can be readily recognized and cheaply computed without the trigger mechanism needing the trace data itself. For example, this may entail adding a trigger call within a service's exception handler, or after checking for outlier latency upon a request's completion. Hindsight provides a library of automatic triggers based on metric percentiles, categorical features, and exceptions. All of our use cases (UC1–UC3) can be implemented using Hindsight's autotriggers. Likewise all existing tail-sampling policies can be implemented using autotriggers, as span-local attribute and metric filters directly translate to metric and categorical autotriggers.

By separating triggers from traces, developers can also implement custom symptom detectors to explicitly decide the conditions for triggering. This further leads to a straightforward integration of triggers into existing metric-monitoring and outlier-detection systems regardless of their architecture.

Lateral traces. Outlier behavior may not map directly to a single request; instead there may be several other related *lateral* requests. For example, to diagnose a bottlenecked queue (UC3), a trigger needs to capture traces for the previous N requests to understand what led to queue buildup [72]; to diagnose a write-ahead log, we desire all requests blocking on a log sync [4, 8]; to diagnose resource contention we require all requests contending for a slow disk or network [3, 5, 6]. By separating triggers from traces, we enable more comprehensive trigger conditions based on factors beyond just a single trace, and triggers that can capture multiple related traces simultaneously. Hindsight enables an application to atomically trigger a group of related lateral `traceIds`; internally Hindsight will ensure that the group as a whole is coherently collected. By comparison, tail sampling cannot easily express cross-trace triggers or sample lateral traces, because `traceId`-based sharding in collector backends is fundamentally at odds with sharing state between traces.

5 Implementation

We have implemented Hindsight's client library in ≈4 KLOC of C and Hindsight's agent and coordinator in ≈5.5 KLOC of Go. We chose C for dataplane efficiency and Go for its ease of use for the more complex control plane logic.

5.1 Data Plane Buffer Pool

Each Hindsight agent pre-allocates a fixed-size *buffer pool* in shared memory for applications to directly write trace data. Hindsight logically subdivides the buffer pool into fixed-size buffers (default 32 kB). Client applications write trace data to buffers via Hindsight's client API. The agent process does not touch data in the buffer pool except when reporting triggered traces. At each point in time, a buffer can only contain trace data of a single request; no two different requests will write trace data to the same buffer at the same time. A single trace will thereby comprise (1) multiple non-contiguous buffers on

`begin(traceId)`	Request begins in the current thread.
`tracepoint({payload})`	Record data for the current trace; `payload` is of arbitrary size in bytes.
`breadcrumb(address)`	Adds a breadcrumb to the current trace, pointing to some other node `address`.
`serialize()`	Obtain the current `traceId` and a `breadcrumb` to the current node.
`end()`	Request ends processing in current thread; flush and remove buffers.
`trigger(traceId,triggerId lateralTraceIds...)`	Instruct Hindsight to collect `traceId` and zero or more `lateralTraceIds`

Table 1: **Hindsight client API.** Applications can invoke the API directly, or indirectly using Hindsight's OpenTelemetry [52] tracer.

each agent and (2) many buffers scattered across numerous agents. Buffers are the granularity of data management within Hindsight. Within clients and agents, a buffer is addressable by its `bufferId`—its offset into the buffer pool.

5.2 Client Library

Table 1 outlines Hindsight's client API. Applications can interact with this API directly, or use Hindsight's OpenTelemetry tracer which acts as a wrapper.

Writing trace data. When a request begins executing in a thread, it must call `begin`; subsequently it may call `tracepoint` an arbitrary number of times; and finally when it completes executing in a thread, it must call `end`. This usage pattern is typical of distributed tracing frameworks. The `tracepoint` function accepts an arbitrary byte payload if called directly; conversely Hindsight's OpenTelemetry tracer serializes trace events as payload. Hindsight internally maintains thread-local state including the current `traceId` and a pointer to a buffer. `tracepoint` writes directly to the thread-local buffer without synchronization. Synchronization is only required when acquiring or returning buffers; these operations touch shared-memory queues but are infrequent. A buffer is acquired during `begin`, returned during `end`, and replaced when filled.

Communicating with agents. The client library acquires `bufferIds` by polling a shared-memory *available queue*; if the queue is empty clients immediately return and instead write trace data to a special 'null buffer' that is simply discarded. When the client fills a buffer, it writes its `traceId` and the `bufferId` to a shared-memory *complete queue*. The agent continually drains the complete queue, and likewise continually returns fresh buffers to the available queue. Shared memory queues are lock-free and support batch operations; using batch operations, agents are robust to queue contention from multiple client writer threads.

This paired channel design forms a natural separator between control and data with two desirable properties: (1) queues only communicate metadata—a single integer `bufferId` represents, by default, a 32 kB buffer; (2) communication is infrequent, occurring only when buffers are filled or a thread switches over to execute a different request, thereby minimizing synchronization. From the client library's perspective, it cheaply and blindly writes trace data into shared memory and forwards only the control metadata to agents; conversely agents are agnostic to buffer contents—they do not inspect data in the shared memory pool and use only the metadata communicated via the complete queue.

`PercentileTrigger(`p`)`	Clients call `addSample(traceID, measurement)`. Trigger fires for measurements >percentile p. (*e.g.* high latency or resource consumption)
`CategoryTrigger(`f`)`	Clients call `addSample(traceID, label)`. Trigger on categorical data that is less frequent than threshold f (*e.g.* rare API calls or attributes)
`ExceptionTrigger`	Trigger on an exception or error code
`TriggerSet(`T`,`N`)`	Tracks the most recent N `traceIds` and includes as `lateralTraceIds` when T fires.

Table 2: **Hindsight autotrigger API** can automatically trigger traces based on certain conditions.

Depositing breadcrumbs. A breadcrumb is an address of a Hindsight agent. When a request arrives at a node, it carries the breadcrumb of the previous node. During trace context deserialization, the `traceId` and breadcrumb is written to a shared memory breadcrumb queue. Agents poll this queue and index breadcrumbs alongside buffer metadata. Agents do not forward or act upon breadcrumbs until a trace is explicitly collected with a trigger. When a request departs a node, it takes that node's breadcrumb. Clients can additionally establish forward-breadcrumbs to a named destination node prior to communication. By following breadcrumbs, we can reconstruct the full request graph starting from any node, including for requests with arbitrary concurrency and fan-out.

Triggering trace collection. Applications initiate trace collection by invoking trigger, which writes the `traceId`, `triggerId` and zero or more `lateralTraceIds` to a shared-memory trigger queue. In addition, Hindsight will propagate the fired trigger with the request similar to the `sampled` flag (cf. Fig. 1) so that later nodes immediately learn of the trigger.

A developer can implement custom outlier detection and invoke trigger directly, or they can make use of Hindsight's autotrigger library (Table 2), a separate collection of triggers that track simple conditions over time and automatically invoke trigger when a condition is met. `TriggerSet` is noteworthy as a building block for lateral tracing; it includes N most recent traces whenever T fires.

5.3 Agent

Trace index. The trace index is a map of metadata, keyed by `traceId`. The metadata for a `traceId` includes a list of `bufferIds` and a list of breadcrumbs. Agents also maintain metadata of the triggers that have fired. Agents continually update the trace index with recently-written buffers, by polling `traceIds` and `bufferIds` from the complete queue. The agent will evict traces when the index exceeds a threshold of buffer pool capacity (default 80%) by removing the least-recently used untriggered `traceId` and returning all of its `bufferIds` to the available queue.

Local triggers. Agents poll the local trigger queue and immediately forward triggers to the coordinator. Agents include the breadcrumbs of the triggered `traceId`, enabling the coordinator to begin recursively disseminating the trigger to other agents. Meanwhile the agent schedules the trigger to be reported. In the case of a spammy local trigger, if the trigger exceeds a per-`triggerId` rate-limit, the agent will immediately discard the trigger instead of forwarding and scheduling it.

Remote triggers. Agents receive remote triggers fired by other agents via the coordinator. To facilitate rapid trigger dissemination, the agent immediately responds to a remote trigger by providing any breadcrumbs it has for the `traceId` and `lateralTraceIds`. Unlike local triggers, agents do not rate-limit remote triggers—they are all scheduled for reporting.

Reporting traces. When a trigger is scheduled for reporting, its `traceId` and lateral `traceIds` can no longer be evicted by the regular buffer eviction cycle. The trigger is inserted into a per-`triggerId` reporting queue. In the normal case when an agent is not backlogged, the reporting queue will be empty. The agent asynchronously pulls triggers from the queues; reads buffers of the `traceId` and `lateralTraceIds` from the buffer pool; sends the buffer contents to the backend collectors; and finally returns the `bufferIds` to the available queue. A trace remains triggered even after reporting its data, in case the request is still generating trace data locally.

Ignoring triggers during overload. If the network or backend collectors are overloaded, reporting queues in an agent can fill up. During overload, the agent continues to report traces as described above for the normal case. The agent implements weighted fair queueing over the reporting queues and supports global and per-`triggerId` reporting rate limits. From a reporting queue, the agent dequeues the *highest-priority trigger*, by using consistent hashing of `traceId`, and reports its data as described above for the normal case.

Simultaneously, past a configured threshold, the agent must begin abandoning triggers to free up buffers. Abandoning a trigger entails removing it from its reporting queue and returning buffers to the available queue. Agents coherently select the *lowest-priority trigger* to abandon, by using the same consistent hashing of `traceId`. In the case of multiple reporting queues, agents will ensure that a well-behaved `triggerId` is not impacted by a spammy `triggerId`: agents implement weighted max-min fair-sharing across reporting queues to choose a queue from which to drop triggers.

Trigger priority ensures coherence during overload. Reporting queues are priority queues that use consistent hashing of `traceId` to determine priority. Across all agents, a given `traceId` will enjoy the same priority relative to other `traceIds`. Thus if multiple agents experience overload, they will coherently bias towards reporting the same high-priority `traceIds` and abandoning the same low-priority `traceIds`.

6 Evaluation

We now evaluate how effectively Hindsight overcomes the fundamental problem of head-based tracing methods in examples (UC1)–(UC3) and meets the goals of retroactive sampling to provide lightweight and effective request tracing.

Systems. We evaluate Hindsight on three distributed systems. To validate our motivating use cases (UC1–UC3), we integrate Hindsight with the Hadoop Distributed File System (HDFS) [63](with a ≈300 LOC JNI-based Java client library) and the DeathStar Social Network Microservices Benchmark (DSB) [24]. To assess Hindsight at greater scale and load, we develop a flexible, configurable RPC benchmark called **MicroBricks**.

MicroBricks is a microservice benchmark written in ≈3 KLOC C++ using gRPC's high-performance `async` library. A MicroBricks deployment comprises a topology of RPC services such that each client request will traverse multiple services. A call to a service will execute for some amount of time, then concurrently call zero or more other RPC services with some probability. Each service is independently configured with its own set of APIs, each with their own execution times, child dependencies, and child call probabilities. We evaluate using several different topologies. In particular, we use Alibaba's microservice trace dataset [42] to derive realistic topologies by calculating per-service execution time distributions, service dependencies, child call probabilities, and client workloads.

Baselines. We configure OpenTelemetry [52] with Jaeger [31] under head-sampling (1% unless indicated) and tail-sampling.

Instrumentation. We instrument MicroBricks with OpenTelemetry to create spans and events for RPC calls and child calls. We use DSB's existing OpenTracing instrumentation and add support for Hindsight. We use Hadoop's existing X-Trace instrumentation [23] and update X-Trace to write its trace data to Hindsight.

Summary. Our experiments demonstrate the following:

- Hindsight effectively addresses the overhead vs. edge-cases trade-off faced by existing tracing frameworks.
- Hindsight captures relevant edge-case traces across real use-cases (UC1–UC3).
- Hindsight is lightweight and not a bottleneck for client applications, unlike OpenTelemetry [52] and Jaeger [31]. Hindsight's trace API imposes nanosecond overheads; Hindsight's impact on end-to-end application latency and throughput is <3.5% when tracing 100% of requests and generating >200 MB/s of trace data per node.
- Hindsight's control/data split provides up to 55 GB/s write throughput.

6.1 Overhead vs. Edge-Cases

In this experiment, we evaluate Hindsight in a large-scale setting with a realistic microservice topology derived from

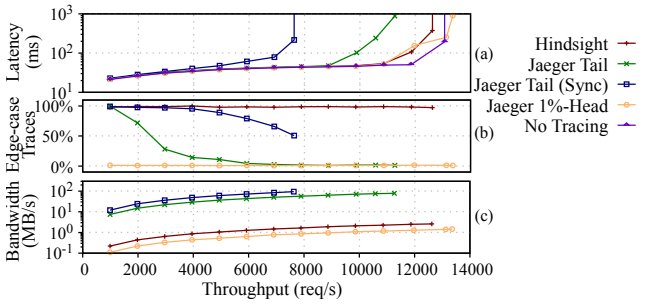


Fig. 3: **Overhead vs. edge-cases** on a 93-service Alibaba MicroBricks topology with 1% edge-cases (§6.1). For different tracing configurations we show: (a) application end-to-end latency-throughput curves; (b) the rate of coherent edge-trace cases captured; and (c) network bandwidth.

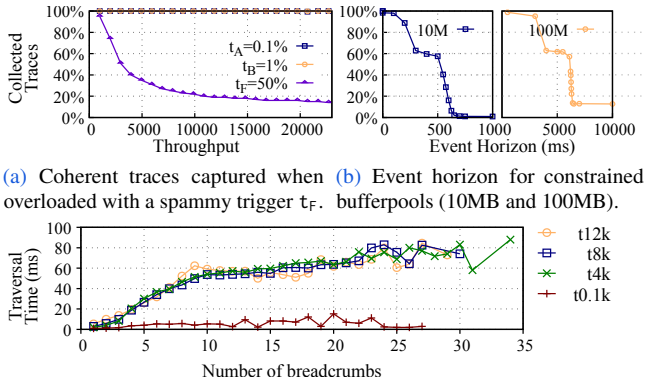
Alibaba request traces [42]. We show that Hindsight overcomes the limitations of head-sampling and tail-sampling.

We deploy MicroBricks with a 93-service Alibaba topology in a 544-core private cluster (comprising 10×Dell R920 48-core 1.5 TB machines and 4×Dell M620 16-core 256 GB machines). We deploy each service in a separate container. We use separate machines to **(i)** generate workload and **(ii)** run the OpenTelemetry collector/Hindsight coordinator+collector.

To directly control the number of edge-case traces, we randomly decide with low probability (1%) to designate a request an edge-case when it completes (later experiments consider autotriggers). We annotate the root span of edge-cases with an additional attribute so that tail-sampling can filter traces on this attribute. Hindsight directly fires a trigger for edge-cases from within MicroBricks. We repeat the experiment multiple times, analyzing results under four tracing configurations:

Head sampling (Jaeger 1%-Head). Fig. 3a shows the average request latency and throughput as we vary the offered load from 0 to 14,000 requests/sec (r/s). Jaeger 1%-Head has comparable peak throughput and latency as No Tracing, since it traces only 1% of requests, thus amortizing the tracing overhead. Fig. 3b plots the percentage of coherent edge-case traces captured per second. Since head-sampling cannot discriminate, it only captures ≈1% of all edge-case traces, peaking at 1.64 per second. Fig. 3c shows the network bandwidth consumption between application nodes and the OpenTelemetry collector. With few requests being traced, Head-sampling only consumes a maximum of 1.4 MB/s of network bandwidth.

Tail sampling (Jaeger Tail). Tail-sampling imposes more burden on the traced application than head-sampling, attaining 14% lower peak throughput (Fig. 3a). At low load (1,000 r/s), tail-sampling successfully captures ≈100% of edge-case traces, at 9.9 per second (Fig. 3b). However, a load of just 2,000 r/s is sufficient for clients to encounter backpressure from the network and the OpenTelemetry collector, and they begin incoherently dropping spans: at 2,000 r/s only 71% coherent edge-cases are captured; at 3,000 r/s only 28%; and so on. Tail-sampling rapidly deteriorates and at peak load captures *fewer* coherent edge-case traces than head-sampling



(a) Coherent traces captured when overloaded with a spammy trigger t_F. (b) Event horizon for constrained bufferpools (10MB and 100MB).

(c) Breadcrumb traversal time as trace size varies, triggering 0.1% (t0.1k) or 50% (t4k, t8k, t12k) traces on different workloads.

Fig. 4: **Scalability and overload.**

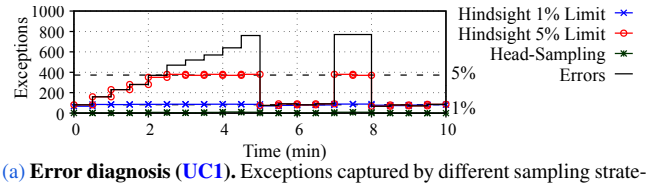
(1.44 edge-cases/s), because 98.8% of captured traces are incoherent. Tail-sampling consumes up to 78 MB/s of network bandwidth (Fig. 3c).

Tail sampling (Jaeger Tail Sync). Jaeger clients asynchronously send spans to OpenTelemetry collectors, and as we just observed, drop spans when client-side queues fill up. We repeat the experiment with a synchronous variant, whereby clients send spans to OpenTelemetry synchronously. Backpressure then manifests as additional critical-path request latency. This approach inevitably increases request latency and reduces peak throughput by 42% (Fig. 3a). However, we can observe the collector ultimately captures more edge-case traces, peaking at 47 edge-cases per second at 6,000 r/s (Fig. 3b) and 72.2 MB/s of network. Beyond this, the OpenTelemetry collector is saturated and cannot process a higher rate of traces; it begins indiscriminately dropping incoming spans, reducing the fraction of coherent edge-case traces.

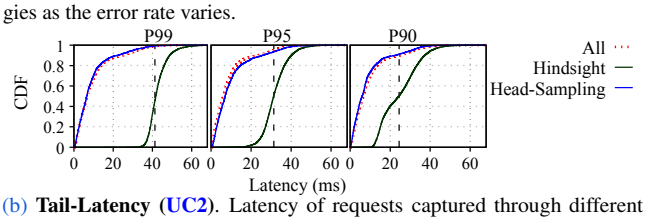
Hindsight. Hindsight achieves comparable peak throughput to No Tracing (<3.5%), and minimal impact on request latency below peak load (Fig. 3a). Hindsight captures 99–100% of edge-case traces at all throughputs (Fig. 3b). Hindsight consumes a maximum of 2.6 MB/s of network bandwidth since only edge-case traces are being collected (Fig. 3c).

6.2 Scalability and Overload

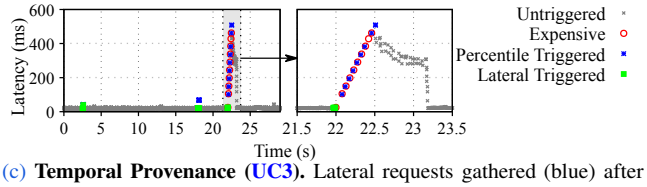
We now focus on two aspects of Hindsight's scalability: its breadcrumb traversal mechanism and its ability to rate-limit spammy triggers. We deploy the 93-service Alibaba topology as described in §6.1. To reach a higher request and trace throughput, we scale down the computation performed at each service and increase offered load up to 28,000 r/s. We install three triggers with probabilities t_A=0.1%, t_B=1%, and t_F=50%. t_F represents a faulty trigger—it fires for 50% of requests and thereby adds substantial load to Hindsight's breadcrumb traversal mechanism. We rate-limit Hindsight's collector bandwidth to 1 MB/s per agent to backlog the agents and inhibit Hindsight's ability to collect traces; thus t_F triggers far more traces than Hindsight can collect.



(a) **Error diagnosis (UC1).** Exceptions captured by different sampling strategies as the error rate varies.



(b) **Tail-Latency (UC2).** Latency of requests captured through different sampling strategies with different tail-latency triggers (left to right).



(c) **Temporal Provenance (UC3).** Lateral requests gathered (blue) after triggering on slow requests (red) due to an overfull queue in HDFS.

Fig. 5: **Hindsight applied on use cases UC1–UC3** (see §2.1).

Coherent rate-limiting. Fig. 4a plots the percentage of coherent traces captured for t_A,t_B and t_F as the offered load increases. Throughout the experiment, Hindsight captures approximately 100% of traces triggered by t_A and t_B, since they fire infrequently. By contrast, t_F triggers far more traces than can be collected. In absolute terms, Hindsight collects ≈2,000 coherent traces per second throughout the experiment, with t_F using capacity not used by t_A and t_B. Thus, higher request rates result in more traces dropped for t_F in both relative and absolute terms.

Breadcrumb traversal. Fig. 4c plots the average breadcrumb traversal time based on the trace size – *i.e.* the number of Hindsight agents that were recursively contacted. We show results for four experiment iterations and label them based on their approximate trigger rates: `t12k`, `t8k`, and `t4k` correspond to triggering 50% traces on 24k, 16k and 8k r/s workloads (≈12k, 8k, and 4k triggers/second respectively). To compare to a non-overloaded setting we also include `t0.1k`, a 12k r/s workload from §6.1 (≈0.1k triggers per second). Traversal time is elevated for `t12k`, `t8k` and `t4k` (up to 86 ms) since spammy triggers substantially increase the load on Hindsight's coordinator. Conversely, traversal time for `t0.1k` is <13 ms since triggers are relatively infrequent. For each experiment, traversal time increases with trace size, but sub-linearly since breadcrumbs can be gathered concurrently from different branches in requests that have fan-out. However, even under the extremely overloaded circumstance, the longest traversal time, which is less than 100 ms, is far smaller than the event horizon as described in the following section and thus is still manageable.

Event horizon. We lastly measure Hindsight's event horizon. Here, we introduce a delay when an agent receives a local trigger. We vary the delay added to triggers and measure how many coherent traces are ultimately collected. At a certain point, triggers will have too much delay and trace data will have been evicted before the trigger even fires. Fig. 4b plots the percentage of coherent traces captured for t_B as we vary the trigger delay. We repeat this experiment with small buffer pools (100 MB and 10 MB per agent) to exacerbate the event horizon effect. Even a 10 MB buffer pool can capture nearly 100% coherent traces in the absence of added delays, but a 500 ms delay drops coherence to 58% and at 600 ms, coherence is <20%. A larger buffer pool improves the tolerance to delays: with a 100 MB buffer pool, coherence surpasses 90% with up to 3s delay, but drops to <20% by 6.4 s. In practice, we believe our default 1 GB pool is a reasonable choice, bringing an event horizon around 1 minute.

6.3 Case Studies

We now turn our attention to the case studies introduced in §2.1, and demonstrate how Hindsight's local triggers are able to support these use cases.

Error diagnosis (UC1). We deploy DSB Social Network, a microservice system with 12 microservices and 17 backends [24], on 13 CloudLab `c6320` nodes [20]. We add an `ExceptionTrigger` from Hindsight's autotrigger library to the ComposePostService, and run DSB's default workload with 300 r/s[1]. We randomly inject exceptions in the ComposePostService module, with exception rates ranging from 1% to 10%. We repeat the experiment twice and rate-limit Hindsight's collector to approximately 1% and 5% of the total trace data generated by the experiment. Fig. 5a plots the exception rate, and the number of coherent exceptional traces captured, for each 30 s time window. When there are few exceptions, Hindsight captures all traces; when the exception rate exceeds collector bandwidth, Hindsight coherently captures as many traces as possible within this limit.

Tail-latency (UC2). We add a `PercentileTrigger` from Hindsight's autotrigger library to the ComposePostService module in the same setting as above, invoking `addSample` at the end of each ComposePost RPC call and providing the measured RPC duration. We set p to 99, 95, and 90, as different thresholds for tail latency. We inject 10% requests at random with 20–30 ms latency. Fig. 5b plots the latency distribution of requests captured by different strategies; the vertical dotted lines mark the tail-latency percentile threshold. Hindsight is able to specifically target traces with high-percentile latency. By contrast, head-sampling is random and thus its captured latency distribution resembles that of all requests – useful for aggregate analysis but not for edge-case troubleshooting. We note that Hindsight does not sacrifice this aggregate analysis use-case; it supports both simultaneously (cf. §6.1).

Temporal provenance (UC3). We add a `QueueTrigger`

[1]We measure a maximum attainable DSB throughput of ≈350 r/s.

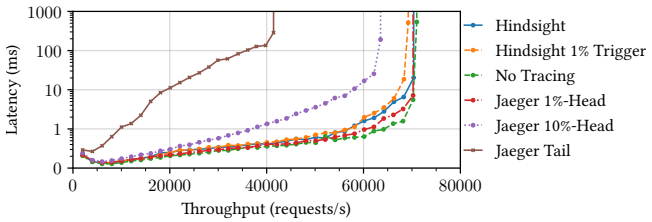


Fig. 6: **End-to-end latency and throughput** for a 2-service MicroBricks topology configured with various tracers, showing minimal application impact for Hindsight despite tracing 100% of requests.

from Hindsight's autotrigger library to the HDFS NameNode queue — the `QueueTrigger` combines a `TriggerSet` with a `PercentileTrigger`, parameterized to capture $N = 10$ most recently dequeued lateral requests when 99.99th percentile queueing latency is observed. We deploy HDFS on 10 machines (8 DataNodes, 1 NameNode, and 1 client) and run a Hindsight agent on each machine. We run a closed-loop workload of random 8 kB reads with 10 concurrent requests.

Fig. 5c (left) shows NameNode queue latency over time. We inject a burst of 10 expensive `createfile` requests 21 seconds into the trace that briefly saturate the queue—Fig. 5c (right) zooms in on this time window. The figure shows high-latency requests (•), requests that fire the autotrigger (**X**), and the additional lateral requests that were triggered to Hindsight (**X**). The first expensive request occurred at 22 seconds, followed by a pause while it was executed. Upon dequeuing the subsequent `read8k` request, `QueueTrigger` fired due to high queue latency, and Hindsight retroactively sampled the 10 prior traces leading up to the trigger. The sample included the culprit expensive request. Overall, all 10 expensive requests were sampled, 8 unrelated requests prior to the first expensive request, and 9 additional `read8k` requests. Moreover, several intermittent latency spikes occurred unrelated to the experiment (Fig. 5c, left), which Hindsight also captured; upon investigation, these were due to garbage collection.

Unlike UC1 and UC2, temporal provenance is unsupported in existing tail-samplers. Moreover, temporal provenance is fundamentally difficult to support with tail-sampling due to scalability issues. Temporal provenance requires knowledge of lateral traces (*e.g.* the 10 previous traces); by implication those traces must all route to the same collector instance. However in practice, tail-sampling necessarily uses `traceId` for routing decisions – thus related traces may arrive at different, oblivious collectors.

6.4 Hindsight Performance

End-to-end application overheads. Hindsight generates trace data for all requests; thus low overheads are a key goal of Hindsight's design. In this experiment, we measure the impact of Hindsight on end-to-end application latency and throughput. We deploy a two-service MicroBricks topology with a 100% call probability from the first service to the second. To highlight tracing overheads, neither service performs additional compute. We vary the offered load and measure end-to-end request latency and throughput (§A.1).

API Call	T=1	T=4	T=8
`begin`	72.7	194.8	237.9
`end`	70.7	205.8	216.6
`Category(.01)`	45.8	44.9	46.7
`Percentile(99)`	275.3	293.5	306.9
`Percentile(99.9)`	407.1	441.9	512.2
`Percentile(99.99)`	629.4	875.8	1134.0
`TriggerSet(10)`	6.57	44.1	52.2

API Call	T=1	T=4	T=8
`tracepoint`	7.9	8.4	8.6
`tracepoint 8B`	3.9	4.0	4.8
`tracepoint 128B`	11.5	13.5	13.0
`tracepoint 512B`	37.7	43.1	40.9
`tracepoint 2kB`	160.2	192.9	174.7

Table 3: Latency measurements (nanoseconds) for Hindsight client API and autotriggers for a microbenchmark application configured with **1**, **4**, and **8 Threads** (§6.4). Default `tracepoint` writes a 32 kB trace event; we also measure 8–2048 B `tracepoint` payloads.

Fig. 6 plots latency-throughput curves under several different tracing configurations. The lowest latency and highest throughput is achieved with No Tracing, peaking at an average 71.0 k requests/s. Similar throughput is achieved by Jaeger when configured with 1% Head-sampling, at 70.2 k r/s. Hindsight peaks at 70.4 k r/s – a decrease of only 0.9% compared to no tracing. Hindsight generates on average 330 MB/s of trace data at peak request throughput, with an event horizon of 5.2 s, and consumes a combined 0.3 CPU cores across agents, coordinator, and collector. By comparison, Jaeger configured with Tail-sampling peaks at only 41.4 k r/s, an overhead of 41.7%; moreover, the workload over-saturates the OpenTelemetry collector, resulting in 94% of trace data being dropped while consuming 4.5 CPU cores.

Client API and autotrigger microbenchmarks. We run a benchmark application that generates traces and measures the overhead of calls to Hindsight's client API and autotrigger library. The benchmark writes traces by calling `begin` to start the trace, writing a total of 16 kB per trace by repeatedly calling `tracepoint`, then calling `end` to finish the trace. Each `tracepoint` call writes a 32-byte event `struct` (3 metadata fields and a timestamp) using Hindsight's OpenTelemetry library. Following each trace, the benchmark invokes five different autotriggers. The benchmark runs a custom number of threads to generate traces; each thread independently runs a continuous loop generating traces, and every thread writes a different trace. We configure Hindsight to use 32 kB buffers and a 1 GB buffer pool, and run a Hindsight agent. We run 1 minute per experiment (≈10–50 million traces).

As shown in Table 3, autotrigger overheads vary. CategoryTrigger is relatively cheap (<47 ns) and TriggerSet adds relatively little overhead to the wrapped trigger (6–53 ns). By contrast, PercentileTrigger overheads grow proportional to the percentile: up to 307, 512, and 1,134 ns respectively for tracking 99th, 99.9th, and 99.99th percentile latency due to larger internal data structures for tracking order statistics.

Table 3 also shows API latency for 1, 4, and 8 threads. Overall, Hindsight achieves nanosecond-scale API latency, and by design the expensive API calls (`begin`, `end`, and autotriggers) are limited to once per trace. `begin` and `end` vary from 70–230 ns, proportional to the number of threads due to contending on shared-memory queues to acquire and re-

turn buffers. By contrast, `tracepoint` call latency is mostly independent of the number of threads, between 7.9–8.5 ns (reduced to ≈4 ns when omitting timestamps). We also measure `tracepoint` latency for larger payloads up to 2 kB; latency increases only up to 175 ns per tracepoint since `tracepoint` is primarily a memory copy into the thread-local buffer established by `begin`.

7 Discussion

We next discuss items peripheral to Hindsight's core design.

7.1 Triggers

Mitigating spammy triggers. Hindsight's design currently isolates triggers based on a trigger ID, whereby different symptom detectors would use different trigger IDs, ensuring that a symptom detector that fires infrequently is not affected by one that fires too often.

Lateral trace IDs. All of Hindsight's autotriggers are lightweight symptom detectors that run within the application itself. In principle, the logic to decide when to trigger, and which `traceIds` to trigger, is arbitrary. Hindsight's autotriggers are classes which the application can instantiate that track state over time. For a `PercentileTrigger` that tracks latency, for example, the application must instantiate the autotrigger within the request handler, and add the new latency sample at the end of each request's execution. A `TriggerSet` can wrap any trigger; internally it maintains a sliding window of the N most recently-seen `traceIds` that tested the wrapped trigger. When we applied the `TriggerSet` in the UC3 experiment (Fig. 5c), for example, we measured queueing latency and the TriggerSet internally held the N most recent `traceIds` that were dequeued. When the wrapped autotrigger finally calls trigger, it will include all N of the `traceIds` in the trigger call.

7.2 Consistent Hashing

Hindsight agents have several forms of queuing and scheduling internally, primarily to decide which traces to evict and which to report to avoid any unbounded queue. Simple queueing (*e.g.* used by OpenTelemetry) indiscriminately drops data when the queue is full. When multiple agents have full queues, each independently dropping arbitrary data seriously compromises trace coherence. Instead, when Hindsight agents are at capacity, they bias towards dropping data from the same victim traces by preferentially discarding items from lowest priority traces. Thus even though Hindsight agents are operating independently, they seek to retain the same high-priority trace IDs when under load.

7.3 The Event Horizon

Parameters. Several factors influence Hindsight's event horizon: (i) the buffer pool size of each agent; (ii) the rate of new trace data being generated; (iii) the time between a request completing and a trigger firing. Inevitably, if there is too much trace data, or if triggers are too slow, Hindsight may be unable to keep the trace before its data is overwritten. For some use cases this means Hindsight cannot use retroactive sampling. However, head-sampling or tail-sampling would still be viable options, equivalent to existing distributed tracing frameworks.

Extending the event horizon. The solution is either to increase the memory available to Hindsight or to scale down the percentage of traced requests using Hindsight's optional *trace percentage*. Trace percentage is a separate configuration knob (defaulting to 100%) that controls the percentage of requests that generate trace data in the first place. The starting premise for Hindsight is that 100% tracing is acceptable, so we used 100% as the default and described Hindsight as such throughout this paper. However, if an application has overhead constraints or limited memory for a buffer pool, the percentage of requests that are traced in the first place can be scaled back. Hindsight enforces scale-back coherently across agents through consistent `traceId` hashing: *e.g.* 50% trace percentage will halve the trace data throughput and double the event horizon.

Mismatched and dynamic event horizons. The global event horizon of an application is dictated by the shortest event horizon among the constituent processes, since the whole trace becomes incoherent the moment the first agent evicts any of its data. This fundamental property of Hindsight can be addressed by enlarging the buffer pool memory on higher throughput nodes. Moreover, the buffer pool needs not be of fixed size. We considered implementing a dynamically-sized buffer pool, *e.g.* that can be configured with a target event horizon, but ultimately chose a fixed-size buffer pool to better bound memory overheads – a desirable property for telemetry systems [71].

Shared buffer pools. In our current design, we deploy one Hindsight agent per traced application process. If multiple containers share a machine, as in our experiments, several agents may run on the same machine. There is no reason why applications could not share a single machine-wide buffer pool, enabling processes to stock their buffer pool capacities and average out the differences between their event horizons.

7.4 Comparison with Tail Sampling

Event horizons. Hindsight's event horizon has an analogue in tail sampling. Since trace collectors cannot immediately perform tail sampling the instant trace data arrives, and must wait for all of the slices of a trace to arrive from all of the machines the request visited. Today, this is done with a timeout (*e.g.* 30s by default in OpenTelemetry [52]), after which the trace objects are constructed and tail samplers can be evaluated. If the application generates a high volume of trace data, then the trace collector can potentially run out of memory while awaiting data for to do tail sampling.

Tail sampling expressivity. Today's tail samplers focus on filters and outliers applied to span attributes and metrics. Yet a

tail sampling decision for one trace cannot influence the sampling decision of other traces. By contrast, Hindsight's lateral traces enable a trigger to specify other, related traces, in addition to the one exhibiting a symptom, allowing it to support use cases like temporal provenance (UC3). Tail samplers do not support such use cases, and they would be challenging to introduce due to the way trace data for different traces route to different collectors based on their `traceId`.

7.5 Robustness

Application failures. If the application process crashes (*e.g.* SEGV/NPE-type crashes), then Hindsight preserves problematic traces since Hindsight's agent continues to run and the trace data is preserved in memory in the shared buffer pool. The agent will also be able to continue responding to breadcrumbs. This is a secondary benefit of externalizing trace data on the critical path of requests, and is currently supported by Hindsight. By contrast, existing distributed tracing frameworks buffer trace data in application memory and would lose unreported data upon an application crash.

Agent failures. If Hindsight's agent crashes (irrespective of whether the application process crashes), then the buffer pool will still exist in-memory on the machine and could be later retrieved to inspect the state just prior to the crash. Hindsight does not currently implement such a recovery process. In addition, if an agent crashes, it will by default prevent Hindsight's coordinator from following breadcrumbs through this crashed agent. This can be overcome by extending Hindsight to propagate breadcrumbs for the last N visited nodes instead of just one; this would both avoid $(N-1)$-hop failures and also speed up Hindsight's collection process.

Kernel and hardware failures. In the case of kernel crashes or hardware failure, application-level traces are only useful if it was the application's behavior that triggered the crash. In this case, Hindsight's data would be lost.

8 Related Work

Distributed tracing. Numerous prior works identify end-to-end requests as a useful granularity for slicing telemetry data and troubleshooting distributed systems. Example use cases include detecting anomalous request structures [37, 64, 72], diagnosing changes in the steady-state [16, 57, 61], modeling workloads [46, 70], and identifying resource and queue contention [25, 44, 72]. Distributed tracing systems have been presented in industry [34, 64], as open-source tools [31, 52, 56, 78], and in academia [23, 45]. Edge-case troubleshooting stands in tension with overheads in distributed tracing, and head-sampling and tail-sampling offer alternative points in this space (§2.2).

Logging frameworks. Distributed tracing is the cousin of log ingestion frameworks that collect and store application-level log data [13, 66]. Log ingestion frameworks are agnostic to concepts like requests, do not record or group log data by requests, and cannot control head-sampling decisions coherently for requests – instead applications generate simple sequential streams of log data all at the same level of logging detail. Consequently, logs are typically far less detailed than distributed tracing and log ingestion frameworks handle a lower volume of data. For example, Chukwa reports on average 10kB/s per node [13]; Splunk limits to 330 kB/s per node [66]; Amazon CloudWatch limits to 5MB/s per log stream [12]. Early distributed tracing works rejected the idea of building distributed tracing atop logging, citing coherence challenges from brittle data, enormous post-processing costs, and fundamental scalability bottlenecks [17, 34, 64]. In practice, trace detail is typically far greater than even non-production debug-level logging [64], and it is easy to see why: head-sampling gives operators leeway to instrument their applications at fine detail, because they can amortize the high cost of a single trace by scaling down the number of collected traces. By comparison, log ingestion frameworks have no such opportunity.

Network provenance. Hindsight is similar in spirit to network packet provenance systems that chronicle the history of network state, enabling use cases such as tracking the origin or path traversed by a packet across the network. Earlier systems, like ExSPAN [77] and SNP [75], adopt this abstraction; more recent works like SyNDB [35] and SPP [14] apply network provenance for packet-level root-cause analysis on Internet scale. Packet provenance systems primarily trace only packet metadata, which is well-structured and can be summarized in-band; these systems tackle additional trust challenges outside of Hindsight's purview. By contrast, handling metadata to reconstruct the path of a trace is but one concern for Hindsight; Hindsight is focused on handling arbitrary payloads (*i.e.* trace data), and the resulting performance, coherence, and fairness challenges. Hindsight also draws inspiration from works focused on temporal provenance [76] and packet reputation [15] in distributed systems, although Hindsight's tracing abstractions operate entirely at the application level.

9 Conclusion

Hindsight circumvents the false dilemma between overhead and usefulness for diagnosing symptomatic edge cases by providing developers detailed traces from the recent past when they encounter symptoms of failures. We believe the retroactive sampling abstraction, and our Hindsight implementation of it, can shift the conversation around tracing away from mechanism (how to collect traces) to a question of policy (what traces should be collected), and allow distributed tracing systems to support edge-cases analysis: a key use case for which they were originally conceived.

Acknowledgements

We are grateful to our shepherd, Harsha Madhyastha, and the anonymous reviewers for their insightful feedback that helped improve our work.

References

[1] Businesses Losing $700 Billion a Year to IT Downtime, Says IHS. Retrieved April 2022 from https://www.businesswire.com/news/home/20160125005188/en/Businesses-Losing-700-Billion-a-Year-to-IT-Downtime-Says-IHS.

[2] Recent AWS outage and how you could have avoided downtime. Retrieved April 2022 from https://medium.com/@datapath_io/recent-aws-outage-and-how-you-could-have-avoided-downtime-7d9d9443d776.

[3] HDFS-3751: DN should log warnings for lengthy disk IOs. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-3751, 2014.

[4] HBASE-8228: Investigate time taken to snapshot memstore. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-8228, 2015.

[5] HBASE-8744: Enable HBase to log the entire latency profile for HDFS packets resulting in slow writes. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-8744, 2016.

[6] HDFS-11461: DataNode Disk Outlier Detection. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-11461, 2017.

[7] Jaeger Issue 425: Discuss post-trace (tail-based) sampling. Retrieved April 2022 from https://github.com/jaegertracing/jaeger/issues/425, 2017.

[8] HDFS-6110: adding more slow action log in critical write path. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-6110, 2018.

[9] Jaeger Issue 1861: Delayed Sampling. Retrieved April 2022 from https://github.com/jaegertracing/jaeger/issues/1861, 2019.

[10] Annanay Agarwal. How Grafana Labs enables horizontally scalable tail sampling in the OpenTelemetry Collector. Retrieved April 2022 from https://grafana.com/blog/2020/06/18/how-grafana-labs-enables-horizontally-scalable-tail-sampling-in-the-opentelemetry-collector/, 2020.

[11] Narayanan Arunachalam. Zipkin Secondary Sampling. Retrieved April 2022 from https://github.com/openzipkin-contrib/zipkin-secondary-sampling, 2019.

[12] AWS. AWS CloudWatch Logs quotas. Retrieved April 2022 from https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/cloudwatch_limits_cwl.html, 2022.

[13] Jerome Boulon, Andy Konwinski, Runping Qi, Ariel Rabkin, Eric Yang, and Mac Yang. Chukwa, a large-scale monitoring system. In *Proceedings of CCA*, volume 8, pages 1–5, 2008.

[14] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 374–388, 2017.

[15] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 115–128, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Mike Y Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *1st USENIX Symposium on Networked Systems Design & Implementation (NSDI'04)*, pages 23–23, 2004.

[17] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[18] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 1–14, 2019.

[21] elastic. Transaction Sampling. Retrieved April 2022 from https://www.elastic.co/guide/en/apm/guide/current/sampling.html#sampling.

[22] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.

[23] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07)*, 2007.

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 3–18, 2019.

[25] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 19–33, 2019.

[26] honeycomb.io. Getting At The Good Stuff: How To Sample Traces in Honeycomb. Technical report, honeycomb.io, 2019.

[27] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[28] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual*, volume 3 (3A, 3B, 3C & 3D): System Programming Guide. Intel, 2016.

[29] Irving Popovetsky. Getting At The Good Stuff: How To Sample Traces in Honeycomb. Retrieved April 2022 from `https://www.honeycomb.io/blog/getting-at-the-good-stuff-how-to-sample-traces-in-honeycomb/`, 2020.

[30] Ivan Topolnjak. Kamon: How to Keep Traces for Slow and Failed Requests. Retrieved April 2022 from `https://kamon.io/blog/how-to-keep-traces-for-slow-and-failed-requests/`, 2021.

[31] Jaeger: Open Source, End-to-End Distributed Tracing. Retrieved April 2022 from `https://www.jaegertracing.io/`.

[32] Jeremy Castile. What You Need to Know About Distributed Tracing and Sampling. Retrieved April 2022 from `https://thenewstack.io/what-you-need-to-know-about-distributed-tracing-and-sampling/`, 2020.

[33] Chris Jones, John Wilkes, Niall Murphy, and Cody Smith. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. `https://landing.google.com/sre/sre-book/chapters/service-level-objectives/`.

[34] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 34–50, 2017.

[35] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 253–268. USENIX Association, April 2021.

[36] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *9th ACM Symposium on Cloud Computing (SOCC '18)*, 2018.

[37] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*, pages 312–324, 2019.

[38] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[39] Lightstep. Learn about Microsatellites: How many Microsatellites do I need? Retrieved April 2022 from `https://docs.lightstep.com/docs/learn-about-micro-satellites`.

[40] Lightstep. OpenTelemetry-Collector Issue #4758: Tail-Based Sampling Scalability Issues. Retrieved April 2022 from `https://github.com/open-telemetry/opentelemetry-collector-contrib/issues/4758`, 2020.

[41] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 321–334, 2018.

[42] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[43] Yu Luo, Kirk Rodrigues, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[44] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[45] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[46] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the parallel execution of black-box services. In *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'11)*, 2011.

[47] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[48] Pulkit A Misra, María F Borge, Íñigo Goiri, Alvin R Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[49] New Relic. Technical distributed tracing details: Tail-based sampling algorithms. Retrieved April 2022 from `https://docs.newrelic.com/docs/distributed-tracing/concepts/how-new-relic-distributed-tracing-works/#tail-sampling-strategy`.

[50] New Relic. Technical distributed tracing details: Trace limits. Retrieved April 2022 from `https://docs.newrelic.com/docs/distributed-tracing/concepts/how-new-relic-distributed-tracing-works/#limits`.

[51] New Relic. Tail-based sampling (Infinite Tracing). Retrieved April 2022 from `https://docs.newrelic.com/docs/understand-dependencies/distributed-tracing/get-started/how-new-relic-distributed-tracing-works#tail-based`, 2020.

[52] OpenTelemetry: An Observability Framework for Cloud-Native Software. Retrieved April 2022 from `http://opentelemetry.io/`.

[53] OpenTelemetry. Tail Sampling Processor. Retrieved April 2022 from `https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor/tailsamplingprocessor`.

[54] OpenTelemetry Specification Issue 307: Allow samplers to be called during different moments in the Span lifetime. Retrieved April 2022 from `https://github.com/open-telemetry/opentelemetry-specification/issues/307`, 2019.

[55] OpenTelemetry Enhancement Proposal 115: Allow Additional Sampling Hooks. Retrieved April 2022 from `https://github.com/open-telemetry/oteps/pull/115`, 2020.

[56] OpenTracing: Vendor-Neutral APIs and Instrumentation for Distributed Tracing. Retrieved April 2022 from `http://opentracing.io/`.

[57] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing latency in multi-tier black-box services. In *4th International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'11)*, 2011.

[58] Maulik Pandey. Building Netflix's Distributed Tracing Infrastructure. Retrieved April 2022 from `https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304`, 2019.

[59] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.

[60] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 401–414, 2016.

[61] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing performance changes by comparing request flows. In *8th USENIX Symposium on Networked Systems Design & Implementation (NSDI'11)*, volume 5, pages 1–1, 2011.

[62] Yuri Shkuro. *Mastering Distributed Tracing*. Packt Publishing, Feb 2019.

[63] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[64] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[65] Splunk. Observability Cloud Usage, Subscription Limits Enforcement, and Entitlements. Retrieved April 2022 from `https://www.splunk.com/en_us/legal/usage-subscription-limits-enforcement-and-entitlements.html`, 2022.

[66] Splunk. Splunk Enterprise Capacity Planning Manual - Summary of performance recommendations. Retrieved April 2022 from `https://docs.splunk.com/Documentation/Splunk/8.2.6/Capacity/Summaryofperformancerecommendations`, 2022.

[67] Splunk. Use cases: Troubleshoot errors and monitor application performance using Splunk APM. Retrieved April 2022 from `https://docs.splunk.com/Observability/apm/apm-use-cases/apm-use-cases-intro.html#nav-Use-cases:-Troubleshoot-errors-and-monitor-application-performance`, 2022.

[68] Cindy Sridharan. *Distributed Systems Observability*. O'Reilly Media, 2018.

[69] Kun Suo, Jia Rao, Luwei Cheng, and Francis CM Lau. Time capsule: Tracing packet latency across different layers in virtualized systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 1–9, 2016.

[70] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06)*, 2006.

[71] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, 2015.

[72] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 395–420, 2019.

[73] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 159–172, 2011.

[74] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 456–468. IEEE, 2016.

[75] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 295–310, 2011.

[76] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed time-aware provenance. *Proceedings of the VLDB Endowment*, 6(2):49–60, 2012.

[77] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 615–626, 2010.

[78] Zipkin: A Distributed Tracing System. Retrieved April 2022 from `http://zipkin.io/`.

A Supplemental Experiment Results

A.1 End-to-end Application Overheads

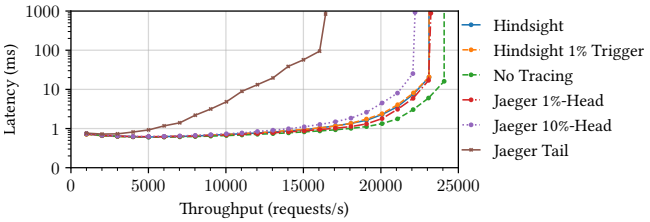


Fig. 7: **End-to-end latency and throughput** for a 2-service MicroBricks topology configured with various tracers, demonstrating minimal application impact for Hindsight despite tracing 100% of requests.

Fig. 7 shows a variant of the experiment described in §6.4. In the original experiment the services are configured to perform no additional compute. We additionally repeat the experiment with services configured to perform approximately 100 microseconds of matrix-multiply compute per service. We observe similar to trends to those discussed in §6.4; in particular Hindsight has a comparable latency and throughput profile to Jaeger configured with 1% head-sampling.

A.2 Head-Sampling and Tail-Sampling Overheads.

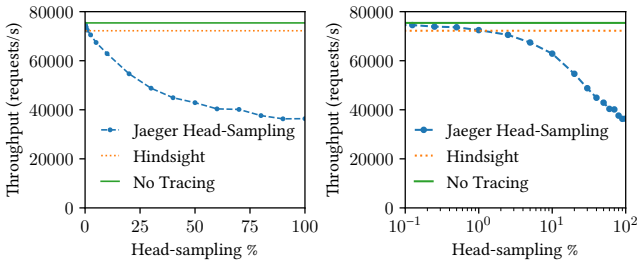


Fig. 8: **Head-sampling impact on end-to-end throughput** of a 2-service MicroBricks topology under a closed-loop workload. We vary the head-sampling percentage configured for OpenTelemetry Jaeger. The right figure plots the same data with a log-scale x-axis to highlight overheads at low head-sampling percentages. Tail-sampling is equivalent to 100% Head-sampling

We further measure the application-level impact of different head-sampling regimes. We run the application described in Fig. 8 and submit a closed-loop workload to saturate the system. We measure the application-level throughput achieved with by OpenTelemetry and Jaeger configured with different head-sampling percentages. We compare the throughput to that of Hindsight and with No Tracing. The results illustrate that the OpenTelemetry Jaeger overheads at typical low sampling percentages (<1%) is negligible, but the client library performance deteriorates at higher tracing percentages. 100% head-sampling is equivalent to tail-sampling.

A.3 Client throughput

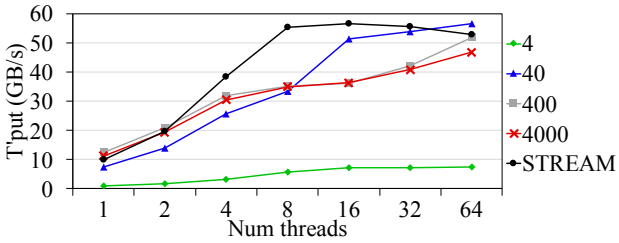


Fig. 9: **Client Throughput** achieved by a microbenchmark that varies the number of threads and the size of payload to `tracepoint` calls. Even modest payloads (40 bytes) can saturate memory bandwidth.

The purpose of this experiment is to evaluate the `tracepoint` write throughput that client applications can achieve, based on the payload size to `tracepoint` calls. The experiment demonstrates the peak attainable data ingestion throughput for different numbers of threads and different payload sizes. Larger payloads can attain higher throughput, but even with small payloads (40 bytes), we can saturate memory bandwidth.

This experiment configures Hindsight to use 32 kB buffers and a 1 GB buffer pool. We run a client application comprising between 1 and 64 threads. Each thread continually writes traces in a loop to Hindsight. Writing a trace entails calling `begin`, 100 `tracepoints`, then `end`. We repeat the experiment for different numbers of threads and varying the size of `tracepoint` calls from 4 bytes to 4000 bytes, resulting in traces between 400 and 400,000 bytes in size.

Fig. 9 plots the throughput achieved in GB/s. Small payloads of 4 bytes fail to fully saturate memory bandwidth, achieving only 887 MB/s with one thread and peaking at 7.55 GB/s with 64 threads. By contrast, even a modest increase in payload size to 40 bytes is enough to nearly saturate memory bandwidth; with 400 byte payloads, we achieve throughput of 12.5 GB/s on a single core. We include in Fig. 9 measurements of peak memory bandwidth from the STREAM benchmark [47].

Hindsight achieves high throughput, despite each trace acquiring and writing a new buffer at an arbitrary non-sequential offset in the buffer pool. This occurs because Hindsight's client library coordinates buffers only during `begin` and `end` (at the start and end of each trace respectively) and stores the buffer pointer thread-local in the interim. Calls to `tracepoint` are then little more than a memory copy to the thread-local buffer acquired by `begin`.

A.4 Control-Data Trade-offs

Hindsight's design emphasizes a control-data split, to enable applications to write trace data at large volume while reducing the amount of indexing work agents must perform. The main factor influencing this trade-off is Hindsight's buffer size. With large buffers, agents index fewer buffers and thus

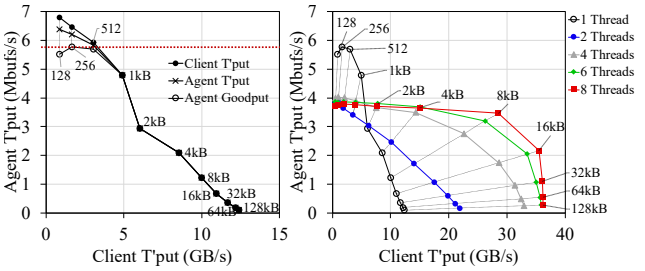


Fig. 10: **Buffer size trade-off.** Each data point is annotated with the Hindsight buffer size. Small buffers require more indexing work from agents, while large buffers are less memory efficient by exacerbating internal fragmentation.

perform less work; however it may exacerbate internal fragmentation when traces only partially fill buffers. Conversely, small buffers are more space-efficient, but require more indexing work from agents. We evaluate this trade-off by measuring client-side and agent-side throughputs, while varying Hindsight's internal buffer size from very small (128 B) to very large buffers (128 kB).

We run the benchmark application with one thread, 100 kB traces, and a payload of 1 kB per `tracepoint` call (Hindsight fragments payloads across multiple buffers when necessary). Fig. 10 (left) plots the client-side throughput of generating data (*x*-axis) and the agent-side throughput of indexing buffers (*y*-axis). We annotate data points with the corresponding buffer size used. Large buffer sizes (128 kB) can support peak client data throughput (12.1 GB/s) while requiring little of the agent. Conversely, tiny buffer sizes (128 B) stress the agent buffer throughput since we more frequently cycle buffers through the queues. Fig. 10 (left) plots three lines and indicates two important phenomena. The client throughput line plots the rate at which the client writes buffers, whereas the agent throughput line plots the rate at which the agent cycles buffers; the delta in-between are 'null buffers', written by the client because the available queue is empty, *i.e.* the agent cannot keep up. Writing to null buffers means lost trace data; the third line, agent goodput, only counts buffers of coherent traces, *i.e.* excluding all buffers for traces that lost data. We observe that the goodput with 128 B buffers is lower than with 256 B buffers due to greater loss. In general, with $\geq$1 kB buffers, the agent is able to consistently keep up without losing data.

Fig. 10 repeats this experiment with varying numbers of threads, and plots client-side data throughput and agent-side buffer goodput. Buffer sizes of 16 kB and higher are sufficient for reaching peak write throughput while remaining comfortably within agent throughput limits; by default, we select 32 kB for Hindsight.

DiSh: Dynamic Shell-Script Distribution

Tammam Mustafa
MIT

Konstantinos Kallas
University of Pennsylvania

Pratyush Das
Purdue University

Nikos Vasilakis
Brown University

Abstract

Shell scripting remains prevalent for automation and data-processing tasks, partly due to its dynamic features—*e.g.*, expansion, substitution—and language agnosticism—*i.e.*, the ability to combine third-party commands implemented in any programming language. Unfortunately, these characteristics hinder automated shell-script distribution, often necessary for dealing with large datasets that do not fit on a single computer. This paper introduces DiSh, a system that distributes the execution of dynamic shell scripts operating on distributed filesystems. DiSh is designed as a shim that applies program analyses and transformations to leverage distributed computing, while delegating all execution to the underlying shell available on each computing node. As a result, DiSh does not require modifications to shell scripts and maintains compatibility with existing shells and legacy functionality. We evaluate DiSh against several options available to users today: (i) Bash, a single-node shell-interpreter baseline, (ii) PaSh, a state-of-the-art automated-parallelization system, and (iii) Hadoop Streaming, a MapReduce system that supports language-agnostic third-party components. Combined, our results demonstrate that DiSh offers significant performance gains, requires no developer effort, and handles arbitrary dynamic behaviors pervasive in real-world shell scripts.

1 Introduction

Unix and Linux shell scripting remains prevalent—8th most popular language on GitHub in 2022 [20]—for data processing, system orchestration, and other automation tasks. Part of this prevalence can be attributed to a unique combination of features: (1) powerful and language-agnostic primitives for composing components available in any programming language; (2) dynamic features such as command substitution, variable expansion, and state reflection on the file system; and (3) a wide range of useful components called commands, available in the broader environment and tailored to specific tasks. These features enable the composition of succinct and powerful programs on a single computer (§2).

Tab. 1: Available options for scaling out shell programs. Compatibility: support unmodified shell scripts. Granularity: support fine-grained distribution. Expressiveness: support arbitrary dynamic behaviors. Agnosticism: support components in any programming language. Equivalence: behavior equivalence with existing shells.

Approach	Compatibility	Granularity	Expressiveness	Agnosticism	Equivalence	Examples
Distributed Shells	□	■	■	■	□	[14, 18, 63]
POSH	■	■	□	■	□	[49]
Cluster Comp. Frameworks (CCF)	□	◪	□	□	□	[44, 57, 68, 71]
Language-agnostic CCFs	□	◪	□	■	■	[25, 30]
Job Scheduling Tools	■	□	□	■	■	[19, 29, 58, 69]
Other languages	□	■	■	□	□	[16, 60, 66]
DiSh	■	■	■	■	■	

Unfortunately, these features also hinder automated shell-script scale-out to multiple computers. Such scale-out is often necessary not only to accelerate computations, but also to compute over data that either do not fit on a single computer or are naturally distributed across multiple computers.

State of the art: Shell users dealing with large datasets that do not fit on a single computer are left with only a few options (Tab. 1). One option is to use a distributed shell [14, 18, 63]. Distributed shells require rewriting scripts manually and only support a small subset of Unix features—often with limited, if any, dynamic features and varying support for composition constructs. A recent distributed shell named POSH [49] can handle a subset of shell scripts without rewriting—although that subset is limited to dataflow-only computations and also does not include arbitrary dynamic shell behaviors. In addition, since POSH is a shell reimplementation, it is not behaviorally equivalent with existing shells and thus risks breaking ported scripts. A second option is to rewrite (parts of) the script in a cluster-computing framework [11, 44, 68, 71].

These only support pure computations (*e.g.*, batch, stream), require manual rewriting, and only rarely [25, 30] support language-agnostic components. Another option is job scheduling tools [19, 29, 58, 69], but these operate at a coarse granularity and do not leverage parallelism available in individual commands. Yet another option is to rewrite scripts in languages that support distribution [3, 40, 42, 66], foregoing the shell's succinctness and language agnosticism. To summarize, these options operate on a subset of the shell, require significant manual effort, risk breaking correctness, or—most often—suffer from a combination of these limitations (see §8 for more details).

Dynamic shell-script distribution: This paper presents DiSh, a system designed to scale out shell scripts operating on distributed filesystems while maintaining full POSIX compatibility. DiSh satisfies all requirements in Table 1: it operates on existing shell scripts; it distributes scripts at the granularity of individual commands; it handles arbitrary dynamic shell features such as substitution and expansion; it allows the use of commands and utilities of any language; and, most importantly, it is behaviorally equivalent to Bash.

DiSh first instruments the execution of a script to identify regions that may benefit from distribution. At runtime, it compiles these regions to an intermediate representation which it then optimizes to introduce appropriate parallelism, buffering, communication, and coordination. DiSh then executes each compiled region in a distributed fashion using the same shell interpreter, components, and data as the original script.

Implementation and results: DiSh is implemented as a shim layer (rather than a shell) that wraps and orchestrates the (completely unmodified) user shell, delegating all execution to the underlying shell available on each computing node. This design hides distribution from the user and avoids modifying the underlying shell interpreter: the user thinks that their original script is being executed (but faster); each underlying shell is given a part of the distributed script to execute. As a result, DiSh achieves a new milestone in automated shell-script distribution: it offers significant performance benefits, it avoids modifications to shell scripts, and it maintains full POSIX compatibility. Additionally, this modular design allows further research and improvements without modifications in the underlying shell.

We characterize DiSh's performance on a 4-node on-premise cluster and a 20-node cloud deployment using 76 scripts—including ones not trivially expressible in modern distributed computing frameworks, such as scripts with `for` loops, side-effects, and complex third-party components. DiSh surpasses the speedups achieved by production-grade systems on existing benchmarks and extends speedups to new ones: it achieves significant speedups over (1) Bash (avg: 13.6×; max: 136.3×), a single-node shell-interpreter baseline; (2) PaSh (avg: 8.9×; max: 108.8×), a shell-script parallelization system; and (3) Hadoop Streaming (avg: 7.2×; max: 32.3×), a cluster computing framework that supports language-agnostic components and shell scripts. Moreover, whereas Hadoop Streaming does not support 27/76 scripts and requires rewriting 7/76 scripts, DiSh runs all scripts without any modifications; in fact, DiSh is able to execute the entire POSIX shell test suite, only diverging in one error code out of thousands of assertions.

Paper outline and contributions: The paper begins with an example and overview (§2) of DiSh's use and techniques. Sections 3–6 present DiSh's key components:

- Dynamic orchestration (§3): DiSh parses, pre-processes, expands, and orchestrates its input script to enable dynamic distribution at runtime.
- Compilation (§4): During script execution, DiSh compiles certain regions to an intermediate representation and applies a series of optimizations.
- Distribution (§5): DiSh distributes each region to a set of workers in a way that promotes co-location of processing primitives and the data blocks these operate on.
- Runtime support (§6): DiSh bundles additional runtime primitives supporting correct and efficient communication in the context of distributed shell script execution.

The paper then presents DiSh's evaluation (§7) and related work (§8), before concluding (§9).

DiSh limitations: DiSh currently does not tolerate failures such as worker aborts or network partitions. In such occasions, users are expected to rerun their scripts similar to how they do in non-distributed executions: due to the shell's dynamic features and its support for third-party components, users often re-run failing scripts from the start. The current DiSh prototype does not implement support for security features such as encryption and containment.

Availability: All the work described in this paper has been implemented and incorporated into PaSh—an MIT-licensed project—and is available by the Linux Foundation at https://github.com/binpash/dish.

2 Background, Example, and Overview

DiSh allows everyday shell scripts to reap the benefits of distributed computing: execute on data that do not fit on a single machine, often also speeding up expensive computations.

Intended use: DiSh is designed to support a variety of use cases, depending on the details of the distributed environment on which the system is executing. The most common case is one where input data are downloaded and stored in a distributed file system such as HDFS[1] and then processed using

[1]The choice of HDFS is not binding. DiSh could work on top of any distributed file system (*e.g.*, NFS or Alluxio [35]) that exposes the locations of file blocks. To achieve performance benefits due to co-location, there also needs to be available compute on the nodes that host that file system.

various analyses. This is useful for datasets that do not fit on a single computer, that are naturally distributed across multiple computers, or that can be processed faster in a data-parallel fashion. DiSh will distribute the computation appropriately, often running data-parallel instances on multiple machines and multiple processors per machine. DiSh also supports hybrid operation where data resides on both distributed and local file systems; this is useful for computations that contain CPU-intensive stages over datasets that do not necessarily reside on distributed file systems.

Example script: Fig. 1 shows a shell script that calculates maximum and average temperatures across the US, on datasets hosted on the National Oceanic and Atmospheric Administration (NOAA). The script is split into three parts: (p. 1) an 11-stage pre-processing pipeline to download data from NOAA and store them on HDFS, with the data range controlled upon invocation via dynamic arguments `$1` and `$2`; (p. 2, 3) two 5-stage pipelines calculating and storing maximum and average temperatures to the local file system.

HDFS is a distributed file system for handling large data sets on commodity hardware. Scripts like the one in Fig. 1 that process files stored in distributed file systems spend most of their execution time moving files across the network. On a 4-node cluster (§7) and 3.6GB of input, running just `hdfs dfs -cat` takes 346s; computing pipeline 2 (maximum temperature) only adds 6s. This phenomenon is due to pipeline parallelism: the execution time of all concurrently executing commands is mostly shadowed by `hdfs dfs -cat`.

Opportunities for scale-out: There are ample opportunities for improving the performance of this script. Since all parts contain stages that operate on large datasets, we should be able to execute (at least some of) their stages in a data-parallel fashion. For example, we should parallelize commands that process their input independently, such as `cut` and `grep`, by having them operate in parallel over partial inputs.

Additionally, carefully colocating computation and data should also improve performance. For example, we should schedule the data-parallel execution of the aforementioned `cut` and `grep` instances on machines that store the respective data segments. Directly operating on distributed file segments, rather than gathering and processing data on a subset of the machines, eliminates most data-movement overheads.

Finally, the execution of program fragments that do not depend on each other could become concurrent: since parts 2 and 3 are independent on each other, we should be able to overlap their execution in a task-parallel fashion.

Key challenges: Unfortunately, exploiting these opportunities to scale out execution automatically is particularly challenging in the context of the shell. First, exposing opportunities at the level of individual commands such as `cut` and `grep` is challenging—and this is why prior systems often focused on coarser, script-level or job-level granularity [19, 69].

Second, pervasive dynamic features, file-system introspection, and other side-effects impede traditional distribution approaches based on static transformation—this is why prior shell-script distribution work [25, 49] focuses on side-effect-free dataflow subsets. These challenges are compounded by the presence of more elaborate control flow such as `for` loops, `break`, and `trap` statements present in ordinary shell scripts.

```
NOAA=${NOAA:-http://ndr.md/data/noaa/}
TEMPS=${TEMPS:-/noaa/temps.txt}
hdfs dfs -mkdir /noaa

## Pipeline 1: Download temperature data
##              and store to HDFS
seq $1 $2 | sed "s;^;$NOAA;" |
 sed 's;$;/;' | xargs -r -n 1 curl -s | grep gz |
 tr -s ' \n' | cut -d ' ' -f9 |
 sed 's;^\(.*\)\(20[0-9][0-9]\).gz;\2/\1\2\.gz;' |
 sed "s;^;$NOAA;" | xargs -n1 curl -s |
 gunzip | hdfs dfs -put - $TEMPS

## Pipeline 2: Compute maximum temperature
##              over all data
hdfs dfs -cat $TEMPS | cut -c 89-92 | grep -v 999 |
 sort -rn | head -n1 > max.txt

## Pipeline 3: Compute average temperature
##              over all data
hdfs dfs -cat $TEMPS | cut -c 89-92 | grep -v 999 |
 awk "{ t += \$1; i++ } END { print t/i }" > avg.txt
```

Fig. 1: Example script. Downloading a temperature dataset, storing on a distributed file system, and running analysis to extract statistics.

Third, behavioral equivalence with existing shells is practically unattainable, especially with shell *re*implementations—after all, even production-grade shells such as Bash and zsh diverge subtly in their POSIX behavior [23]. A new distributed shell [14, 49] has little hope of *not* breaking some scripts.

DiSh overview: To overcome these challenges DiSh (1) extracts details about the behavior of commands through command annotations, (2) deals with dynamic features and side-effects by analyzing scripts at runtime using dynamic orchestration, and (3) achieves behavioral equivalence with Bash by only performing script transformations and delegating execution to the underlying interpreter. DiSh is designed to dynamically orchestrate, compile, schedule, and support the execution of shell scripts (Fig. 2). DiSh's orchestration (§3) kicks in when a potentially distributable script region is identified, saves a snapshot of the user's shell environment (variables, configuration) and invokes the DiSh compiler with the candidate region (Fig. 2a). The compiler analyzes this region and if possible, translates it to a dataflow graph—which it then optimizes to introduce parallelism, buffering, *etc.* (§4), finally passing it off to the scheduler (Fig. 2b); or aborts compilation (Fig. 2d) because it cannot guarantee that the region is pure,

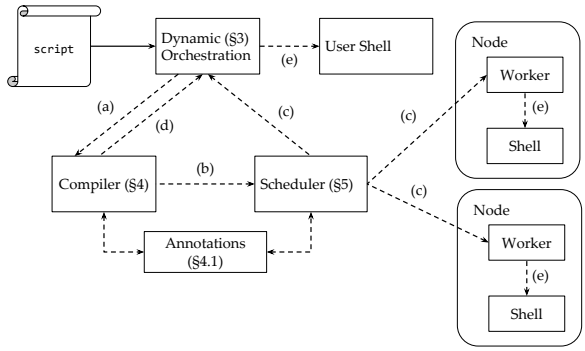


Fig. 2: DiSh architecture overview. Steps: (a) compile script region; (b) schedule compiled dataflow; (c) send dataflow subgraphs to workers; (d) compilation failed, fall back to original region; and (e) execute script region (compiled or original).

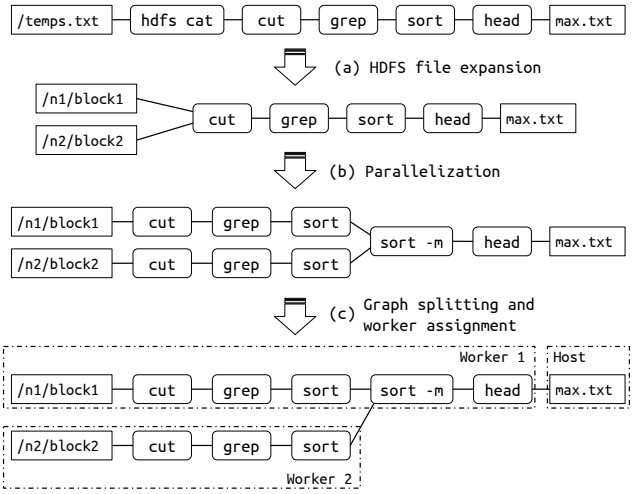


Fig. 3: DiSh dataflow graph stages. (a) HDFS files are expanded to sequences of blocks. (b) the graph is parallelized based on the command specifications. (c) the scheduler splits the graph and assigns subgraphs to workers.

i.e., side-effect-free. The scheduler (§5) divides the compiled dataflow graph into different subgraphs which it sends to available cluster workers (Fig. 2c). In response to these execution requests, workers apply a second pass of optimizations to better utilize available resources, translate the dataflow graph back to a shell script (Fig. 2e), load the snapshot of the shell environment stored by the orchestrator, and execute the script using the local, unmodified shell interpreter (§6).

Applying DiSh: DiSh preprocesses the script in Fig. 1 to identify script regions that could benefit from distribution—in this case, all three pipelines. It then replaces each of these regions with calls to the dynamic orchestrator and attempts to distribute them at runtime. During execution, the orchestrator queries the DiSh compiler to determine whether a region is pure and thus distributable: if the compiler succeeds, it translates the region to a dataflow graph. Since regions contain arbitrary black-box commands, DiSh cannot analyze them directly. Instead, it employs a command specification framework that contains partial specifications of command invocations such as their inputs and outputs. For example, DiSh's compiler uses these specifications to determine that `hdfs dfs -cat /noaa/temps.txt` reads from the HDFS file `/noaa/temps.txt` and writes to `stdout`. Once a region is in dataflow form, DiSh applies transformations to distribute it.

Fig. 3 shows the distribution stages for pipeline 2 (maximum temperature). DiSh first detects operations on HDFS files (*i.e.*, HDFS `cat`) and expands each distributed file to its segments (datablocks), often stored on different physical machines. Informed by command annotations, DiSh applies parallelization transformations: commands like `cut` and `grep` are parallelizable directly and can be executed on the machine with the raw input datablock. The scheduler then splits the compiled graph into subgraphs and maps them to workers in a data-aware fashion. Finally, each worker translates the graph back to a shell script, adds additional runtime primitives (commands), and executes it locally.

The result? DiSh drops the execution of pipeline 2 from 352s to 6s while maintaining full behavioral equivalence and requiring no modifications to the user shell.

3 Dynamic Shell Orchestrator

To facilitate adoption, an important desideratum in the design of DiSh is to achieve behavioral equivalence with the underlying shell interpreter. To achieve this, DiSh is not designed to operate as another shell, but rather wraps the user's existing shell interpreter and the shell interpreters on the worker machines. As a result, DiSh hides parallelization and distribution from both the user and the underlying shells: the user thinks that their original script is being executed—just faster—and each underlying shell simply executes a standard non-distributed shell script. This allows DiSh to achieve exceedingly high compatibility with the underlying shell implementation (§7.3), while also minimizing maintenance costs since updates and modifications on the underlying shell are reflected in DiSh without any change.

Fig. 4 shows an overview of the structure of DiSh's dynamic orchestration. To achieve dynamic shell script orchestration without any shell-interpreter modification, DiSh opts for a light-weight script instrumentation pre-processing step: it instruments *potentially* distributable regions with invocations to the orchestration engine. It chooses regions with the goal of maximizing distribution benefits: intuitively, it focuses on commands and pipelines rather than control-flow statements and variable assignments. However, the choice of these region boundaries is not binding—the preprocessor just needs to be precise enough to determine potential regions, but DiSh will eventually decide whether or not (and if yes, how) to distribute

a candidate region at runtime. The preprocessor first parses the original script, it then replaces the relevant program regions with orchestration prefixes, and then un-parses (emits) it back as an instrumented script that is given for execution to the user's shell interpreter.

The instrumented script then makes calls to the orchestration engine. The orchestration engine is itself a shell script coordinating with the compiler and worker manager and attempting to distribute the upcoming region (see §4 and 5 for details). If it succeeds, it runs the distributed version of the region. If it aborts, it just falls back to the original region, executing it normally. Reasons for aborting include the region being side-effectful, *e.g.*, modifying some environment variable, or lacking relevant command annotations.

Preprocessor: The preprocessor searches for maximal potentially distributable regions by processing the AST bottom-up, combining distributable subtrees when they are composed using constructs that do not introduce scheduling constraints (*e.g.*, `&`, `|`). When a region cannot outgrow a certain subtree, DiSH replaces it with a call to the orchestration engine. If the region is successfully compiled (at runtime), DiSH translates it to a dataflow representation—a convenient and well-studied model amenable to transformation-based optimizations [26]. At a later point, DiSH running on each node translates the instrumented AST resulting from the compilation back to shell code and passes it to the underlying shell for execution.

Parsing library: DiSH invokes parsing and unparsing routines frequently, and therefore needs them to be very efficient. To that end, it uses an internal Python implementation [32] of POSIX-shell-script parsing and unparsing based on `libdash` [22, 23]. The DiSH parser contains several optimizations such as caching, inlining, and careful array appending to achieve improved performance.

Orchestration engine: DiSH's orchestration engine is designed to maintain the original script behavior and minimize runtime overhead—as it is invoked multiple times per script. The engine is a reflective shell script: it coordinates transparently with the compiler to determine whether or not to parallelize a script by inspecting the state of the shell and that of the broader system. DiSH constantly switches between two execution modes when executing scripts: (1) conventional shell mode, where scripts execute in the original shell context, and (2) DiSH mode, where the runtime reflects on shell state and invokes the compiler to determine whether to execute the original or an optimized version of the target region. To switch from shell mode to DiSH mode, the engine saves the state of the user's shell; to switch back, it restores the state of the user's shell. The state of a shell is quite complex: apart from saving and restoring variables, DiSH must account for various shell flags along with other internal shell state (*e.g.*, the previous exit status, working directory). During an invocation, the engine first switches to DiSH mode, communicates with the compiler and scheduler to determine whether a region can be safely distributed, and it then switches back to shell mode to execute the original or distributed version of the script.

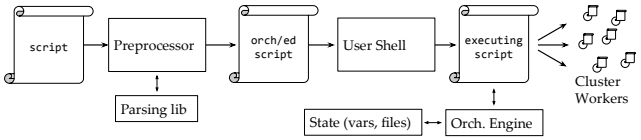


Fig. 4: Dynamic orchestration overview. DiSH instruments scripts with calls to the orchestration engine, which passes program fragments to the worker manager at run-time.

Environment sharing: The distributed version of the script region might execute on a different shell (or even machine). Therefore, a challenge that DiSH needs to address is to make sure that all regions execute in the correct environment—including access to the latest variable values and function definitions. To achieve that the engine takes a snapshot of the environment right before execution. It then transfers the snapshot to the distributed workers, which they load before executing the incoming script fragment. This is safe to do since successful distribution of a region implies that it is pure (and therefore does not affect the environment), and thus the snapshot will be valid until the region finishes execution.

String expansion: To correctly determine if a script region is safe to distribute, the compiler needs to expand all strings in that region. Since DiSH performs compilation and distribution of each script region at runtime, right before execution, it has access to all the latest variables and system state to fully expand all strings in the region. DiSH only implements a common and safe subset of all available expansions, and avoids implementing side-effectful expansions that have the risk of affecting the environment (*e.g.*, `${x=foo}`: set `x` to `foo` if `x` is unset). Note that DiSH keeps expansion local: it does not expand regions succeeding the target region, as these might depend on the execution of the target region.

4 Compiler

This section describes the compiler of DiSH, which builds on the PaSH parallelizing compiler [64]. The compiler is given the AST of an input script fragment and information about the commands in that fragment (§4.1). It then attempts to transform it to a dataflow graph (§4.2), an intermediate representation amenable to parallelization transformations. If the compiler succeeds in transforming a script region to a parallel dataflow graph, that graph is then passed to the scheduler which then decides how to map subgraph components to the available worker nodes. As the compiler operates at runtime in a just-in-time fashion, it exploits ample opportunities for parallelization even across subgraphs (§4.3).

4.1 Command Annotations

DiSh needs to support analyses and transformations over third-party commands, without access to their source code. To achieve this, DiSh uses annotations á la PaSh [64] and POSH [49], capturing information about a command invocation's parallelizability class, inputs, and outputs. Command annotations act as an intermediate layer that provides restricted but sufficient information about the behavior of a command to analysis and transformation systems like DiSh. They also enable reuse, as they are not tied to a particular analysis and can thus be reused by different tools. For this work, DiSh reuses the set of annotations developed by the authors of PaSh [64] extended annotations for commands that appear in the evaluation of DiSh (§7).

A command annotation in DiSh encodes information at the level of individual command invocations, *i.e.*, precise instantiations of a command's flags, options, and arguments. Among other information, annotations determine how a command invocation affects its environment, and specifically whether it is pure, *i.e.*, whether it only affects its environment by writing and reading to and from a well-defined set of files—information which DiSh uses when translating commands to and from dataflow nodes (§4.2). For example, the annotation for `grep` can be used to extract that the script fragment `grep -f dict.txt src.txt > out.txt` contains two input files `dict.txt` and `src.txt` and one output file `out.txt`. This knowledge of input and output files is used by DiSh to enable location-aware distribution, by scheduling the computation on nodes that contain relevant data blocks. Additionally, annotations describe parallelization opportunities—*e.g.*, that `grep "pattern" src.txt` processes each line of `src.txt` independently and thus can be parallelized at a line boundary.

4.2 Dataflow Model

The core of DiSh's compiler is an order-aware dataflow model that captures pure shell script regions that read from a well-defined set of input files and write to a well-defined set of output files—*i.e.*, they do not modify their environment in any other way. This model is expressive enough to capture a shell subset used pervasively in data processing scripts [26].

In this model, nodes represent commands and edges represent files, pipes, named FIFOs, and file descriptors. The model is order-aware in the sense that it keeps information about the order in which nodes read from their inputs, which is important for the script's semantics. For example, `grep "pattern" in1.txt - in2.txt` first reads from `in1.txt`, then from its standard input, and then from `in2.txt`. This order awareness allows DiSh to perform transformations that optimize execution of a script—*e.g.*, by exposing parallelism—but preserve its original behavior.

Translation workflow: Given an AST representation of an input script region, the compiler uses annotations to deduce whether commands are pure *i.e.*, they only affect their environment through a well-defined set of output files, and attempts to transform them to dataflow nodes. If all commands in the region are pure the compiler transforms the region to a dataflow graph. It then applies transformations (described below), optimizing the graph to expose parallelism and improve the script's performance. Finally, it serializes the graph back to a (now optimized) shell script, by translating every node back to a command and connecting them all together with appropriate channels (*e.g.*, FIFOs, RFIFOs, redirections).

Transformations: DiSh's transformations enable data-parallel execution by replicating nodes in the graph and adding appropriate split and merge nodes around them. They apply a pass over the graph to remove pairs of inverse nodes—*i.e.*, pairs of nodes whose semantic effects cancel out but whose performance effects are additive—for example, a concatenation-style merge followed by a linear split. For commutative commands, *i.e.*, commands that produce the same output irregardless of their input-line order, DiSh applies transformations that pack and unpack metadata across the graph—achieving better performance by avoiding unnecessary blocking and buffering. Finally, to improve the flow of data across the graph, DiSh applies additional transformations that inject hybrid memory-disk buffer nodes in points in the graph that are likely to become bottlenecks.

Remote file resources and HDFS files: To support scripts that perform data analysis on a combination of HDFS and local files, DiSh extends the dataflow model with remote-file resources (RFRs) that encode file blocks in different nodes. RFRs usually represent blocks of files that are partitioned and replicated in HDFS, and contain information about the location of the data in the distributed environment. This information could contain multiple locations to support replication, and is used by the scheduler to assign script fragments to different workers. When the DiSh compiler comes across an HDFS file path, it queries HDFS to determine the locations of its file blocks and then expands that file to a sequence of RFRs, each of which represents a block.

4.3 Dynamic Dependency Untangling (DDU)

Scripts often contain regions that are independent, *i.e.*, they have different (file) working sets. Independent regions could potentially run in parallel, better utilizing computational resources and improving the execution times of the scripts in which they belong. However, inferring independence statically and ahead of time is challenging as shell scripts make extensive use of dynamic features. Figure 5 shows an example script that contains independent fragments but also features dynamic behavior. This script iterates over all files in an HDFS directory, compresses them using `gzip`, and finally stores them as independent files.

Determining independence statically in this script would

```
for item in $(hdfs dfs -ls -C ${IN});
do
    output_name=$(basename $item).zip
    hdfs dfs -cat $item |
        gzip -c > $OUT/$output_name
done
```

Fig. 5: Example of independent regions. This shell script compresses all files in a directory—but each iteration results in an independent body region that can be executed in parallel.

require inferring values of environment variables (like `IN` and `OUT`) and the state of the file system, *e.g.*, `hdfs dfs -ls`. DiSh's dynamic orchestration (§3) circumvents this challenge by making distribution decisions during the execution of the script when environment variables and the file system state are known. DiSh further exploits this by discovering independent dataflow regions at runtime and executing them in parallel—even if they were not parallel in the original script.

When DiSh successfully compiles a dataflow region (at runtime), it knows that the region is pure and therefore can determine the region's inputs and outputs—and it does so for free, without additional analysis or inference stages. DiSh then uses this information to check for read-write or write-write dependency conflicts with regions that are running concurrently. If none is found, DiSh passes the region to the scheduler, which orchestrates distributed execution, and then immediately continues the execution of the script until it reaches the next dataflow region. Whenever the compilation of a dataflow region fails, DiSh cannot safely detect the input and output information of this region—and thus it needs to wait until every previous region is done executing to ensure that no dependency will be violated.

Since DDU is done at runtime it is both sound, *i.e.*, it does not execute dependent fragments concurrently, and precise, *i.e.*, it offers significant benefits due to improved parallelism and resource utilization—especially for scripts that do not contain highly data-parallelizable commands, such as the commands in the aforementioned compression script (Fig. 5). Compared to analyses over static languages, DDU cannot identify global optimizations such as reordering the final command in the script to run first. This lack of optimality is not specific to DDU, but applies to any shell script analysis; in fact, as far as we know there is no sound and precise static analysis for shell scripts.

5 Distributed Scheduling

This section describes how DiSh's scheduler distributes a compiled script to a set of workers. The scheduler is given a dataflow graph that is already parallelized and has HDFS files expanded to sequences of remote file resources (RFRs) representing their blocks. The task of the scheduler is then to distribute this graph with the goal of optimizing performance

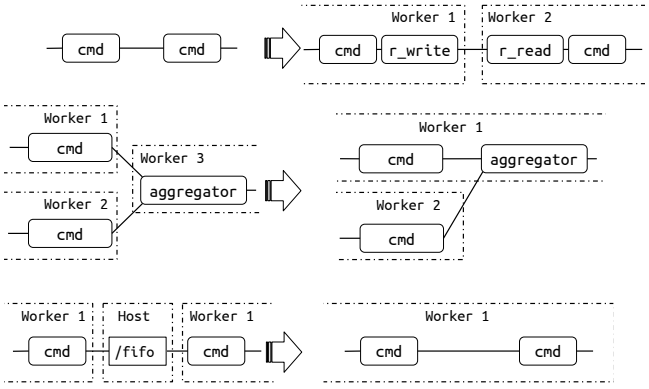


Fig. 6: (Top) Remote writes and reads added during distributed scheduling. **(Mid)** Worker-first aggregation. **(Bot)** Named FIFO teleportation.

by both utilizing available resources and moving computation close to the data. Currently the scheduler knows about the workers in the cluster ahead of time using a configuration file.

The scheduler makes a decision on how to split the graph based on a policy that optimizes performance through co-location of data blocks and the commands that execution over them. The scheduler processes the top-level dataflow graph to generate a set of subgraphs, one for each worker and one for the host machine executing the script. It then replaces edges corresponding to communication channels (*e.g.*, FIFOs, pipes) at the boundaries of each subgraph with remote channels—adding a remote write node on the sender side and a remote read node on the receiver side (see Fig. 6, Top). It also inserts remote reads for subgraph nodes that access files stored on remote workers. The final generated subgraph represents the script fragment that is passed for execution to the user shell running on each worker: the compiled script handles all the redirection to and from local files and the standard input, output, and error streams to and from the worker.

Data-aware scheduling policy: The highest performance overhead when executing distributed shell scripts is networked data movement across workers. DiSh addresses this overhead by introducing a greedy scheduling policy that allocates subgraphs in a way that attempts to minimize data movement across workers. If a data file (or block) is available on a worker, then DiSh maps the maximal dataflow subgraph that starts from that file to that worker—*i.e.*, scheduling as much of the processing as possible on the worker. The scheduler also tracks the amount of work that each worker currently has scheduled, which can vary due to dynamic dependency untangling (§4.3): if a data file is replicated across multiple workers, DiSh chooses the worker with the least amount of pending work to execute that subgraph.

Worker-first aggregation: The distributed dataflow graphs that DiSh executes often contain aggregation (*i.e.*, merge)

nodes, similarly to the reduce stages in Hadoop Streaming. Regardless of the worker on which the aggregation is performed, data from different workers will need to be combined onto a single worker and thus these dataflow nodes will necessarily result in data movement. DiSh prioritizes performing aggregation on one of the participating workers, because workers already contain a subset of the data used in the aggregation (see Fig. 6, Mid). This optimization is particularly beneficial for scripts that filter and aggregate data, often containing commands such as `grep` and `uniq`, because any filtering stages prior to aggregation result in reduced data transfer.

It is worth noting that, absent additional information about commands [49], the location of aggregators involves challenging trade-offs not addressable with a single optimization policy. For scripts that include aggregators that do not reduce data sizes, DiSh's worker-first aggregation optimization risks transferring more data. As DiSh's evaluation confirms (§7), however, worker-first aggregation results in performance benefits for most scripts.

Delegated script concretization: DiSh's scheduler sends workers dataflow subgraphs, encoded in DiSh's intermediate representation, instead of concrete shell scripts ready for execution. Each dataflow subgraph contains holes that workers are expected to fill in, based on the specifics of their local environment. This choice simplifies DiSh's distributed execution, as the scheduler does not need to have up-to-date information about several worker details such as the temporary directories they use. Additionally, this choice enables better resource utilization in a heterogeneous environments with different worker capabilities: a worker can apply another optimization pass to the dataflow subgraph it receives to better manage and utilize its resources.

Named FIFO teleportation: Scripts often use named FIFOs to share data between concurrently executing processes. Named FIFOs introduce a performance challenge, because they are local files that reside on the host machine where the script was executed. Therefore, by default, all data that would normally go through named FIFOs in the original execution would now have to go back and forth between workers and the machine for which the script was developed. DiSh addresses this challenge by observing that named FIFOs are ephemeral, *i.e.*, they maintain no data after the execution of a dataflow region. Based on this observation, DiSh migrates named FIFOs to workers closer to the data, eventually deleting the migrated versions after the dataflow region has finished executing (see Fig. 6, Bot). This transformation, termed FIFO teleportation, improves performance by avoiding unnecessary data movement in scripts that use FIFOs.

6 Runtime Support

DiSh has to address several runtime challenges: communication among workers, identification of HDFS data block locations, and correctness in view of HDFS blocks split independently of newlines—an assumption necessary for several dataflow transformations. This section describes several components of DiSh's runtime that address the above challenges.

Remote FIFO channel: As described earlier (§5), connections between dataflow nodes are instantiated using Unix FIFOs in a single-machine setting. Unfortunately, FIFOs do not support networked operation and thus cannot cross worker boundaries. To address this challenge, DiSh introduces a remote FIFO primitive (RFIFO) that is implemented in Go and uses socket-based communication. RFIFOs are intended to operate identically to FIFOs, *i.e.*, implement the semantics of dataflow graph edges, but with support for operation over the network. They have a unique identifier and two ends—a read end and a write end.

Since shell streams are lazy, *i.e.*, a producer blocks until its consumer requests input, the network link is often not fully utilized, lowering throughput and risking introducing significant latency. To avoid these throughput and latency challenges, DiSh adds two buffer nodes to the dataflow graph: one before the write end of the RFIFO, to allow uninterrupted access to data, and one after the read end of the RFIFO, to force the read to request data. This lazy-to-strict optimization maintains correctness and improves performance in most cases; in rare cases, it may lead to unnecessary data transfer between nodes—*e.g.*, when there is a `head` command right after the read end of an RFIFO.

Port discovery service: As transformations and optimizations are applied during the execution of a script—contrary to most other distributed environments—DiSh's scheduler cannot statically predict which ports will be available at runtime for RFIFOs at each worker: different scripts and script fragments running concurrently during a single execution may collide on port usage. To address that, each DiSh worker implements a port discovery service (PDS) that can be accessed by remote FIFOs to (1) advertise their port, and (2) discover the port that their other end uses. The discovery service is implemented in Go with gRPC [61] and supports a few remote procedure calls (RPCs), central among which are a `put` call for advertisement and a `get` call for discovering the port of a remote end. RFIFOs are extended with gRPC clients to advertise ports among local PDS or identify the ports corresponding to their other end by querying the PDS of the respective worker. By deferring port selection until runtime execution, DiSh's port discovery service facilitates loose subgraph coupling and simplifies remote subgraph execution on multiple workers.

HDFS data retrieval: During transformations, the DiSh compiler (§4) needs to retrieve information about HDFS paths to expand them into block sequences. This expansion happens on a critical runtime path and thus needs to be efficient. A prior implementation of DiSh invoked this expansion on every HDFS path using a shell command—by wrapping `fsck`, a command offered by HDFS API for querying the health of

the disk in the cluster, returning information about a file and its partitioning into blocks. This implementation ended up incurring significant latency (> 1s), and thus DiSH switched to the web API reducing expansion to sub-10ms latency.

Enforcing logical block boundaries: A key challenge when processing separate file blocks in HDFS is the mismatch between compiler assumptions about the block shape and how blocks are actually stored in HDFS: HDFS blocks might not be split on newline boundaries, but the parallelizing transformations performed by the DiSH compiler (§4) assume that all blocks are logically separated by newlines. This assumption is crucial and depends on the way commands process their input, *e.g.*, `sort` processes its input line by line, and therefore would require a significantly more complex parallelization transformation if its input could be split at arbitrary points. Developing complex custom parallelization transformations for each command would be infeasible in practice due to the sheer number of available commands and would not allow DiSH to reuse the parallelization transformations developed for PaSH [64].

Instead of relaxing the compiler assumption, DiSH addresses the mismatch by ensuring it holds during script execution using additional runtime support. DiSH implements a distributed file reader (DFR) primitive that runs as a service on every worker. The DFR service ensures that parallel dataflow nodes only process batches that are split in newline boundaries, independent of how the actual physical blocks are split—providing the illusion of a logical block that ends at a newline to its consumer. Given a distributed file path, DFR reads the local file or block from the worker's disk going beyond the first newline character in its block. If the block is not terminated with a new line, then the DFR communicates with the reader of the next block (and potentially any readers after that), returning a complete logical block to its consumer. When a compiled dataflow graph is translated back to a script, DiSH prefixes file paths with a command invoking a DFR client that communicates with the relevant DFR service to retrieve the relevant logical block. Both service and client are implemented in Go, communicating using gRPC and protobufs [21].

7 Evaluation

We are interested in evaluating two aspects of DiSH: (1) its performance, and (2) its compatibility with Bash.

Experiments: We perform four experiments using several real-world shell scripts taken from a variety of sources (Tab. 2). The first two experiments focus on the performance gains (§7.1) achieved by DiSH's distribution on (1) a 4-node on-premise cluster, and (2) a 20-node cloud deployment—both over a variety of benchmarks and workloads. We compare DiSH's performance against (1) GNU Bash [50], the *de facto* sequential shell-script execution environment; (2) Apache Hadoop Streaming [25] (AHS), a production-grade distributed data-processing framework that supports language-agnostic executables; and (3) in the case of the 4-node setup, PaSH [32, 64], a shell-script parallelization system from the Linux Foundation. PaSH's parallelism benefits make it a likely alternative to DiSH for smaller clusters, where DiSH's anticipated benefits of distribution might be smaller, but this likelihood diminishes as the size of the cluster grows.

Tab. 2: Benchmark summary. Summary of all the benchmarks used to evaluate DiSH, and their characteristics.

Benchmark	Scripts	Pure HS	LOC	Input	Source
1 Classics	10	7/10	123	3G	[5, 6, 31, 39, 59]
2 Unix50	34	30/34	142	21G	[7, 34]
3 COVID-mts	4	4/4	79	3.4G	[62]
4 NLP	21	-	306	120 books	[9]
5 AvgTemp	1	1/1	31	3.6G	[68]
6 MediaConv	2	-	35	0.8 & 0.4G	[49, 56]
7 LogAnalysis	2	-	63	0.7 & 1.3G	[49, 56]
8 FileEnc	2	-	44	1.3G	[41]

The last two experiments evaluate DiSH's dynamic dependency untangling (§7.2) and DiSH's correctness (§7.3), *i.e.*, its compatibility with respect to Bash across all scripts and the POSIX shell test suite.

Benchmarks: We use 8 sets of real-world benchmarks, totaling 76 shell scripts and 823 LoC. Classics and Unix50 contain classic and recent (c. 2019) scripts that make heavy use of Unix and Linux built-in commands. COVID-mts contains four scripts used to analyze real telemetry data from mass-transit schedules during a large metropolitan area's COVID-19 response. NLP contains several scripts from Unix-for-poets, a tutorial for developing programs for natural-language processing out of Unix and Linux utilities. AvgTemp contains a large script downloading and processing multi-year temperature data across the US. MediaConv contains two scripts that process, transform, and compress video and audio files. LogAnalysis contains two scripts that apply typical system-administration and network-traffic analyses over log files. Finally, FileEnc contains aliases that encrypt and compress files.

Baselines and implementations: Bash, PaSH, and DiSH executed every shell script completely unmodified. Apache Hadoop Streaming (AHS) posed significant expressiveness limitations. Only 42 scripts in Classics, Unix50, COVID-mts, and AvgTemp out of 76 scripts can be implemented natively (Tab. 2, col. Pure HS). Another 7 scripts required manual porting by splitting them into mappers, reducers, and additional components: These components were not available natively by AHS—for example, components for reading from two pipelines for `diff.sh` and for sorting after the reducer for `bigrams.sh` (both in Classics). During porting, we put significant care to avoid limiting AHS's parallelism: we modified 3 AHS scripts in Classics to help HS introduce additional

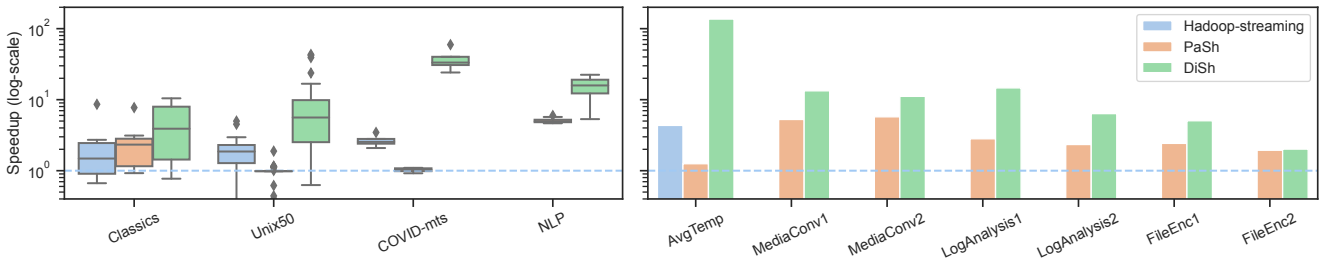


Fig. 7: DiSh performance on a 4-node cluster. DiSh speedup (*vs.* PaSh and Hadoop Streaming whenever possible) over Bash for Tab. 2 rows 1–4 (left, box) and 5–8 (right, bar) (*Cf.*§7.1). (Log y-axis; higher is better.)

parallelism—for example, we manually expanded `tr -cs` into `tr -c | grep -v` (both stateless). None of the scripts in NLP, MediaConv, or LogAnalysis can be implemented in AHS as they perform processing in loops, the iterations of which depend on the files in a statically indeterminable directory (see Fig. 5) and are therefore not expressible in AHS. We attempted to replace the body of the loop with an AHS invocation but the startup overhead ended up dwarfing the execution time by a factor of ten on average.

Hardware & software setup: The 4-node cluster consists of four 6-core Intel(R) Core(TM) i7-10710U CPU nodes each with 64GBs of RAM, located in the same room and connected with an average bandwidth of 90.8 Mbits/sec. The 20-node deployment consists of xl170 Cloudlab [15] nodes, each equipped with 10 × Intel Core E5-2640 2.4 GHz CPUs and 8GB of memory. Single-machine shells (Bash & PaSh) were evaluated on a machine with 20 × 2.80GHz Intel(R) Core(TM) i9-10900 CPUs and 32GB of memory.

For ease of deployment and reproducibility, we used Docker `swarm` to deploy (1) HDFS, and (2) the DiSh runtime. The containers were created using the standard Ubuntu 18.04 image. We use Bash v.5.0.3, PaSh v.6e2ecba, and HDFS/Hadoop streaming version 3.2.2. We explicitly disabled checksum verification from HDFS in all configurations, scripts, and measurements. All scripts were executed completely unmodified, using environment variables, loops, and other shell constructs. To minimize statistical non-determinism we repeated the experiments several times noticing imperceptible variance.

The DiSh implementation comprises 6784 lines of Python (preprocessor, compilation server, expansion, compiler, and parser), 1011 lines of shell code (JIT engine and various utilities), and 1174 lines of C (commutativity primitives, and other runtime components). All counts include only semantically meaningful lines of code.

7.1 Performance

How does DiSh's distributed perform on small on-premise clusters and multi-node cloud deployments, and how does it compare to state-of-the-art systems?

Tab. 3: DiSh performance in 20-node cloud deployment. DiSh speedup over Hadoop Streaming for scripts that AHS supports.

DiSh speedup over AHS						
Benchmark	Avg	Min	25th	50th	75th	Max
Classics	2.74×	0.92×	2.41×	2.60×	2.85×	6.55×
Unix50	6.64×	0.91×	2.85×	5.38×	10.4×	16.9×
COVID-mts	10.4×	6.64×	8.91×	9.27×	-	16.8×
AvgTemp	7.85×	-	-	-	-	-

Results: Fig. 7 (note the log y-axis) shows the performance of DiSh, PaSh, and AHS on a 4-node on-premise cluster across all benchmarks of Tab. 2. Box plots (left) show result quartiles for multi-benchmark suites (Tab. 2, rows 1–4) and bars (right) show results for individual scripts (Tab. 2, rows 5–8). Across all benchmarks, DiSh achieves an average speedup of 13.6× (*vs.* 2.55× for PaSh and 2.1× for AHS) and a maximum speedup of 136.3× (*vs.* 7.8× for PaSh and 8.6× for Hadoop Streaming). The average execution time of all scripts on Bash is 299s, ranging from 1s for `34.sh` in Unix50 to 2840s for `nfa-regex.sh` in Classics. DiSh is only slower than Bash (737s vs 568s) in the case of `diff.sh` from Classics, for which AHS is even slower (766s). DiSh achieves a performance comparable to Bash (1-2s) in `4.sh` and `34.sh` from Unix50, because both perform a short-running `head`.

Tab. 3 shows the speedup of DiSh *over AHS* on a 20-node Cloudlab deployment across all scripts implementable with AHS (Classics, Unix50, COVID-mts, AvgTemp). Across all benchmarks, DiSh achieves an average speedup of 6.17× and a maximum speedup of 16.95× over AHS. DiSh is slower than AHS only for three scripts: `nfa-regex.sh` from Classics (0.92×), `29.sh` and `30.sh` from Unix50 (0.91× and 0.94×).

Across all scripts in both deployments, DiSh's overheads (startup cost, dynamic orchestration, preprocessing, compilation, scheduling) take less than 1 second.

Discussion: DiSh is faster than Bash, PaSh, and AHS across Tab. 2's suites (rows 1–4) with respect to average, and across all of Tab. 2 individual benchmarks (rows 5–8)—often by a significant margin (*e.g.*, 134× for AvgTemp against PaSh).

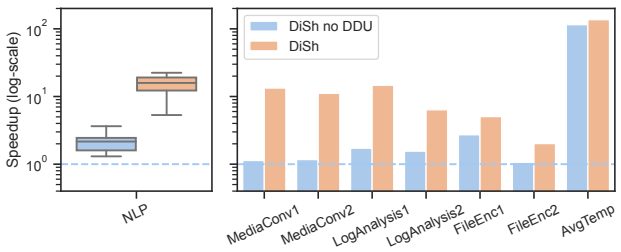


Fig. 8: Dynamic dependency untangling. DiSh speedup over Bash when toggling DDU (higher is better).

DiSh's (and PaSh's) speedup over Bash is due to parallelism. DiSh's speedup over PaSh is due to DiSh' co-location of data and computation: PaSh cannot offload computation and thus first gathers all data onto a single machine—a time-consuming stage—and then starts processing in parallel. DiSh is slower than Bash only for `diff.sh`, because (1) it is not highly parallelizable and (2) it performs no filtering, *i.e.*, its output is the same size as its input. In contrast to Bash, which simply fetches all data and processes it locally, DiSh tries to allocate most commands on the workers, but this leads to increased data movement since moving data between workers does not avoid sending the whole output to the client.

DiSh's speedup over AHS is due to a few different reasons. One reason is the increased expressiveness of DiSh's dataflow model: DiSh accepts and parallelizes complete scripts, discovering more opportunities for parallelism. Many of the AHS scripts are broken into multiple map and reduce stages, often leaving pipeline parallelism and data parallelism unexploited. Another reason is DiSh's dynamic independence discovery, which allows for additional parallelism and better utilization of resources—in ways that AHS does not support; we zoom into these benefits below (§7.2). In the Cloudlab deployment, DiSh is (marginally) slower than AHS in only two cases: (1) a script that is embarrassingly parallel and thus implementable in AHS using only a single mapper (`nfa-regex.sh`), and (2) two scripts in Unix50 that see slightly more benefits from our manual, hand-optimized AHS rewrite than they do from DiSh's automated distribution.

We found porting scripts to AHS a serious challenge. Many scripts required significant manual effort, resulted in multiple error-and-fix cycles, and led to script size increases. To overcome AHS's expressiveness limitations, we had to modify a few scripts in unintuitive ways—often combining plain Bash scripts with AHS mappers and reducers. These modifications made scripts significantly more complex and compounded the effort to test and maintain them. Instead, DiSh distributed scripts successfully without any such challenges.

7.2 Dynamic Dependency Untangling

What is the speedup due to dynamic dependency untangling?

Results: Figure 8 shows DiSh's speedup over Bash with and without dynamic dependency untangling (DDU, § 4.3). It excludes scripts that contain a single dataflow, for which DDU is not applicable. DiSh's average speedup over DiSh-w/o-DDU is 6.9×, ranging between 1.2–13.9×.

Discussion: Enabling DDU improves performance significantly across all relevant scripts, by running independent dataflow regions in parallel. This allows DiSh to expose parallelism not just within data pipelines but across them, improving utilization. DDU also improves the distributed execution of scripts that operate on many files, many or most of which are small enough to fit on a single HDFS block.

DDU is the main reason why DiSh gets an edge over Bash on scripts that (1) have implicit independences that are not highly parallelizable, and (2) operate on small data that incur imperceptible data-movement costs. Examples of such scripts include MediaConv1 and FileEnc2.

7.3 Correctness

What is DiSh's output compatibility with respect to Bash?

Results: To check the correctness of DiSh across all benchmarks, we check that its stdout and exit status are equivalent to the ones produced by Bash. Across all benchmarks, totaling over 650 millions lines (18GB) of output, DiSh produces the same output and exit status as Bash.

We additionally execute the complete POSIX shell-test suite to evaluate DiSh's compatibility with Bash. Out of all relevant tests, DiSh diverges from Bash in two cases and only with respect to the exit status it returns: both exit with an error, but Bash returns 1 whereas DiSh returns 127, which is outside of the POSIX mandated exit status range between 1–125. The reason is that DiSh always invokes the underlying Bash interpreter using the `-c` flag to set the `$0` variable, and Bash (contrary to most other shells, *e.g.*, dash, ksh, mksh, sash, Smoosh, yash, zsh) exits with 127 in particular failing cases when called with `-c`.

Discussion: All benchmarks in Tab. 2 were executed with DiSh repeatedly. After hundreds of runs over several weeks, we observed dozens of different execution orders. Comparing the output on every run provides significant confidence about the correctness of the resulting distributed execution. The POSIX test suite mostly evaluates the correctness of dynamic orchestration (§3), as it does not feature many opportunities for parallelization and features no opportunities for distribution.

8 Related Work

DiSh is related to a large body of prior work.

Distributed data processing: Several environments assist in the development of distributed software systems: distributed computing frameworks [11,44,45,57,71] and domain-specific

languages [3, 8, 13, 40, 42] simplify the development of distributed systems that fall under certain computational classes such as batch processing, stream processing, *etc.* These systems deal with many of the challenges of distribution, but require developers to (re)write their computations manually in models that differ significantly from UNIX shell programming.

Hadoop Streaming and Dryad Nebula are abstractions that allow using third-party language-agnostic components similar to the UNIX shell, atop cluster-computing engines (Hadoop and Dryad, respectively). Both require their users to understand and rewrite their shell scripts using the abstractions provided by each framework. DISH can operate on arbitrary shell scripts automatically, without requiring any manual effort from its users.

Distributed shells and tools: Several packages expose commands for specifying parallelism and distribution on modern UNIXes—*e.g.*, `qsub` [19], SLURM [69], calls to GNU `parallel` [58]. Different from DISH, their effectiveness is predicated upon explicit and careful invocation and is limited to embarrassingly parallel (and short) programs. Often, these commands provide options to support an array of special sub-cases—a stark contradiction to the celebrated UNIX philosophy. For example, `parallel` contains flags such as `--skip-first-line`, `-trim`, and `--xargs`, that a UNIX user can achieve using `head`, `sed`, and `xargs`; it also includes other programs with complex semantics, such as the ability transfer files between computers, separate text files, and parse CSV. DISH embraces the UNIX philosophy, attempting to rewrite shell programs to leverage distributed infrastructure.

Several shells [14, 38, 56] add primitives for non-linear pipe topologies—some of which target distribution. Here too, however, developers are expected to manually rewrite scripts to exploit these new primitives.

POSH [49] is a recent shell for scripts operating on NFS-stored data. It brings pipeline components closer to the data on which they operate, but operates only on shell pipelines that are fully expanded—*i.e.*, ones that do not use dynamic features. DISH operates on shell scripts that use (1) any POSIX composition primitive, and (2) the full set of dynamic features present in the UNIX shell.

Distributed operating systems: There is a long history of networked and distributed operating systems [4, 12, 43, 46, 48, 51–53, 67]. These systems offer abstractions that (1) are similar, but not identical, to the ones offered by UNIX, (2) operate at a lower level of abstraction (*e.g.*, that of system calls, rather than shell primitives), and (3) often aim at simply hiding the network rather than offering scalability benefits. Instead of implementing full-fledged distributed operating system, DISH shows that a thin but sophisticated rewriting-based shim can operate on completely unmodified programs, avoid requiring any user input, and achieve significant speedups by executing fragments in parallel across nodes.

Annotation-based transformations: Recent systems [47, 65, 70] lower the developer effort of scaling out program components by performing program transformations based on user-provided annotations. These systems operate in single-language environments, offering declarative DSLs for tuning the semantics of the resulting distributed program. DISH uses a similar approach, leveraging command annotations from prior projects [49, 64], but operates on-the-fly—within an environment that makes extensive use of dynamic features and that allows combining components from multiple languages.

PASH-JIT [32] parallelizes scripts by dynamically interposing between a shell script and the underlying shell interpreter. This kind of interposition offers significant performance benefits without jeopardizing correctness, *i.e.*, maintains compatibility with the underlying shell interpreter. DISH uses similar insights and interposition architecture, but operates on a distributed multi-node setting and addresses challenges that are specific to this setting—such as integration with a distributed file system and distributed environment passing.

Cloud build systems: Several cloud build systems [1, 2, 17, 27] distribute and parallelize the execution of large builds by constructing dependency graphs using dependency information explicitely specified by their users. Contrary to these systems, DISH operates on general shell programs without exploiting domain-specific information—*e.g.*, build dependencies—and by taking a just-in-time approach that resolves dependencies during the execution of the script.

Correct distribution of dataflow graphs: The DFG is a prevalent model in several areas of data processing, including batch- and stream-processing. Systems implementing DFGs often perform optimizations that are correct given subtle assumptions on the dataflow nodes that do not always hold, introducing erroneous behaviors. Recent work [28, 33, 37, 54] attempts to address this issue by performing optimizations only in cases where correctness is preserved, or by testing that applied optimizations preserve the original behavior. DISH uses its dynamic orchestration to achieve compatibility with the underlying shell and then achieves correct distribution on a per-region level by building on prior work on provably correct transformations for order-aware dataflow graphs [26]. Similarly to other automated shell script transformation works [49, 64], DISH's correctness is predicated upon the correctness of the annotations describing commands.

Resurgence of shell research: Recent shell research [10, 23, 24, 32, 36, 41, 49, 55, 56, 64] highlights renewed interest in shell scripting both as a vehicle for impactful research and as a target worthy of scientific attention. We see DISH as a natural continuation of the insights and research behind recent systems [24, 26, 32, 49, 64], allowing other researchers to leverage DISH's POSIX-compliant high-performance dynamic distribution in their future work.

9 Discussion

Programmability: An important consideration with any automated system is how it affects programmability, and specifically the ability to debug a misbehaving program or to test a program for correctness. DiSH does not negatively affect the developer experience compared to a shell: a developer can use a combination of the many existing tools and commands—*e.g.*, `head` and `grep`—as they would normally do to inspect their script's output and determine what is wrong. When it comes to shell scripts intended for distributed environments, DiSH in fact improves developer experience: a developer may use the same set of commands for local or distributed interactions—*e.g.*, to inspect and project parts of a file, regardless of whether that file is stored in HDFS or the local system. Furthermore, developers using DiSH can reap the scalability benefits of distribution in analyzing or testing scripts by automatically scaling out load to multiple computers.

Command annotations: DiSH's transformations depend on the existence of command annotations. To maintain soundness, DiSH will avoid compiling and distributing a script region if some command lacks annotations. To increase the distributability of their scripts, DiSH users could opt for more constrained commands—*e.g.*, `cut` instead of `awk` for data projection—and thus enjoy tighter annotations and more applicable optimization transformations. The correctness of applying DiSH's transformations depends on the correctness of these annotations, and thus annotations are currently expected to be authored by command developers or other experts (but not script developers). Developing automation for testing or synthesizing correct annotations is an interesting avenue for future research that would benefit several systems that use them—*e.g.*, DiSH, PaSH [64], and POSH [49].

Fault tolerance: DiSH does not tolerate failures such as worker aborts or network partitions (§1). In such cases users are expected to rerun their scripts as shell users normally do in the non-distributed case. Achieving fault tolerance in the context of general shell scripts is in fact particularly challenging due to the prevalence of black-box components that may perform arbitrary side-effects. A fault-tolerant version of DiSH should be able to track all these side-effects and re-execute them appropriately when a script fails. This is in contrast to constrained cluster computing frameworks such as MapReduce and Spark that have precise information about the inputs and outputs of purely functional program components enabling simplified re-execution of dependency graphs (lineage) in the presence of failures. DiSH's design however combined with incremental script execution [10] creates an opportunity for addressing this challenge with a hybrid approach: employ conventional fault tolerance approaches for script fragments with annotation information, and instrument the rest of the script to capture and replay its side-effects appropriately in cases of failure.

Conclusion: DiSH is the first system able to distribute unmodified shell scripts that use (1) any POSIX composition primitive, (2) the full set of dynamic features present in the Unix shell, and (3) distributed file systems such as HDFS. DiSH uses a dynamic orchestration approach that instruments a given script and dynamically distributes it at runtime to then execute it using the underlying shell interpreter. As a result, DiSH avoids modifications to shell scripts and maintains compatibility with existing shells and legacy functionality. Evaluated against several alternatives available to users today, DiSH offers significant speedups, requires no developer effort, and handles arbitrary dynamic behaviors pervasive in shell scripts. DiSH is open-source software, available by the Linux Foundation.

Acknowledgments

We would like to thank Ayush Bhardwaj, Felix Stutz, Hannah Gross, Lily Tsai, Malte Schwarzkopf, Michael Greenberg, Neil Ramaswamy, the Brown Systems Group, the NSDI 2023 reviewers, and our shepherd, Rebecca Isaacs, for discussions and feedback on the paper. This material is based upon work supported by DARPA contract no. HR00112020013 and no. HR001120C0191, and NSF award CCF 2124184.

References

[1] Bazel dynamic execution. https://bazel.build/remote/dynamic, 2022. [Online; accessed Feb 1, 2022].

[2] Google cloud build. https://cloud.google.com/build/docs/overview, 2022. [Online; accessed Feb 1, 2022].

[3] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[4] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4):361–372, 1998.

[5] Jon Bentley. Programming pearls: A spelling checker. *Commun. ACM*, 28(5):456–462, May 1985.

[6] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[7] Pawan Bhandari. Solutions to unixgame.io, 2020. Accessed: 2020-04-14.

[8] Martin Biely, Pamela Delgado, Zarko Milosevic, and Andre Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

[9] Kenneth Ward Church. Unix™for poets. *Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods*, 1994.

[10] Charlie Curtsinger and Daniel W Barowy. Riker: Always-Correct and fast incremental builds from simple specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 885–898, 2022.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[12] Sean Dorward, Rob Pike, David L Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom. Inferno. In *Proceedings IEEE COMPCON 97. Digest of Papers*, pages 241–244. IEEE, 1997.

[13] Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, 2016.

[14] Tom Duff. Rc—a shell for plan 9 and unix systems. *AUUGN*, 12(1):75, 1990.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[16] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[17] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. Cloudbuild: Microsoft's distributed and caching build service. In *SEIP*. IEEE - Institute of Electrical and Electronics Engineers, June 2016.

[18] Jim Garlick. pdsh. https://github.com/chaos/pdsh, 2022. [Online; accessed September 15, 2022].

[19] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36. IEEE, 2001.

[20] Inc. GitHub. The 2021 state of the octoverse: Top languages over the years. https://octoverse.github.com/#top-languages-over-the-years, 2021. [Online; accessed June 1, 2022].

[21] Google. Protocol Buffers, 2022. Accessed: 2022-06-01.

[22] Michael Greenberg. libdash. https://github.com/mgree/libdash, 2019. [Online; accessed December 6, 2021].

[23] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the posix shell: Smoosh: the symbolic, mechanized, observable, operational shell. *Proc. ACM Program. Lang.*, 4(POPL):43:1–43:30, January 2020.

[24] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix shell programming: The next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Hadoop. Hadoop streaming. https://hadoop.apache.org/docs/r1.2.1/streaming.html, 2022. [Online; accessed September 15, 2022].

[26] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An order-aware dataflow model for parallel unix pipelines. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[27] Jason Hickey and Aleksey Nogin. Omake: Designing a scalable build process. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, pages 63–78, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[28] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46:1–46:34, March 2014.

[29] Lluis Batlle i Rossell. *tsp(1) Linux User's Manual*. https://vicerveza.homeunix.net/ viric/soft/ts/, 2016.

[30] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[31] Dan Jurafsky. Unix for poets, 2017.

[32] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, Just-in-Time shell script parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 769–785, Carlsbad, CA, July 2022. USENIX Association.

[33] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: Differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[34] Nokia Bell Labs. The unix game—solve puzzles using unix pipes, 2019. Accessed: 2020-03-05.

[35] Haoyuan Li. *Alluxio: A virtual distributed file system*. University of California, Berkeley, 2018.

[36] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, pages 9–15, 2021.

[37] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 670–685, New York, NY, USA, 2019. ACM.

[38] Chris McDonald and Trevor I Dix. Support for graphs of processes in a command interpreter. *Software: Practice and Experience*, 18(10):1011–1016, 1988.

[39] Malcolm D McIlroy, Elliot N Pinson, and Berkley A Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[40] Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, eventually consistent computations with crdts. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[41] Jürgen Cito Michael Schröder. An empirical investigation of command-line customization. *arXiv preprint arXiv:2012.10206*, 2020.

[42] Adrian Mizzi, Joshua Ellul, and Gordon Pace. D'artagnan: An embedded dsl framework for distributed embedded systems. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–9, 2018.

[43] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[44] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[45] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[46] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[47] Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 291–305, New York, NY, USA, 2019. ACM.

[48] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[49] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A data-aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631, 2020.

[50] Chet Ramey. Bash reference manual. *Network Theory Limited*, 15, 1998.

[51] Richard F Rashid and George G Robertson. *Accent: A communication oriented network operating system kernel*, volume 15. ACM, 1981.

[52] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. Overview of the chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70. Seattle WA (USA), 1992.

[53] Jan Sacha, Jeff Napper, Sape Mullender, and Jim McKie. Osprey: Operating system for predictable clouds. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6. IEEE, 2012.

[54] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2):504–517, Feb 2015.

[55] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic synthesis of parallel unix commands and pipelines with kumquat. corr abs/2012.15443 (2021). *arXiv preprint arXiv:2012.15443*, 2021.

[56] Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[57] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. Jetstream: a self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pages 1–8, 2015.

[58] Ole Tange. Gnu parallel—the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[59] Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. No Starch Press, 2004.

[60] Elixir Core Team. Elixir. https://elixir-lang.org/.

[61] The gRPC Authors. grpc, 2018. Accessed: 2019-04-16.

[62] Eleftheria Tsaliki and Diomidis Spinellis. The real statistics of buses in athens. https://bit.ly/3s112R5, 2021.

[63] Junichi Uekawa. dsh. https://www.netfort.gr.jp/~dancer/software/dsh.html.en, 2022. [Online; accessed September 15, 2022].

[64] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. Ignis: Scaling distribution-oblivious systems with light-touch distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1010–1026, New York, NY, USA, 2019. ACM.

[66] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[67] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, 1983.

[68] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[69] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[70] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. Offload annotations: Bringing heterogeneous computing to existing libraries and workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 293–306, 2020.

[71] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication

Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho*, Michael Ferdman, Peter Milder, Shuai Mu
*Stony Brook University, *Microsoft*

Abstract. State machine replication (SMR) is a core mechanism for building highly available and consistent systems. In this paper, we propose Waverunner, a new approach to accelerate SMR using FPGA-based SmartNICs. Our approach does not implement the entire SMR system in hardware; instead, it is a hybrid software/hardware system. We make the observation that, despite the complexity of SMR, the most common routine—the data replication—is actually simple. The complex parts (leader election, failure recovery, etc.) are rarely used in modern datacenters where failures are only occasional. These complex routines are not performance critical; their software implementations are fast enough and do not need acceleration. Therefore, our system uses FPGA assistance to accelerate data replication, and leaves the rest to the traditional software implementation of SMR.

Our Waverunner approach is beneficial in both the common and the rare case situations. In the common case, the system runs at the speed of the network, with a 99th percentile latency of 1.8 μs achieved without batching on minimum-size packets at network line rate (85.5 Gbps in our evaluation). In rare cases, to handle uncommon situations such as leader failure and failure recovery, the system uses traditional software to guarantee correctness, which is much easier to develop and maintain than hardware-based implementations. Overall, our experience confirms Waverunner as an effective and practical solution for hardware accelerated SMR—achieving most of the benefits of hardware acceleration with minimum added complexity and implementation effort.

1 Introduction

Variants of State Machine Replication (SMR) are responsible for all reliable, consistent, and highly available online services. SMR is at the core of massive-scale systems such as cloud infrastructure coordination services [6, 26], large-scale distributed databases [13,25,63,69], and many other systems that require high availability and consistency [2, 14]. Due to their central nature in critical infrastructure, SMR implementations must be extremely robust and resilient. At the same time, the performance characteristics of the SMR dictate the performance of the overall service. As a result, mechanisms for reducing SMR operation latency and increasing throughput have received significant research attention.

A fundamental requirement of SMR implementations is that networked hosts must exchange multiple messages to agree on the shared state. While implementations that use traditional NICs and process all packets through the OS network stack are the most straight-forward, their performance is bounded by the large amount of CPU time spent on packet processing [8, 22, 28, 46, 57, 58, 68], limiting the throughput and drastically impacting the operation latency due to many traversals of the software network stack.

To overcome the CPU bottleneck, researchers have begun exploring hardware acceleration of network processing. For example, a recent work demonstrated the ZooKeeper broadcast protocol implemented entirely in reconfigurable hardware on an FPGA (Field-Programmable Gate Array). This implementation is able to approach line-rate throughput with operation latencies that are only marginally higher than the on-the-wire latency of the messages [30].

Unfortunately, although it is an impressive demonstration of hardware-acceleration capabilities, the hardware-only approach is too complex and brittle for practical deployment. Despite the improvements in the ease-of-use of hardware development toolchains, the ZooKeeper FPGA implementation required significant expertise, including a hardware version of the TCP/IP stack and all of the protocol details such as leader election and failure recovery. Implementing and debugging distributed protocols in hardware is significantly more difficult than user-level software implementations. Moreover, consensus algorithms—the core of SMR—are well known for being complex and error-prone in design and implementation. Properly capturing all corner-case behaviors in a hardware implementation is challenging and difficult to verify.

We observe that the complexity of SMR is actually in the uncommon routines. Indeed, the most common operation, the one that limits throughput and dictates operation latency, is extremely simple: a leader node receives requests and broadcasts them to the follower nodes, locally committing requests only after receiving acknowledgements from the followers. Other SMR routines, such as leader election and failure recovery, are indeed considerably more complex.However, these complex operations are used only in special circumstances, such as system bootstrap or replica failures. These operations are rare, not performance critical, and their traditional software implementations are fast enough for all practical purposes.

We propose a new approach to accelerate SMR by creating a hybrid hardware/software organization that implements only a small, simple, but performance-critical portion of the protocol in hardware and leaves the vast majority of the imple-

mentation in traditional user-level software. We showcase Waverunner, our approach for hardware acceleration of Raft [52], a well-known consensus protocol, where the entirety of the robust software implementation remains intact, adding only an extra high-performance hardware-accelerated path for the common-case operation. The Waverunner approach permits a clean separation of the complex operations from simple ones in the Raft application software, and reduces the complexity of transport protocol handling. Waverunner uses UDP for the common routines, but leaves all complex cases, such as error handling and view changes, to the traditional TCP-based software. Moreover, by restricting the Waverunner hardware to only the common case, we are able to leverage FPGA HLS (high-level synthesis) tools to automatically generate the hardware from its C++ description, significantly reducing implementation and adoption effort. Although our prototype system is based on Raft, the Waverunner technique is generic and can be easily adapted for other distributed protocols.

Waverunner is notable for its performance and simplicity. We achieve line-rate operation of Raft (85.5 Gbps after accounting for Ethernet frame overheads on a 100 Gbps network), with the vast majority of the SMR broadcast requests handled in three wire-delay latencies (median latency is 1.8 μs). At the same time, we retain the robustness, flexibility, and completeness of a software implementation—Waverunner includes a fully operational implementation of Raft without hardware acceleration and can smoothly transition between hardware-accelerated and software-only modes. In a failure test, Waverunner can recover from a leader crash within 1 second. While the implementation of Raft is by no means simple, leaving it in software is appealing for debugging, upgrades, and maintainability. Moreover, the 220-line HLS-based C++ implementation of the hardware, made possible by limiting the hardware only to a simple core routine, greatly simplifies modification and maintainability of the hardware components compared to a full-hardware implementation.

2 Background & Motivation

In this section, we briefly introduce the concepts of state machine replication and FPGAs, as well as the key idea and motivation of our Waverunner approach.

2.1 State Machine Replication

State machine replication (SMR) is the standard approach to build highly consistent and available systems [6, 13, 26]. It aims to provide a consistent view among replicas while tolerating replica failures in a practical environment where messages can be arbitrarily delayed and there are no perfect failure detectors. In the most common model, SMR replicates a sequence of log entries that contain the operations to be executed by each replica. The application is modeled as a deterministic state machine so that all replicas will have identical states after executing the same sequence of operations.

At the center of an SMR system is a consensus protocol, which is known to be complex and delicate. Most consensus protocols share a common leader-follower model, from early academic ones (e.g., Viewstamped Replication [51] and Paxos [40]) to more recent ones that are widely adopted in industry (e.g., Zookeeper Atomic Broadcast [26], and Raft [52]). Despite their differences, these protocols largely follow a two-stage structure: a leader election and recovery stage when the system elects a new leader and synchronize all replicas after possible failures, and a data replication stage when the leader replicates log entries to the followers as new requests arrive.[1]

2.2 Programmable NIC with FPGA

The last three decades have witnessed the emergence of a great mismatch between network speed and CPU performance. The bottleneck of a networked system has gradually moved from the network (NIC and switch) to the CPU and the OS software stack. To mitigate this problem, the use of programmable NICs with Field Programmable Gate Arrays (FPGAs) has emerged. Equipped with on-chip processing units and memory, FPGA-based NICs can process packets at line rate (e.g., 100 Gbps), with stable nanosecond-range latency. In comparison, the traditional software approach can process only several gigabits per second on a CPU core, with latencies measured in milliseconds.

Although capable of high throughput and low latency, FPGA hardware is notoriously difficult to program. Programming FPGAs is particularly challenging because it requires hardware development skills and knowledge of hardware description languages. Although high-level synthesis (HLS) tools make the FPGA programming process more accessible by translating functions from C++ to hardware, these tools are difficult to use effectively when the logic being implemented is complex. As a result, one of our goals in the design of Waverunner is to make sure that all functions that we implement in FPGA hardware are straightforward and simple, so they can be easily implemented with HLS.

2.3 Motivation

As the critical building block of large-scale systems, the performance of SMR has been a focus in many recent studies. High-performance SMR implementations use a wide range of advanced hardware, including programmable NICs with FPGAs [30], RDMA [2], and programmable switches [16,32,42]. In this work, we propose a unique hybrid approach to accelerating SMR with FPGAs: only accelerating the data plane and leaving the control plane, including election and recovery, to

[1] (Multi-)Paxos does not have a universally agreed algorithm, whether to implement it as a two-stage structure depends on the implementation [2, 10].

	Usage Ratio (%)		
	Control plane	Data plane	Application
Our Raft	∼0 (1e-8)	88	12
NuRaft	∼0 (1e-4)	92	8
etcd	∼0 (1e-4)	72	28

(a) CPU cycle breakdown of Raft implementations.

	LOC (approx.)		
	Control plane	Data plane	Application
Our Raft	1500	200	5400
NuRaft	3600	1100	6000
etcd	1700	180	8900

(b) LOC estimates of Raft implementations.

	Usage ratio (%)
Network descriptor read	6.1
Network descriptor write	16.8
Other RPC cost	5.9
Memory allocation	9.6
Reference counting and memory free	13.3

(c) Breakdown of the data plane usage on system-related (non-logic) code in our Raft implementation.

Table 1: Raft Implementation Analysis

be processed by software. Our approach leverages the observation that the data plane dominates the system performance when the system is stable (most of the time), while the recovery part is only used in rare cases and is usually fast (seconds or less), so there is no need to accelerate it.

Table 1a and 1b show a CPU cycle analysis of various Raft implementations, including our own implementation in C++. In our experiments, we set the failure rate to once a year, and observe that the data plane consumes the majority of the CPU cycles while the control plane consumes almost none. Moreover, the majority of data plane usage is on common utilities such as networking, data serialization, memory allocation, etc. (Table 1c). Implementing the data plane in FPGA can avoid these costs. Implementing the control plane with software can minimize the programming effort needed in the hardware; it also allows us to rapidly iterate on the software implementation, as required when developing complex modules.

In this paper, we choose Raft as our acceleration target. This is another advantage of our approach: we can accelerate existing protocols and systems rather than develop entirely new ones. The hardware acceleration in deployment can then be optional. That is, the system can run either with or without the programmable NIC. This approach adds great flexibility in practice. Though we use Raft for our prototype, we believe our approach similarly applies to other common consensus protocols, as they have similar structure [64, 67].

3 Waverunner System Overview

Waverunner takes the standard state machine replication model: it replicates a sequence of operation log entries (messages) identically onto each replica. The target of replication is referred to as the application (e.g., a key-value store or a

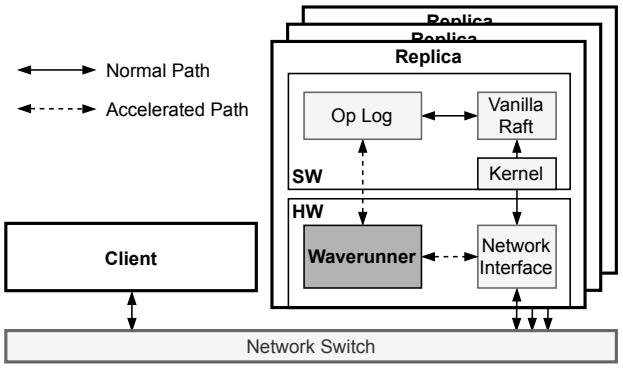


Figure 1: System Design Overview. The solid black lines represent the normal path when Waverunner is disabled, where network packets travel between the application (Vanilla Raft) and NIC via the POSIX interface and Kernel device driver; the dashed lines represent the accelerated path, where the NIC diverts incoming packets to Waverunner, which can send messages into the network via the NIC and/or deliver them directly into a buffer allocated by the application.

lock service). The application must be deterministic; after all replicas process the messages in the order they are recorded in the log, the replicas will all reach the same state. A client for SMR can be either the replicated application itself or an independent client that sends requests to the application. A replica server is either a leader or a follower. Only the leader accepts client requests and replicates these requests to the followers. Followers reject client requests, causing the clients to resend requests to the leader.

System Architecture. The system architecture, including its hardware and software components, is shown in Figure 1. The software includes a complete vanilla Raft implementation that uses conventional TCP sockets provided by the Linux kernel. Additionally, the software can access hardware configuration registers to control the Waverunner hardware. The software can disable Waverunner hardware acceleration, causing all received network packets to be delivered via the conventional network stack. However, if the software enables hardware acceleration, the network interface examines all received packets to identify those carrying Raft messages, and directs packets containing the most common Raft messages (client requests and data replication messages) to the Waverunner hardware protocol handler instead of the kernel network stack. To communicate with the software, the Waverunner hardware writes messages into a pre-allocated user-space log buffer, bypassing the kernel. Uncommon Raft messages and all other network traffic (e.g., ARP requests and ssh connections) are handled by the kernel like with a conventional NIC, regardless of whether Waverunner hardware acceleration is enabled or not.

Typical Waverunner Workflow. Waverunner takes advantage of the Raft leader election protocol to coordinate enabling and disabling hardware acceleration (Figure 2). When the system is first initialized, hardware acceleration is disabled by

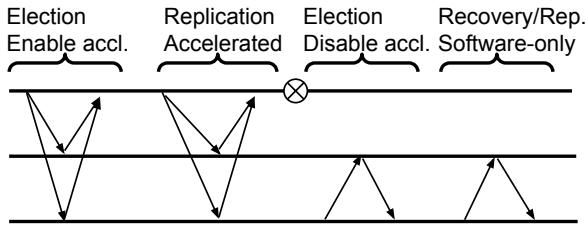


Figure 2: Waverunner Workflow

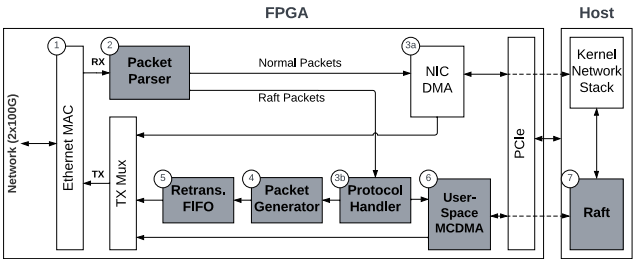


Figure 3: Waverunner Hardware Overview

default. The Raft software triggers a leader election, selecting one of the replicas as the leader, and enables hardware acceleration. The system then replicates data with the assistance of hardware acceleration. The hardware takes care of the data replication operations and deposits the committed log messages into the user-space log buffer for the software application to handle. If a leader failure occurs, the system disables acceleration, conducts another leader election, and performs the requisite Raft protocol actions to resolve the problem before re-enabling acceleration. Other uncommon and complex operations are handled in a similar way. For example, after a series of failure events, the replicas may have diverged log sequences. The leader software will synchronize replicas by re-sending the missing log entries and potentially overwriting existing entries in the followers if necessary. After the complex conditions are addressed and all replicas are in the same state, the system can conduct another leader election and re-enable hardware acceleration.

The safety and consistency properties of SMR guarantee that 1) all replicas will commit the same log entry at the same position in the buffer, and 2) if an operation starts after another one commits, then the two operations will appear in the same order in the logs of all replicas.

4 Waverunner Hardware

In this section, we describe the Waverunner hardware (§ 4.1), explain its hardware data replication operation (§ 4.2), and detail the design of the communication mechanism necessary for Waverunner to achieve high performance (§ 4.3).

4.1 Hardware Architecture

Our Waverunner prototype is based on a traditional PCIe NIC architecture, where a NIC DMA engine, controlled by the host kernel network stack, streams data between the network interface and the host using the PCIe bus. When receiving packets from the network interface, the NIC transfers the packet contents into host memory and raises an interrupt to alert the CPU that the transfer is completed. To transmit packets, the NIC uses the PCIe bus to traverse a queue of packet contents populated in host memory by the software, transferring packets to the network interface.

Waverunner uses a bump-in-the-wire organization to extend the traditional NIC functionality with SMR hardware acceleration, as shown in Figure 3.

When packets arrive over the network ①, they are first streamed through a packet parser ② module to identify packets containing SMR protocol messages. Packets that do not contain SMR messages are streamed to the NIC DMA engine ③a and are handled by the host kernel. If hardware acceleration is enabled and the packet parser detects a supported Raft message, the message is streamed to the Waverunner hardware protocol handler ③b instead of the NIC DMA. The protocol handler performs internal bookkeeping on the protocol state, tracking Raft messages forwarded to the followers and their acknowledgements. If the received messages require one or more packets to be sent out (e.g., client requests must be replicated to the follower nodes), the protocol handler streams the messages to a packet generator module ④, which generates the packets and transmits them into the network. Outgoing packets are buffered in a circular buffer ⑤ that includes retransmission logic; when packet loss or reordering is detected, the protocol handler can signal the buffer to retransmit its contents. Notably, in this case, Raft packets are received and transmitted with minimum latency, entirely without host CPU involvement. Finally, the protocol handler determines if the received message must also be sent via User-Space MCDMA ⑥ to the Raft software for further processing ⑦. For each data replication Raft message, Waverunner will write two operation messages into the user-space log buffer in the host memory: 1) when the client request is received and forwarded to the follower replicas, the request is also written into the log buffer, and 2) when a sufficient number of acknowledgements are received from the followers, a commit operation is written into the log buffer. Application software then processes the log in order and performs the committed operations.

Messages to be delivered to the software are streamed to a descriptor-based, high-performance Multi-Channel DMA (MCDMA) engine that transfers the message contents directly to the user-space Raft software, bypassing the kernel network stack in a similar way to DPDK [21]. Each channel of MCDMA has a ring buffer of descriptors. Each descriptor consists of metadata such as the size and address of the data buffer, complete bit, and the pointer to the next descriptor.

Once the MCDMA completes a transaction, it updates the metadata, allowing the CPU software to consume the data. In the case that Raft traffic arrives too fast to transfer to the host and the MCDMA ring buffer becomes full, the Waverunner hardware will drop the packets before the packet parser, preventing the scenario where a message is processed by the Raft protocol handler but not transferred to the software.

To facilitate coordination between the host software and the hardware acceleration modules, the Waverunner hardware exposes a set of control and status registers (CSRs) accessible from the host software. Some registers, such as follower MAC and IP address, are used to configure the Waverunner system before the operation. An ACC_ENABLE register can be used to enable or disable the accelerated packet processing. Finally, the Waverunner hardware and software use several CSRs to maintain the Raft protocol state. After disabling hardware acceleration, the software can read the latest values of these registers from the hardware. The complete set of Raft protocol registers is listed in Appendix (§ C).

4.2 Raft Leader and Follower Operation

Although Raft is complex, Waverunner implements only the most common operations in hardware (pseudocode shown in Figure 4) and relies on the software to handle uncommon and complex interactions. Uncommon hardware-generated messages such as `AppendEntryReject` are sent to the software via the user-space log without additional processing, while complex messages such as leader election are not identified by the packet parser at all, allowing these messages to be delivered via the traditional NIC DMA (also passing through the OS network stack, and therefore allowing these messages to naturally leverage features such as reliable TCP transport).

When configured to act in a leader role, the hardware accelerator includes protocol handling logic for only two message types: client requests and follower acknowledgements. Upon receiving a client request (Figure 4, lines 2-12), the accelerator updates the Raft protocol state (e.g., `lastLogIndex`, `lastLogTerm`), streams the message contents to the packet generator (to transmit `AppendEntry` messages to the followers), and also sends the message to software. Upon receiving a follower acknowledgement (Figure 4, lines 33-43), the accelerator updates the protocol state and, if the operation is ready for commit (half of the followers have acknowledged), the acknowledgement message is sent to the user-space log. Only one acknowledgement is delivered to the software, subsequent acknowledgements for the same request are ignored. All other protocol messages identified by the packet parser are delivered to the software without updating protocol state and without response packet generation by the accelerator.

The follower role is even simpler, as it handles only `AppendEntry` messages. Upon receiving a message (Figure 4, lines 15-30), the follower first conducts several safety checks, including the `is_leader` check, checking if the previous log

```
1  // FPGA receives client request
2  function FPGA-AppendEntry(fs, op):
3    if fs.isLeader
4      prevLogIndex = fs.lastLogIndex
5      prevLogTerm = fs.lastLogTerm
6      logEntry = makePair(op, fs.currentTerm)
7      push(fs.host.log, logEntry)
8      fs.lastLogIndex++
9      fs.lastLogTerm = fs.currentTerm
10     send <'FPGA-append', op, prevLogIndex,
           prevLogTerm, fs.currentTerm, fs.commitIndex>
           to all except self
11   else
12     reject
13
14 // FPGA receives <'FPGA-append', op,
       prevLogIndex, prevLogTerm, term,
       commitIndex>
15 function FPGA-ReceiveAppend(fs, op, prevLogIndex,
       prevLogTerm, term, commitIndex):
16   if not fs.isLeader
17      and fs.currentTerm == term
18      and fs.lastLogTerm == prevLogTerm
19     if prevLogIndex > fs.lastLogIndex
20       reply with retransmission request
21     else if prevLogIndex < fs.lastLogIndex
22       ignore and return
23     fs.host.commitIndex = commitIndex
24     logEntry = makePair(op, fs.currentTerm)
25     push(fs.host.log, logEntry)
26     fs.lastLogIndex++
27     fs.lastLogTerm = fs.currentTerm
28     reply <'FPGA-appendOK', term, fs.id, fs.
           lastLogIndex>
29   else
30     reply <'FPGA-appendReject', fs.id>
31
32 // FPGA receives <'FPGA-appendOK', term, id,
       lastLogIndex>
33 function FPGA-ReceiveAppendAck(fs, term, id,
       lastLogIndex):
34   if fs.isLeader and fs.currentTerm == term
        and fs.matchIndex[id] < lastLogIndex
35     fs.matchIndex[id] = lastLogIndex
36     if fs.commitIndex < lastLogIndex
37       if a (majority-1) elements in matchIndex
             >= lastLogIndex
38         fs.commitIndex = lastLogIndex
39         fs.host.commitIndex = fs.commitIndex
40         if the request does not read the
             system state (e.g., a blind write
             in a key-value store)
41           notify the client of commit and skip
               the reply from the host
               software (in ApplyLog)
42   else
43     halt and notify host to handle failures
```

Figure 4: Pseudocode of the hardware accelerator.

index and term match (ruling out the case of lost or duplicated packets), and confirming that the message `term` value matches the `currentTerm` variable. Then the follower updates the protocol state, streams the message into the packet generator to produce an acknowledgement (`AppendEntryAck`) or rejection (`AppendEntryReject`) message, and updates its local `commitIndex` according to the message `commitIndex`.

It is worth noting that the protocol-specific Waverunner hardware accelerator actions and logic are intentionally primitive. For an FPGA implementation, the logic of these operations is automatically translated from their C++ description.

4.3 User-Space Log Considerations

The behavior of communication between the hardware and software is critical for high Waverunner performance. Raft messages must be delivered by MCDMA (shown in Figure 3) to the log accessible in user-space software without being a bottleneck in the system. We describe three critical aspects of the design of the User-Space MCDMA block.

First, the MCDMA component requires *fast write response*. In our platform, MCDMA is connected to the PCIe bridge using AXI Memory Mapped (AXI-MM) interfaces, where the MCDMA's AXI master interface writes data to the PCIe bridge's AXI slave interface to transfer data to the user-space buffer. Under the AXI-MM protocol, MCDMA first issues the write address to the PCIe bridge, followed by the write data. After the data transfer is complete, the PCIe bridge sends a write acknowledgement response to the MCDMA. In our experiments, we observed a very high latency (300 to 400 cycles) for the PCIe bridge to send the write acknowledgement response after the data transfer. During this period, MCDMA stops processing descriptors and accepting packets from the protocol handler, negatively affecting the system performance. To solve this problem, we insert a small custom FIFO between the MCDMA and the PCIe bridge. The custom functionality of this FIFO is to send write acknowledgement responses immediately after the write operations are completed on the MCDMA side, thereby hiding the high acknowledgement latency introduced by the PCIe bridge and allowing MCDMA to process descriptors for the other channels.

Second, we introduce *batched MCDMA operation* in the hardware. For each MCDMA operation, in addition to the data transfer, there are also descriptor read and write operations across PCIe. The descriptor reads and writes are overheads, which significantly reduce the effective bandwidth for the actual PCIe data transfers. To minimize the descriptor overhead, we designed a hardware module to batch multiple consecutive message writes into a single transfer to amortize the descriptor overheads, solving the performance bottleneck and improving throughput over PCIe. The batching hardware collects messages until one of two conditions is met: either a pre-configured batch size is reached or a pre-configured timeout is reached without new messages arriving on the given channel. With the second condition, the latency increase caused by batching is negligible.

Finally, although Waverunner hardware acceleration significantly reduces the work that must be done by the CPU of the leader replica, the application code that executes committed operations still consumes CPU resources and can become the bottleneck in the system. One design option is to use a software dispatcher to handle incoming log messages from the hardware and coordinate spreading the handling of the log messages across software threads running on different cores. However, at our target throughput, a software dispatcher would itself become the system bottleneck. Instead, Waverunner shards log messages in hardware, using separate DMA channels to write log messages destined for processing by different application software instances. The leader replica runs multiple application processes, one per core, with each process having its own user-space log buffer into which the hardware deposits Raft messages belonging to the corresponding shard. This approach mirrors the operation of high-performance NICs that allow the software (e.g., DPDK) to install rules into the NIC hardware to steer incoming packets to different descriptor rings or queues to be handled by different cores.

4.4 Transmission with UDP

Our implementation uses UDP to transmit packets between FPGAs. UDP is unreliable for transmission and suffers from packet loss, duplication, and reordering with traditional hardware and software stacks. However, using UDP in the Waverunner hardware greatly reduces the hardware complexity compared to a TCP implementation. To handle the cases of UDP packet loss and reordering, our hardware implements a small retransmission buffer. The buffer, placed between our packet generator and the TX Mux (shown in Figure 3), holds all recently sent packets. In the event of packet loss or reordering being detected in `AppendEntry`, the protocol handler requests the packet generator to create a retransmission request packet (Figure 4, lines 19-20). When a retransmission request arrives at the retransmission buffer, instead of writing it into the buffer, the control logic triggers a retransmission of all packets currently in the buffer. The retransmission is finished when all the packets in the buffer have been transmitted. During the retransmission, incoming packets continue to be written to the tail of the buffer. The buffer is 256 KB, ensuring that in the worst-case scenario in our system (192 byte packets at 26 Mpps) packets will remain in the buffer, eligible for retransmission, for 52 μs. This time is sufficient to tolerate 28 consecutive retransmission requests in our testbed, and is well beyond the round-trip latency of modern datacenter networks. Notably, the retransmission buffer does not affect system correctness; it simply avoids triggering software failure recovery in case of UDP packet loss or reordering in the network. In the extremely unlikely case that persistent retrans-

missions repeat until a lost packet is no longer present in the retransmission buffer, the hardware triggers a conventional Raft failure recovery in software.

5 Waverunner Control Plane

In this section, we describe our software components. The software is primarily in charge of the uncommon routines of the system, such as bootstrapping and recovering from an abnormal system state. This functionality includes electing a new leader, synchronizing data between a new leader and the replicas, and controlling hardware acceleration. We begin with a vanilla Raft software implementation and add support for interaction with the accelerator hardware. For completeness, we include a discussion of the full Raft implementation, and we explicitly highlight the Waverunner-specific design decisions and additions for software-hardware interaction.

5.1 Switching to Software via Leader Election

Leader failures are handled by re-electing a new leader. Leader election can be initiated by any replica. Each replica keeps a timer in software, starting an election if a timeout is triggered due to a lack of new messages received from the leader. Each replica uses a randomized timeout value, thereby reducing the probability of competition. When the system is idle, a timed loop in the leader's software sends empty requests to the hardware to avoid unnecessary elections. This does not cause extra overhead compared to a software-only Raft implementation, as it would have a similar timed loop to send empty `AppendEntry` heartbeats.

When a follower triggers an election and requests to become leader (referred to as a candidate), its software will first disable the hardware acceleration and wait for the hardware to complete processing of the packets in the hardware accelerator pipeline and MCDMA batch queues.

The candidate software will increment its own `term` and stop responding to replication requests with lower terms, and send a `RequestVote` message to the other replicas. Even if the replicas that receive the message are using hardware acceleration, the message will pass through the hardware transparently and be directly handled by the software. Each replica will confirm that its `term` is smaller than that of the candidate, and then disable hardware acceleration and check if the candidate has a more up-to-date log by comparing the `lastLogIndex` and `lastLogTerm` of the candidate with its own. If the candidate is more up-to-date (or the same), the receiver grants the vote and sets the leader `id` to the candidate. If the candidate receives enough votes (including itself for a majority), it transitions into the leader role.

When the new leader software takes over, the system is a fully capable software Raft. It can perform any traditional system maintenance operations, such as view change. The software handles all Raft routines not implemented in hardware, such as synchronizing logs on the replicas.

5.2 Synchronizing Missing Logs

When a leader crashes or communication with it fails, the replicas may be left unsynchronized, such that some replicas may have longer logs. In more complex cases, such as election competition or consecutive leader crashes, replicas may even have different *uncommitted* logs at the same log position. Raft's (or any consensus protocol's) logic for handling these situations is complex. To ensure a simple hardware implementation, Waverunner keeps the implementation of replica log synchronization entirely in software.

After a new leader is elected, it initiates synchronization of the replica logs by sending an `AppendEntry` message containing a special `noop` operation to all replicas. If a follower has fallen behind or has non-matching logs, it will reply with an `AppendEntryReject` message, indicating a log mismatch. The leader will then send earlier log entries until the follower acknowledges accepting these logs, ensuring that the follower is synchronized with the leader. The `noop` commit entry is necessitated by the Raft protocol, as simply counting existing log entries in all replicas may fail due to a corner case in the Raft algorithm. (This is a documented idiosyncrasy of the Raft protocol (§5.4.2) [52].)

Note that there is a limit on the maximum number of entries that a replica can hold in its in-memory log. If a replica is down for an extensive amount of time, or a new replica is added to the system, that replica cannot catch up via the aforementioned approach because the leader will have discarded its older logs from memory. In this case, the leader should send a snapshot of the application dataset to the failed replica. The snapshot will contain the system state up to a particular log position which is still in the leader memory, allowing the leader to catch up the replica by sending it the entries starting from the snapshot log position. Similar to the original Raft work [52] and other recent works on speedy SMR [2, 49], the creation of the snapshot is application-specific and is orthogonal to the scope of this paper. For example, a standard approach is available in [65].

5.3 Enabling Hardware Acceleration

After the logs of all reachable replicas are synchronized, the leader will increment the `term` and send out `RequestVoteFPGA` messages to the synchronized followers. This message is identical to a normal leader election, with the additional side-effect of causing the followers to enable hardware acceleration. Once the leader receives acknowledgements from half of the followers (reaching a majority when including itself), the leader enables its own hardware acceleration. All future log entries are replicated by the hardware, without involving the CPU.

Separating hardware and software into different `term`s provides several benefits. First, it keeps the Raft algorithm intact, eliminating the need to prove correctness of our changes. Second, this approach is easy to implement, debug, and maintain, because the `term` of a log entry indicates whether it was initially replicated by software or hardware.

If a replica does not receive `RequestVoteFPGA`, but later receives an `AppendEntry` (e.g., a crashed replica rejoins), it will see a higher `term` in the message. Whenever a replica sees a higher `term` in `AppendEntry` than its `currentTerm`, or if it cannot find an entry at `prevLogIndex` in its log, it will disable hardware acceleration (if it is enabled) and reply with `AppendEntryReject`. When the leader receives an `AppendEntryReject`, it will disable hardware acceleration, trigger an election to go into a new `term`, and synchronize with the straggling replica using software. Afterwards, the system will transition back to running with hardware acceleration using the previously described steps. To summarize, Waverunner uses a unified approach to address all possible cases that deviate from the normal replication routines, including the possible loss of messages, message delays, temporary server anomalies, etc.

6 Evaluation and Results

We evaluate Waverunner with real-world applications and off-the-shelf hardware. The major questions we answer are:

- What is Waverunner's replication performance?
- Can Waverunner efficiently recover from a failure?
- How well does the Waverunner approach perform with real-world applications?
- How does Waverunner compare to other hardware-accelerated SMR approaches?

6.1 Setup

We conduct our evaluation on a 3-replica Waverunner cluster, with several additional client machines to issue requests in an open-loop manner. Each replica has two Intel E5-2695v4 CPUs, 1TB DDR4 memory, and a Xilinx U280 FPGA connected via PCIe gen 3 x16. Each FPGA has two 100 Gbps QSFP28 ports. Our replicas are connected to one switch and clients to another, and there is a 100 Gbps fiber connecting the two switches. On the FPGAs, we implemented the Waverunner hardware accelerator using Vivado HLS. For the control plane, we modified our C++ Raft implementation to coordinate with the Waverunner FPGA hardware.

In addition to Waverunner, we also evaluate the replication performance of two SMR systems for comparison:

- Mu [2]: An RDMA-based SMR implementation, which aims to provide microsecond level latency for application replication. It has a custom leader-follower consensus protocol. In the Mu implementation, all requests originate from the leader. This gives it an advantage over Waverunner, where requests are sent over the network from clients.
- DPDK-Raft: An in-house DPDK-based Raft implementation. We built our own DPDK-Raft implementation because the state of the art Raft implementation (in eRPC [34]) is equipped to perform latency tests and we are interested in both latency and throughput experiments. Our DPDK-Raft achieves similar latency as eRPC Raft.

Both Mu and DPDK-Raft use a 100 Gbps Mellanox ConnectX-4 NIC included in the replicas with the same connection specifications as the FPGA (i.e., PCIe gen 3 x16 and 100 Gbps QSFP28).

6.2 Methodology

We present two key metrics: throughput and latency. We report throughput in millions of request packets per second (Mpps) and total network bandwidth used (Gbps). For latency, we report the time in microseconds (μs) and present the median (50th percentile) and tail (90th and 99th percentile) measurements. To improve accuracy, whenever possible, we collect the results using internal hardware performance counters on the FPGAs, NICs, and switches.

We implement our client using DPDK to achieve high performance and accurate measurement. Precise control of the offered load at the client is difficult to achieve, so we set approximate targets and plot all results by using the actual measured request rates. As a result, experiments that vary the request rate may not have results with round throughput values (e.g., our plots may show 4.1 and 5.2 Mpps, rather than precisely 5 Mpps). For measuring end-to-end latency from the client and to avoid subjecting latency measurements to client-side queueing, we use a sampling approach by concurrently running two DPDK client configurations: one to apply the target load and a second lightly-loaded client (using a separate NIC) to precisely measure the latency.

6.3 Replication Performance Results

We first focus on evaluating the SMR replication performance without a specific application (where the "application" simply discards committed operations) to better understand the capability of Waverunner in a clean environment. The clients send requests to the replica cluster using small random packets (50 bytes for Waverunner and DPDK-Raft, 64 bytes for Mu due to its implementation restriction). We sweep the request rate in steps of approximately 1 Mpps and measure the replication throughput and latency at each step. We described Waverunner's MCDMA batching with timeout mechanism in Section 4.3. An adaptive batching strategy can minimize latency under all load scenarios, however, we found that for our evaluation, a constant batch size was sufficient to limit PCIe transfer overheads.

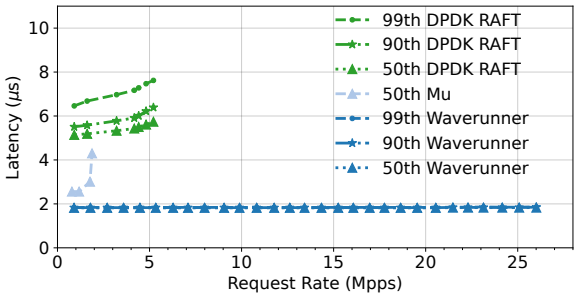


Figure 5: Performance of Packet Replication

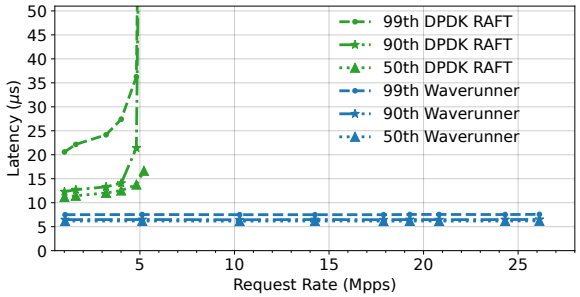


Figure 6: End-to-end Latency

Throughput. Figure 5 shows that the maximum request rate Waverunner can achieve is 26 Mpps, which is bounded by the leader's network receive bandwidth. Under this request rate, the bandwidth utilization of the leader FPGA approaches 85.5 Gbps, the maximum theoretical bandwidth achievable by the network transceivers and switch.[2]

Beyond this throughput, the FPGA transmit FIFOs experience back pressure from the MAC, which cascades to the Waverunner protocol handler and causes packet loss of incoming client requests and follower acknowledgements. This result indicates that Waverunner can fully utilize the available network bandwidth and achieve the maximum request rate.

On the contrary, Mu and DPDK-Raft cannot saturate the network bandwidth with one request per packet, resulting in only ∼2 Mpps and ∼5 Mpps peak request rate, respectively. Mu has to rely on client-side batching (aggregating multiple requests into one packet) to increase the request rate and utilize the available network bandwidth. However, doing so also drastically increases the latency. For DPDK-Raft, the throughput is bottlenecked by descriptor ring handling via PCIe. At peak throughput, DPDK-Raft starts to drop packets because the CPU cannot process and release RX descriptors at the same rate as the incoming packet stream, a situation we overcome in Waverunner by transferring multiple requests using each descriptor (§ 4.3).

Latency. Figure 5 also shows the replication latency for Waverunner, Mu, and DPDK-Raft at various request rates. Replication latency is measured from when the leader receives a client request, until the leader receives the corresponding acknowledgements from half of the followers.

The Waverunner replication latency is effectively constant at 1.8 μs, only marginally higher than the RTT of a minimum-sized packet in our network (1.68 μs). There are two characteristics of Waverunner's latency that are notable: the median, 90th-, and 99th-percentile latencies are all nearly identical, and as the request rate increases, the latency does not increase, all the way until network bandwidth is exhausted. These latency characteristics are a unique advantage of an FPGA implementation [30], as most of the components in the FPGA hardware have low and constant latency that is immune to queuing effects, allowing the replication latency to remain stable. In contrast, both Mu and DPDK-Raft exhibit substantially higher 90th- and 99th-percentile latency compared to the median latency, and the latency grows as the request rate increases and the CPUs become busier, amplifying interference and system queuing effects. As a result, the replication latency of Waverunner is significantly lower than Mu and DPDK-Raft. The worst Waverunner 99th-percentile tail latency is approximately 1/3 (36%) of the best median latency of DPDK-Raft (5 μs) and 40–80% of Mu (2.5–4.3 μs).

For completeness, we also measure the end-to-end latency on the client for Waverunner and DPDK-Raft, as shown in Figure 6. The end-to-end latency on the client includes the replication latency, the RTT between the client and the leader (with 1 switch placed between the two), and the time for the client to process the packets. For Waverunner, this adds another 4–6 μs for the median and tail latencies. For DPDK-Raft, the additional time is much larger because DPDK-Raft relies on batching and buffering to achieve maximum throughput, which add extra cost to overall latency.

Performance with Different Packet Sizes. In addition to minimum sized packets, we investigate the effect of larger requests on Waverunner, shown in Figure 7. We maintained a constant throughput of 1 Mpps and varied the payload size accordingly. For minimum-sized packets, the replication latency is 1.79 μs. As the payload grows, the latency increases slowly to a maximum of 2.13 μs. Importantly, the 99th-percentile latency remains approximately the same as the median.

CPU Utilization. Compared to the RDMA and DPDK approaches, Waverunner has an important advantage, especially at high request rates: it places far less pressure on the host CPU cores. For example, to achieve peak performance in our tests, DPDK-Raft saturates 18 CPU cores. In contrast, Waverunner consumes negligible host CPU resources because it only needs to manage the MCDMA descriptors for the operation log. This leaves CPU resources almost entirely free, allowing them to be used by the target application.

[2] Each Ethernet frame includes a 7-byte preamble, a 1-byte start of line delimiter, and a 12-byte inter-packet gap, which together account for the approximately 14.5 Gbps gap to the advertised 100 Gbps line rate.

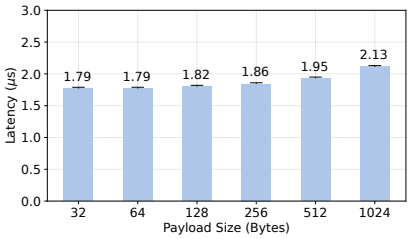


Figure 7: Latency across payload sizes. Bars show 99th percentile.

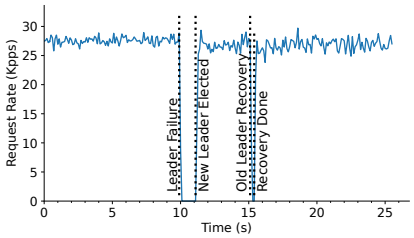


Figure 8: Request rate during failure recovery.

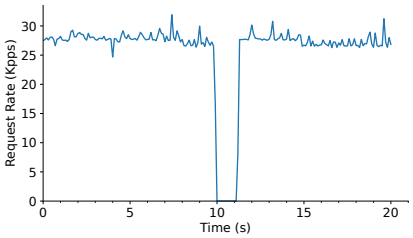


Figure 9: Request rate during view change.

6.4 Fault Tolerance and View Change

We evaluate Waverunner fault tolerance by injecting a leader failure on a healthy cluster. As expected, the leader failure triggers a leader election. At a later point, we resume the old leader, allowing it to rejoin the SMR cluster. Figure 8 presents the behavior by showing the request rate measured at the client as these failure-related events take place.

In this experiment, the system first runs normally for ten seconds, then we halt the leader to simulate a failure. At this point, requests from the clients fail to proceed because the system has no available working leader, and the throughput of the system drops to 0.

We configured the followers with a one second timeout. After timeout, a new leader election begins. This can be seen on the graph slightly more than one second after the failure, where the system resumes processing requests after the clients discover the new leader and resume sending requests.

At 15 seconds, we resume the old leader; this is recognized by the new leader when the old leader rejects the replication requests that it receives from the new leader. Because the old leader is missing log entries from its down time, the new leader starts recovery by catching up the old leader's replica. After approximately 200 ms, the log recovery completes and the system re-enables hardware acceleration, showing that the hybrid architecture of Waverunner can correctly and efficiently recover from failure.

Similar to the failover test, we performed a view change test; results are shown in Figure 9. Initially, the system runs with three replicas. After ten seconds, we send a view change command to the leader to reconfigure the system down to two replicas (removing one follower). The leader disables hardware acceleration, initiates a leader election to advance to a new term, completes the view change, and then re-enables hardware acceleration.

6.5 Real-world Applications

To understand how Waverunner performs with real-world applications, we evaluate three key-value stores: an in-memory hash table, Memcached, and Redis. We modified the applications to receive requests from Waverunner instead of the conventional network sockets. This enables low latency and high throughput operation as Waverunner bypasses the kernel to send and receive packets. Scalability across cores is achieved through sharding; the number of replication groups is the same as the number of threads. Unless otherwise indicated, we use 8-byte keys and values, which makes the packets (including network header) 135 bytes for the hash table, 150 bytes for Memcached, and 156 bytes for Redis. We use open-loop clients that perform operations on uniformly distributed keys. Although the applications were not originally designed with SMR in mind, using them with Waverunner transforms them into consistent high-availability systems.

Throughput. To evaluate applications throughput, our client can send a mixture of PUT and GET requests. Both PUT and GET requests are replicated in Waverunner, but are processed differently. For PUT requests, Waverunner responds to the client when the request is committed in Raft (acknowledged by the majority of replicas), allowing the application to handle the request log in the background, eventually updating the key-value store. For GET requests, Waverunner does not generate a client response, instead relying on the application to execute the operation from the log by retrieving the relevant data and sending them to the client. Figure 10 shows the peak sustained throughput we observed. The peak throughput of the original Memcached and Redis implementations is 1.5 Mpps, while the Waverunner implementations reach 20.7 Mpps and 19.9 Mpps, respectively. Redis has a lower throughput because it dynamically increases the size of its hash table and needs to rehash every entry. However, Memcached has a constant-size hash table that is initialized at the start.

We also examined the effect of different GET/PUT ratios on Memcached, shown in Figure 11. We observed that Memcached needs more CPU cycles for PUT requests than GET requests because, in addition to fetching the query in the key-value store (like a GET does), it also locks the region containing the key to update. As a result, higher GET ratios observe higher throughput. For Redis, we observed that changing the GET/PUT ratio does not affect the throughput.

Latency. We measured the end-to-end latency of GET and PUT operations in a 50% GET/PUT test for Memcached and Redis, as shown in Figure 12. To show the effect of different

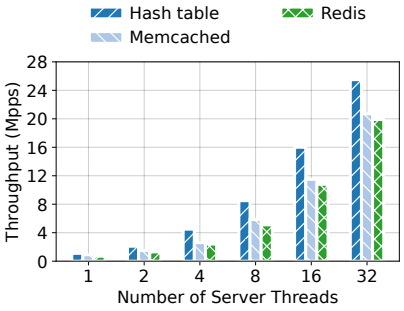


Figure 10: Peak application throughput.

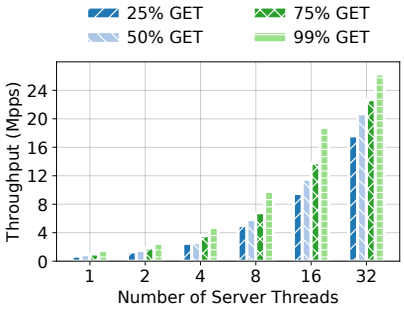


Figure 11: Memcached throughput across request ratios.

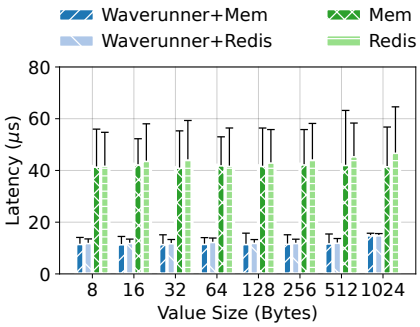


Figure 12: End-to-end latency. The bars indicate 99th percentile latency.

packet sizes, we varied the size of the value from 8 to 1024 bytes. Without Waverunner, Memcached and Redis exhibit a stable end-to-end latency of 41–44 μs, which is much lower than reported in the prior work [2, 17]. We attribute the lower latency to our high-end Mellanox NICs, both in client and server. When run on commodity NICs, the results (not shown) are much higher and closer to the previously published work.

Waverunner has a lower latency of 11.69 μs and 12.07 μs for Memcached and Redis respectively in most cases. With Waverunner, the leader FPGA generates responses for PUT requests without any CPU involvement, resulting in significantly lower end-to-end latency. For incoming GET requests, Waverunner delivers them to the application and transmits the responses to the network without involving the kernel. In summary, applications using our Waverunner framework can achieve performance comparable to kernel bypassing techniques (e.g., DPDK) for processing GET requests, and better performance for PUT requests.

6.6 Comparison to Prior Work

In this section, we discuss a comparison of Waverunner with Consensus in a Box [30] (referred to as ZABFpga below), a recent implementation of the ZooKeeper SMR protocol on an FPGA. We did not find the exact code release corresponding to ZABFpga online, which complicated our ability to study its operation in our environment. Although we did locate a project that appears to include ZABFpga's code [1], we found it challenging to port to our platform and extract from it just the ZABFpga components. A ground-up re-implementation of ZABFpga would constitute a major development effort. Such difficulties highlight the challenges in the development, portability, and maintenance of FPGA-based systems, stressing the benefits of the Waverunner approach in leaving the majority of the SMR protocol in software and implementing the hardware components using relatively portable HLS.

Based on what we can infer from the description of ZABFpga, the system has excellent performance, and would likely exhibit throughput and latency on par with Waverunner if ported to our environment, which has 100 Gbps NICs compared to 10 Gbps in the original paper. However, the ZABFpga system clearly required a drastically more complex development effort and would incur massively higher maintenance and troubleshooting cost. This is because ZABFpga implemented the ZooKeeper protocol completely, including the leader election and failure recovery, in custom FPGA hardware. This approach also required implementing the application (a key-value store) on the FPGA, including the ability to store the replication log in DRAM connected to the FPGA. Based on the description in the paper, the replication latency is 3 μs while Waverunner has 1.8 μs. It would be unfair to compare two designs on the throughput as the ZABFpga uses 10Gbps NICs to communicate with other nodes.

7 Related Work

Hardware Accelerated Networking. Early works on hardware acceleration in NICs offered a range of features, from simple ones such as checksum calculation and receive-side-scaling (RSS), to complex ones such as RDMA and TCP offloading engines [48]. Although earlier network hardware accelerators hard-wired the acceleration functionality, the trend has shifted toward programmability, with modern *SmartNIC* devices comprising programmable CPU cores [47] or programmable FPGA fabrics [50]. Modern advanced accelerators include functionality such as in-line handling of protocol encapsulation, VLAN processing, and encryption and decryption of data streams [18, 43]. Waverunner is a SmartNIC that accelerates replication routines of the Raft protocol. Like other SmartNICs, we utilized a bump-in-the-wire architecture to accelerate the replication routines in the FPGA. AccelNet [20] accelerates network services for virtual machines on SmartNICs in data centers. However, the acceleration is only loosely coupled with the application, such that whenever AccelNet does not have a rule for a packet, it consults the application to install the missing rule. On the other hand, Waverunner is more specialized, as it identifies Raft packets

and does not need software support to handle other protocols. P4 [4] is a high-level language for network functions with implementations on FPGAs [3, 7, 9, 27, 45, 66]. hXDP [5], an FPGA-based NIC, uses soft cores to execute eBPF, another high-level language to describe network functions. These systems target system-wide packet processing, whereas Waverunner is specialized and optimized for processing only the Raft replication routines.

Caribou [29] implements a hardware accelerator for high-performance databases computations, including the full functionality of a fault-tolerant key value store inside an FPGA. KV-Direct [41], CliqueMap [61], Xenic [59], and RedN [55] extend RDMA primitives to enable remote key value operations to main memory. Waverunner takes a similar approach which puts requests in the follower's memory through PCIe. This is unlike Caribou, which does not use the host and relies on the FPGA to execute the database application. Floem [53], NICA [19], iPipe [44], FlexTOE [60], and FairNIC [23] provide a framework that can offload network applications such as Memcached on programmable SmartNICs. Although its goals of low latency and high throughput are similar to Floem and NICA, Waverunner targets the Raft distributed protocol, using hardware acceleration for the communication among multiple nodes rather than for the application logic.

State Machine Replication. State machine replication achieves fault tolerant, highly available services by leveraging consensus protocols [26, 40, 52]. From among the popular consensus protocols, Waverunner implements the Raft protocol [52], offloading the replication routines to a hardware accelerator. Several prior studies proposed ways to increase the performance of SMRs. eRPC [34], FaSST [35], and Breakwater [12] use an RPC library on top of the NIC API and RDMA, respectively, to provide low latency communication for applications. PigPaxos [11] relays messages by subgouping followers. These works optimize the network IO bottleneck, increasing the performance considerably, but they still suffer from the CPU bottleneck for implementing all parts of the protocol. Increasing parallelism of SMR [14, 24, 36, 56, 62] can further improve the performance, which Waverunner can benefit from for the application design. HovercRaft [39] moves SMR from the application layer to the transport layer and optimizes Raft to avoid the CPU and network IO bottleneck. Similarly, Waverunner addresses the same bottlenecks by offloading the network communication and replication to the hardware accelerator. Some SMR systems leverage high performance programmable switches [15, 32, 33, 42]. Rather than changing the network infrastructure, Waverunner employs a hardware accelerator in the NIC of each replica to accelerate the replication communication and operations.

There are several studies on low latency SMR through RDMA [17, 31, 37, 38, 54, 65], some of which are based on variants of Paxos. Although these works offer low latency, they are still bounded by the CPU bottleneck, as all of them cannot send packet at line rate with minimum size packets, and have high replication latency. Mu [2] introduces a microsecond latency SMR in which the leader writes requests in the log of each replica in only one round of RDMA transfers, without involvement from the CPUs on the follower nodes. Comparably, Waverunner achieves constant microsecond latency using FPGAs without changing core routines of the Raft protocol, while achieving high throughput on minimum size packets. ZABFpga [30] accelerates the Zookeeper consensus protocol using an FPGA and shows the benefits of hardware accelerator for SMRs in terms of latency and throughput. Waverunner achieves similar performance, but presents a design for the replication routines of Raft protocol while leaving all complex functionality of the Raft protocol (such as leader election and failure recovery) and the application (a key-value store) in traditional software.

8 Conclusions

We presented Waverunner, a hardware-software hybrid approach for implementing state machine replication. Our approach relies on the observation that, despite the complexity of SMR, the most frequently used routines can be easily implemented in hardware, while leaving the complex protocol and application logic in traditional software. Using this approach, we attain the best characteristics of the prior work, achieving the performance of full-hardware implementations while retaining the flexibility of software implementations with hardware-assist mechanisms such as DPDK and RDMA.

Waverunner is a practical realization of our approach. It is elegant and simple, leveraging a complete software implementation of the Raft protocol at its core and demonstrating how the most-frequently used functionality can be offloaded to hardware using only 220 lines of C++ HLS code. Waverunner achieves network line-rate throughput, nearly constant mean and tail (99th percentile) replication latency regardless of throughput, and leaves the majority of the CPU processing power available for the target application.

Acknowledgements

We thank our shepherd Maria Apostolaki and the anonymous reviewers of NSDI '23, OSDI '22, SOSP '21. We thank Yida Wu for implementing the first version of DPDK Raft; we also thank Satya Jain and Sergey Madaminov for their contribution to this work in the early stage. This research was supported in part by NSF CCF 2007362, CNS 1763680, CNS 2130590, and CNS 2214980.

References

[1] FPGA ZooKeeper source code. https://github.com/fpgasystems/caribou.

[2] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[3] P. Benáček, V. Puš, J. Kořenek, and M. Kekely. Line rate programmable packet processing in 100gb networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(3), July 2014.

[5] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[6] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[7] Jakub Cabal, Pavel Benáček, Lukáš Kekely, Michal Kekely, Viktor Puš, and Jan Kořenek. Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 249–258, 2018.

[8] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2021.

[9] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen, and C. Zhang. P4 to FPGA-a fast approach for generating efficient network processors. *IEEE Access*, 8, 2020.

[10] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, August 2007.

[11] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. PigPaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, June 2021.

[12] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[14] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[15] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking (ToN)*, 28(4), August 2020.

[16] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review (CCR)*, 46(2), April 2016.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2014.

[18] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. Flexdriver: A network driver for your accelerator. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2022.

[19] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. Nica: An infrastructure for inline acceleration of network applications. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, July 2019.

[20] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2018.

[21] Linux Foundation. Data plane development kit (DPDK).

[22] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2021.

[23] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, July 2020.

[24] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, April 2014.

[25] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP Database. *The Proceedings of the VLDB Endowment (PVLDB)*, 13(12), 2020.

[26] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, June 2010.

[27] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The P4->NetFPGA workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

[28] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021.

[29] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, August 2017.

[30] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: inexpensive coordination in hardware. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, March 2016.

[31] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2), April 2019.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2018.

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[34] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, February 2019.

[35] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2016.

[36] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[37] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2020.

[38] Mikhail Kazhamiaka, Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong, and Khuzaima Daudjee. Sift: resource-efficient consensus with RDMA. In *The International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2019.

[39] Marios Kogias and Edouard Bugnion. Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, April 2020.

[40] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), 2001.

[41] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[42] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2016.

[43] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[44] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2019.

[45] Thomas Luinaud, Jeferson Santiago da Silva, J.M. Pierre Langlois, and Yvon Savaria. Design principles for packet deparsers on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[46] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2019.

[47] Marvell. OCTEON TX2 LiquidIO III SmartNIC. https://www.marvell.com/products/data-processing-units.html.

[48] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2003.

[49] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[50] NVIDIA. NVIDIA Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/.

[51] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, June 1988.

[52] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, June 2014.

[53] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018.

[54] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on rdma networks. In *Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, June 2015.

[55] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is turing complete, we just did not know it yet! In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2022.

[56] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In *The International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, November 2017.

[57] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, June 2012.

[58] Timothy Roscoe. Keynote: It's time for operating systems to rediscover hardware. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021.

[59] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-accelerated distributed transactions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2021.

[60] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2022.

[61] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo MK Martin, Amanda Strominger, Thomas F Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2021.

[62] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, April 2020.

[63] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The resilient geo-distributed sql database. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, June 2020.

[64] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4), 2014.

[65] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable Paxos on RDMA. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, September 2017.

[66] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A rapid prototyping framework for p4. In *ACM Symposium on SDN Research (SOSR)*, 2017.

[67] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. On the parallels between paxos and raft, and how to port optimizations. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, July 2019.

[68] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2021.

[69] Siyuan Zhou and Shuai Mu. Fault-tolerant replication with pull-based consensus in MongoDB. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2021.

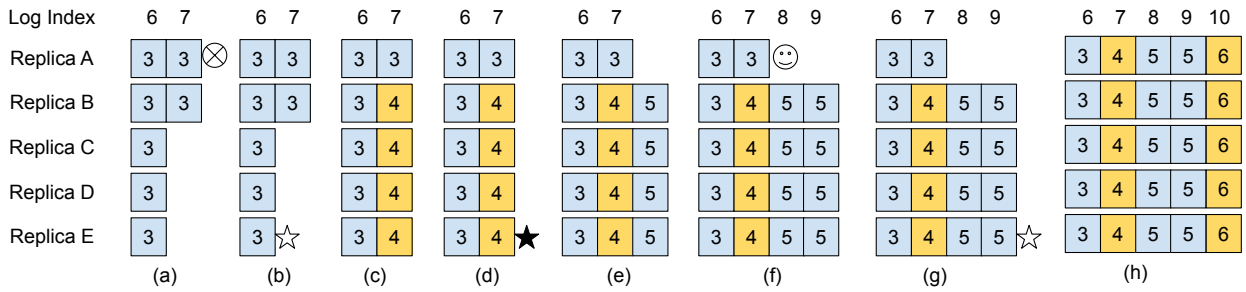


Figure 13: Waverunner Operation. ☆ represents an election that disables hardware acceleration; ★ represents an election that enables the hardware acceleration; ⊗ represents a fail stop and ☺ a recovery. The numbers inside the boxes refer to the `term` numbers in each log entry. The blue boxes refer to regular log entries; the yellow boxes refer to empty `noop` log entries.

A Correctness

Here we discuss the correctness of our approach. One reason we choose to implement Raft instead of inventing a consensus protocol is that Raft is widely used and proved correct.[3] Using Raft can help us avoid having any errors in inventing a new protocol, which is known to be an error-prone process.

We show that with or without the hardware acceleration, including the transition, the system follows Raft protocol.

Fact 1. When the hardware acceleration is off at a replica, the replica follows the Raft protocol.

Therefore, if hardware acceleration at all replicas is off, the system design is a standard Raft and it is correct.

Lemma 2. When the hardware acceleration is on (and during the process it is switched on) at a replica, the replica follows the Raft protocol.

This is the principle throughout the system design. The hardware part is designed to switch back to software whenever it sees a message that it is not expecting. Not responding to that particular message is not a behavior that violates Raft's safety because Raft's original assumption is that the network is asynchronous and messages could be lost. Therefore, the replica as a whole (both hardware and software) is still following the Raft protocol, except that it requests an election. In Raft (and other consensus protocols), doing an election is always safe.

As a replica follows the Raft protocol regardless of whether the hardware acceleration is on or off (or during transition), the system is a Raft and thus correct.

[3] By "correct" we mean the system has both safety and liveness. Because Raft has already proved on these, we will use "correct" to refer to our system is either a standard Raft or is equivalent to it.

B An Example of Waverunner Operations

Figure 13 walks through an example of Waverunner failure recovery with five replicas A,B,C,D, and E. The numbers in the boxes refer to the `term` numbers in each log entry.

(a) Replica A is the leader, using hardware acceleration to replicate log entries to followers until it stops.
(b) Replica E is first to detect a lack of new messages from leader A. E disables hardware acceleration and triggers a leader election, which it wins (becoming the new leader) after receiving votes from C and D.
(c) Replica E commits a `noop`, indicated in yellow, to all replicas except A (which remains unavailable). Note that a log entry in replica B is overwritten because it was ahead of the new leader. This operation is safe because the log entry was not committed.
(d) Replica E starts a round of `RequestVote2FPGA`, enabling hardware acceleration on all replicas.
(e) Replica E operates as the leader, replicating log entries using the hardware accelerator.
(f) Replica A recovers, immediately observing new `AppendEntry` messages arriving from leader E. Replica A reports a mismatch with its existing logs by rejecting the new entries.
(g) Having learned of the mismatch on replica A from the rejection message, replica E disables hardware acceleration with another leader election.
(h) Replica E then commits another `noop` and sends the missing log entries to replica A. In this process, the mismatched logs on replica A are also overwritten.

```
Variables shared by hardware and software:
 Used only by leader:
  matchIndex[] for each follower, index of highest
               log entry known to be replicated,
               initialized to 0, increases
               monotonically
  commitIndex  index of highest log entry known to
               be committed, initialized to 0
 Used by all replicas:
  id           a globally unique integer that
               identifes the server
  isLeader     hint that suggests whether the
               server is leader
  currentTerm  latest term server has seen,
               initialized to 0
  lastLogIndex index of the last log entry, is a
               sequentially increasing counter,
               initialized to 0
  lastLogTerm  term of the last log entry

Variables in host software:
 Used only by leader:
  nextIndex[]  for each server, index of the next
               log entry to send to that server,
               initialized to leader's
               lastLogIndex+1
 Used by all replicas:
  votedFor     candidateId that received vote in
               current term (or null if none)
  log[]        log entries; each entry contains
               command for state machine, and term
               when entry was received by leader
               (first index is 1)
  lastApplied  index of highest log entry applied
               to state machine (initialized to 0,
               increases monotonically)
```

Figure 14: Variables in Hardware and Software.

C Hardware and Software Variables

Figure 14 presents the complete set of Raft protocol variables that are used by hardware and software.

LeakyScatter: A Frequency-Agile Directional Backscatter Network Above 100 GHz

Atsutse Kludze and Yasaman Ghasempour
Princeton University

Abstract

Wireless backscattering has been deemed suitable for various emerging energy-constrained applications given its low-power architectures. Although existing backscatter nodes often operate at sub-6 GHz frequency bands, moving to the sub-THz bands offers significant advantages in scaling low-power connectivity to dense user populations; as concurrent transmissions can be separated in both spectral and spatial domains given the large swath of available bandwidth and laser-shaped beam directionality in this frequency regime. However, the power consumption and complexity of wireless devices increase significantly with frequency. In this paper, we present *LeakyScatter*, the first backscatter system that enables directional, low-power, and frequency-agile wireless links *above 100 GHz*. LeakyScatter departs from conventional backscatter designs and introduces a novel architecture that relies on aperture reciprocity in leaky-wave devices. We have fabricated LeakyScatter and evaluated its performance through extensive simulations and over-the-air experiments. Our results demonstrate a scalable wireless link above 100 GHz that is retrodirective and operates at a large bandwidth (tens of GHz) and ultra-low-power (zero power consumed for directional steering and $\leq$ 1 mW for data modulation).

1 Introduction

Low power communication has become increasingly important in emerging applications such as home automation, smart healthcare, high-quality video streaming, and localization [22, 44, 57, 58]. The number of low-power wireless devices is expected to grow to 41 billion by 2025 [36]. While the last few years have seen rapid innovations in the design and implementation of low-power wireless communication [17, 25, 29, 33, 40, 62], existing networks are limited in the number of nodes that they can concurrently support [10, 31].

The use of frequencies above 100 GHz (henceforth referred to as the terahertz (THz) band) provides unique opportunities for concurrent transmissions. First, the availability of a large swath of spectrum above 100 GHz will facilitate dense user populations to operate concurrently at orthogonal frequency channels. Second, narrow-beam directional transmission and reception, which is required to combat the high path loss in this regime, provides additional opportunities for simultaneous transmission through space division multiple access. The sub-THz frequencies offer the best of the RF and Optical spectrum: like RF, they can be phase modulated and experience lower penetration and reflection losses when compared to optical while still providing a large swath of continuous bandwidth and laser-shaped beams. Indeed, these properties have made THz frequencies promising for 6G wireless technology. Despite these advantages, operation at such a high frequency is fundamentally power demanding since the power consumption of the RF circuitry is proportional to the frequency. This high power consumption has even stalled the deployment of mobile 28 GHz nodes and will worsen at the THz regime [8, 46–48, 52]. Furthermore, creating directional beams requires large antenna arrays, drastically increasing the power consumption and complexity of the device. This challenge worsens under mobility when constant beam adaption is needed to maintain the link. For these challenges, most existing low-power solutions have been limited to sub-6 GHz bands [35, 45, 54], with a few recent narrow-band demonstrations at 24 GHz band [30, 37, 53].

In this paper, we present *Terahertz Leaky-Waveguide Backscatter (LeakyScatter), a novel architecture for frequency-agile directional backscattering above 100 GHz*. Instead of generating and emitting THz signals, LeakyScatter piggybacks its data on the impinging THz signals that are emitted from a THz transceiver. To enable directional connectivity, we exploit the unique properties of leaky-wave antennas. Leaky-wave antennas are traveling wave structures that can be realized with metal waveguides or on CMOS. In its simplest form, a leaky-wave antenna is a parallel-plate metal waveguide having open slit(s) on one side, with the interesting characteristic that guided waves (inside the waveguide) can *leak out* into free space such that the emission

angle is correlated with the frequency of the signal [21]. These frequency-dependent radiations have been recently exploited for THz path discovery and localization [12, 13, 28] and have also been demonstrated in CMOS technology [50]. In this paper, we leverage angle-frequency coupling in leaky waveguides to enable the *first* passive ultra-wideband and retrodirective structure above 100 GHz.

Our key insight is that leaky waveguides are reciprocal devices, i.e., its reception characteristics are identical to its transmission's. When it acts as a receiver, the impinging signals couple into the waveguide only if their spectral content is in agreement with the incident angle (i.e., where the angle of emission and reception from the waveguide is frequency-dependent). Hence, we devise two symmetrical slits in LeakyScatter: one for accepting free-space ambient signals into the waveguide, and the other for leaking those signals back into the air. Specifically, a far-field active transceiver emits a THz signal toward the LeakyScatter. The impinging signal would then couple into the waveguide where it is guided toward the second slit and radiates out back to free-space forming a directional beam that points to the transceiver's location, thereby enabling retrodirectivity. We emphasize that our retrodirective structure is truly wideband and spectrally agile as LeakyScatter can operate between 100 GHz and 500+ GHz. Further, the directionality is achieved with zero power consumption. The active transceiver, however, is expected to be power demanding as it is capable of generating tunable wideband signals and steering them to any desired direction. Note that the discrepancy in RF capabilities between active transceivers and low-power nodes is often the case in backscatter wireless networks.[1]

In LeakyScatter, we enable data transmission by modulating the amplitude of the THz backscattered signal.[2] Our key observation is that the size of the slit directly impacts the amount of coupling efficiency (or backscatter power). Yet, the physical dimension of the slits in leaky antennas are fixed upon production; instead, we control the *effective size* of the aperture by re-configuring the trajectory of guided waves inside LeakyScatter's waveguide. In particular, we employ an mm-sized electrostatic MEMS mirror inside the waveguide's cavity to dynamically and electronically guide EM signals toward the slit or steer them away from it, as shown in Fig. 1. We characterize the backscattered power as a function of the micro-mirror's orientation using ray optics principles. We then optimize the architecture of LeakyScatter for maximum link budget and modulation order.

We fabricate a custom LeakyScatter and deploy it together with our in-house THz transceiver. We present the *first experimental exploration of wireless backscattering above 100 GHz*. We evaluate the performance of LeakyScatter with extensive COMSOL simulations and experiments in various settings. We experimentally validate the reciprocity in leaky-wave devices, and characterize the retrodirective beams that radiate from our LeakyScatter nodes. Finally, we demonstrate low-power amplitude modulation and evaluate the feasibility of concurrent THz backscatter links in dense LeakyScatter networks.

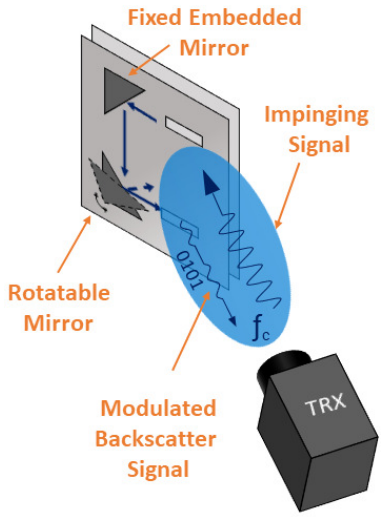


Figure 1: Illustrating the retrodirective backscatter link between our LeakyScatter and a broadband transceiver.

2 Background and Related Work

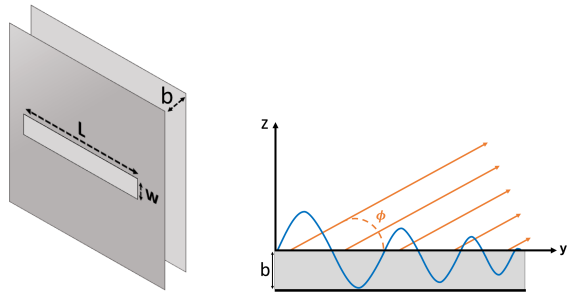
2.1 Wireless Backscatter Communication

Backscatter technology is introduced for energy-efficient communication between power-constrained wireless devices [3, 20]. The underlying idea is to allow low-power nodes to piggyback their data on an ambient signal instead of generating their own RF signal, which would demand power-hungry components such as mixers, oscillators, and amplifiers [34]. In particular, an access point (AP) transmits an RF signal to the node which will then be modulated and reflected back to the AP for processing. For example, a simple On-Off Keying (OOK) can be implemented where reflecting the AP's signal translates to sending a '1' bit and absorbing the signal represents the '0' bit.

Recently, there has been significant work on extending the communication range [9, 23, 41, 56] and improving the data rate of backscatter communication links [26, 30]. In particular, employing low-power coding techniques such as chirp spread spectrum has shown promise for decoding backscattering signals below the noise floor [6]. However, while these techniques can achieve long-range communications, they often do not scale well with the number of devices, i.e.,

[1] In this work, we only focus on the backscatter architecture. Designing efficient high-frequency transceiver is an active field of research and is beyond the scope of this paper.

[2] We emphasize that noncoherent on–off keying is identified as one of the two modes in the first standardization of sub-THz bands by the IEEE 802.15.3d task group [42].



(a) (b)

Figure 2: Parallel plate leaky waveguide: (a) front view (b) side view.

they are often limited to very few (1-2) concurrent links. The state-of the-art protocol (NetScatter [18]) allows for concurrent transmission of 256 backscatter devices over a bandwidth of 500 KHz. NetScatter relies on the ability of the low-power nodes to generate cyclic shifted chirps, which is relatively power-demanding. Further, the number of concurrent users is inherently bounded to the bandwidth and the spectral resolution at endpoint devices.

Instead, in this paper, we take a fundamentally new approach to backscattering and introduce a spectrally-agile and retrodirective node architecture that scales the number of concurrent users through the division of signals in frequency and space. In this context, moving to higher frequencies is advantageous for two reasons: *(i)* the large availability of bandwidth allows for serving more concurrent users on non-overlapping channels; and *(ii)* directional communication (required for mmWave and THz links) provides opportunities for concurrent transmission of nodes that are sufficiently separated in space.

Therefore, unlike traditional backscatter nodes that operate below 6 GHz, our backscatter architecture aims at establishing backscatter links above 100 GHz. A few recent works have demonstrated retrodirective backscattering at mmWave bands (up to 28 GHz) using the Van Atta technique [19, 30, 37, 53]. Unfortunately, Van Atta Arrays are inherently narrow-band as the transmission lines are optimized at a particular wavelength, which is fixed (non-configurable) after fabrication.

2.2 Leaky-Wave Antennas

Leaky Wave Antennas (LWAs) belong to the general class of traveling wave antennas that can be implemented with circuits [50] or simply with an air-filled parallel-plate waveguides with an open slit on one of the plates [2]. A traveling wave inside the waveguide may *"leak"* into free-space through the open slit, as depicted in Fig. 2. Notably, Maxwell's equations and the boundary condition suggest that the direction of emission from the slit is correlated with the frequency of the guided wave. By assuming infinitely thin metal plates that are infinitely conductive, the phase matching conditions for the TE_1 mode yields [24]:

$$\phi(f) = asin\left(\frac{c}{2bf}\right), \tag{1}$$

where ϕ is the emission angle (relative to the waveguide plate), f is the frequency of the input signal, b is the plate separation, and c is the free-space speed of light. Eq. (1) indicates that signals with higher frequencies emit out at lower azimuth angles and vice versa.

While Eq. (1) captures the optimal emission angle as a function of frequency, in practice, the signals leaking from the waveguide can appear at a range of angles, albeit encountering different coupling losses. To understand this broader angular width, we can treat the leaky waveguide slot as a finite-length aperture, which produces a diffraction pattern in the far-field. According to Huygen's principle [15, 55], for a diffracting aperture (i.e., slot length) of L and the dominant TE_1 mode, the far-field radiation pattern can be written as

$$\begin{aligned} G(\phi, f) &\propto \int_{-L/2}^{L/2} e^{-j\beta_y y} e^{-\alpha y} e^{jk_0 cos(\phi) y} dy \\ &= sinc\left((\beta_y - j\alpha - k_0 cos\phi)\frac{L}{2}\right), \end{aligned} \tag{2}$$

where $sinc(x) = sin(x)/x$, $k_0 = \omega/c$ is the free-space wave-number, α is the leakage attenuation, and β_y is the propagation constant. β_y can be written as $\beta_y = \sqrt{k_0^2 - (\pi/b)^2}$ when the parallel-plate waveguide has an air core. Eq. (2) also confirms the dependency of emission pattern on frequency where the peak output radiation occurs at the angle at which $Re\{(\beta_z - j\alpha - k_0 cos\phi)\frac{L}{2}\} = 0$, yielding Eq. (1). It should be noted that the coupling efficiency and reception behavior of a leaky-wave antenna are independent of polarization [43].

Recently, leaky-wave antennas have been used in multiple sensing tasks including link discovery [12, 14], mobility tracking [13], 3D localization [28], physical-layer security [61], and even for wireless authentication [27]. However, this paper is the first work toward exploiting the angle-frequency relation and the antenna reciprocity for frequency-agile backscattering.

3 Design

In this section, we describe the underlying principles and key components of LeakyScatter.

3.1 Design Overview

The large amount of available bandwidth above 100 GHz together with directional transmission opens up opportunities for concurrent high-rate networks beyond 5G. Yet, operation at such high frequencies is fundamentally power-demanding since the power consumption of the RF circuitry is

proportional to the frequency. Further, creating directional beams would require large antenna arrays, which itself increases the power and complexity. Previously-established backscattering technology addresses the first challenge by allowing nodes to piggyback their data on an ambient signal instead of generating their own RF signal, thereby, eliminating the need for power hungry components. However, existing backscatter designs are either directional and narrow-band [30, 37, 53] or omnidirectional at much lower frequency bands (e.g., sub-6 GHz) [35, 45, 54].

Our proposed LeakyScatter is the first frequency-agile directional backscattering network at THz bands. Instead of conventional systems that use phased array antennas for beam steering, we introduce a fundamentally novel node architecture based on leaky-wave antennas. Our key observation is that a leaky wave antenna is inherently reciprocal. In particular, the first-principle model suggests that the frequency-dependent radiation in leaky-wave antennas is identical in both transmission mode (guided signals coupling out into free-space) and reception mode (free-space waves coupling into the waveguide). We leverage this interesting characteristic to build a fully-passive retrodirective node. We design a two-slit waveguide (one for receiving signals from the transceiver (TRX) and another for transmitting the signal back directionally toward the transceiver), as shown in Fig. 1. We also embed two mm-sized mirrors inside the waveguide to guide the propagation path between the two apertures.

Given the angle-frequency relationship, the TRX should choose the *correct* frequency to communicate with LeakyScatter nodes depending on the node location and orientation relative to TRX. Fortunately, single-shot angular localization with leaky-wave antennas has been successfully demonstrated in the literature [28]. Here, we assume that the TRX can localize all LeakyScatters by implementing such prior schemes. Further, we assume that the TRX has flexible ultra-wideband transmission and detection capabilities. Such asymmetry between the RF capabilities at the TRX and low-power nodes is typical in all backscatter networks.

In order to piggyback information bits on backscattered signals, we enable amplitude modulation by changing the trajectory of in-coupled waves. Specifically, a small rotation in one of the embedded mirrors would cause significant fluctuations in the amount of power leaked into free-space. Hence, we realize amplitude modulation by dynamically changing the voltage of a MEMS mirror according to the bit stream. We discuss other potential modulation strategies in Sec. 6. Finally, we emphasize that LeakyScatter is ultra-wideband (supporting bandwidths upto few 10s of GHz) and frequency-agile (supporting carrier frequencies from 100 GHz to 500 GHz), retrodirective, and scalable to large-scale networks. Next, we will illustrate the key components of LeakyScatter in detail.

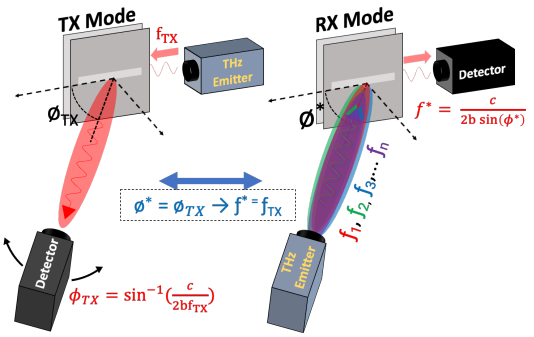


Figure 3: Illustration of reciprocity in leaky waveguide: the angle frequency coupling holds in both transmission and reception modes.

3.2 Retrodirectivity in LeakyScatter

The retrodirectivity in LeakyScatter is rooted in the antenna reciprocity of leaky-wave antennas. First, we formally model this reciprocity. Consider a tunable source, a broadband detector, and a single-slit waveguide as shown in Fig. 3. First, the tunable source excites the waveguide (TE1 mode) with a signal at frequency of f_{TX}. Given the angular-frequency coupling, the waveguide will act as a directional transmitter creating a directional beam in the far-field (labeled as TX mode in Fig. 3). The detector then measures the received power at various angles to estimate the radiation pattern of the leaked waves. Eq. (1) suggests a direct one-to-one relationship between the angle at which maximum power is received (denoted as ϕ_{TX} in Fig. 3) and f_{TX}. Next, we swap the detector and source while keeping the same waveguide. The source emits out signals at different frequencies ($f_1, f_2, ..., f_N$) while the broadband detector captures the power of coupled waves at each corresponding frequency. Antenna reciprocity suggests that given a fixed leaky-antenna structure (e.g., aperture size and plate separation), and waveguide positioning (i.e., $\phi^* = \phi_{tx}$), the maximum coupling happens when $f^* = f_{TX}$, which is equal to $\frac{c}{2bsin(\phi_{tx})}$, according to Eq. (1). We will experimentally evaluate this in Sec. 5.

LeakyScatter utilizes two symmetrical apertures, one meant for reception and another meant for transmission. Embedded within the waveguide cavity are two mirrors as illustrated in Fig. 4 to passively redirect the in-coupled signal from the receiving aperture toward the transmitting aperture. More specifically, a THz signal impinging on LeakyScatter at angle θ_{TX} interacts with the first mirror inside the cavity, which results in a 90° rotation in the propagation direction. Upon impinging on the second embedded mirror, the signal deflects in the reverse direction and moves towards the second slit. The signal then leaks out into free-space through the second slit with an emission angle of θ_{RX}. Given the reciprocity, we have $\theta_{TX} = \theta_{RX}$, as also illustrated in Fig. 4. Such retrodirectivity is essential to our backscatter networks as a single active

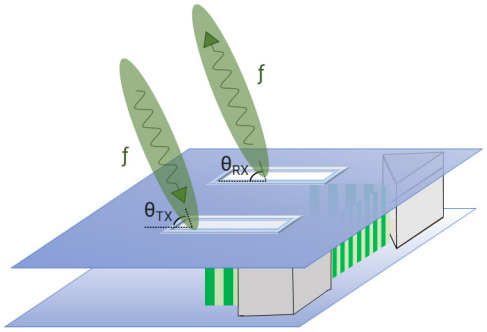


Figure 4: A schematic of LeakyScatter, showing two open slits and embedded mm-sized mirrors to guide the THz waves in between.

transceiver (co-located emitter and detector) is now able to establish a directional link with a truly passive architecture.

We can model the amount of backscattered power by incorporating internal waveguide losses as well as the far-field pattern of LeakyScatter in both TX and RX modes. Specifically, when a transceiver sends an unmodulated signal towards the waveguide, the total power that couples into the waveguide is a function of frequency, f, and the impinging angle θ_{TRX}, and can be modeled by Eq. (2). Once the impinging signals couple into LeakyScatter, they will experience propagation losses within the waveguide en route to the transmitting aperture. Finally, the waves emit out with a similar far-field behavior. Therefore, the total backscattered power (P_{bsc}) at frequency f can be modeled as:

$$P_{bsc}(f,\phi_{TRX}) \propto G^2(f,\phi_{TRX}) \times (\frac{1}{L_{WG}}), \tag{3}$$

where L_{WG} represents the incurred losses in the guided mode, ϕ_{TRX} is the relative angle between the transceiver and the LeakyScatter device. $G(f,\phi_{TRX})$ is defined in Eq. (2). We can see that the maximum backscattering power is achieved when the impinging angle and signal's frequency satisfies $Re\{(\beta_z - j\alpha - kcos\theta)\frac{L}{2}\} = 0$. Next, we explain how L_{WG} is modulated based on the data bit stream.

3.3 Data Modulation and Demodulation

So far we have explained how LeakyScatter realizes a retrodirective communication link with an active external transceiver. Now, we explain how LeakyScatter modulates the backscattered signal to transmit information bits. Our design is based on amplitude modulation such that different backscattered power levels translate to distinct sequences of bits. Fluctuation in power level is achieved by an on-demand modification to the power loss inside the waveguide. In particular, one of the aforementioned embedded mirrors is replaced with a mm-sized rotatable MEMS mirror. By changing the mirror's orientation, we seek to control the amount of power that is guided toward the second slit and hence can potentially leak out back to free-space.

Fig. 5 explains how a slight misalignment in the orientation of the MEMS mirror yields a non-negligible drop in the backscatter power. In principle, if the dispersion was minimal and the slit was infinitely thin, even an arbitrarily small rotation of the mirror (i.e., $\delta\theta_{rot}$) could cause zero backscattered power as the guided waves would just miss the open slit. However, in practice, due to dispersion and non-zero slit width, changing the propagation path of guided waves would yield a drop in power. The exact amount of power drop is a complicated function of the frequency, slit dimension, wave propagation constant, the leakage attenuation factor (denoted as β and α in Eq. (2)), and more importantly, the amount of rotation (i.e., θ_{rot}).

Our key insight is that the amount of radiated power is proportional to the leakage area and changing the trajectory of guided waves would impact the effective aperture. For simplicity, we consider a rectangular slit (width of W and length of L) and assume that signal leakage is uniform across the slit length. As shown in Fig. 5, when θ_{rot} increases and the guided waves are further directed away from the open slit, the effective aperture area seen by the guided waves should decrease. Using ray optics, we write a first-order approximation of the internal waveguide losses as:

$$L_{WG}(\theta_{rot}) = c_1 + c_2 \frac{A_{slit}}{A_{eff}(\theta_{rot})} = c_1 + c_2 |\frac{2L}{W}\, tan(2\theta_{rot})|, \tag{4}$$

where c_1 represents the constant losses such as propagation loss within LeakyScatter and c_2 is a constant aperture coupling loss. A_{slit} is the area of the slit and A_{eff} is the effective aperture at rotation angle θ_{rot}. Note that c_1 and c_2 is a deterministic function of the waveguide geometry (e.g., the internal path length between the two slits) and can be measured and known upon production. Clearly, L_{WG} is minimum at $\theta_{rot} = 0$ and increases with θ_{rot}. Combining

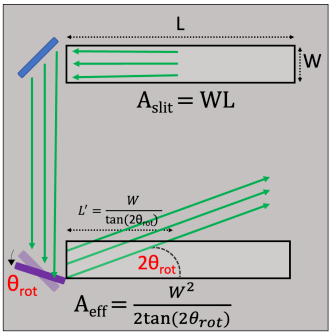


Figure 5: An electronically rotatable mm-sized mirror embedded within LeakyScatter changes the propagation path of guided waves and thus effectively controls the size of transmitting aperture seen by the guided waves.

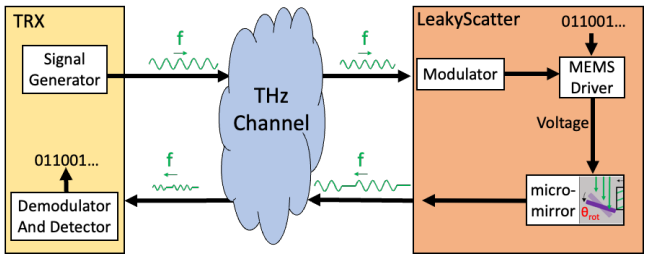


Figure 6: The block diagram of a THz backscatter network.

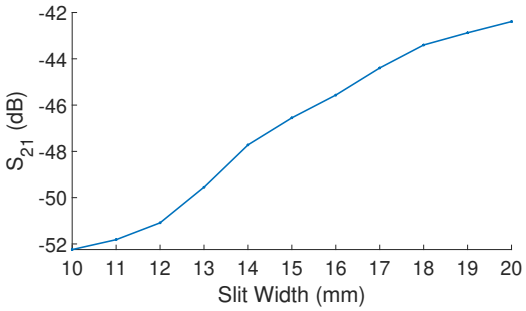


Figure 7: The impact of slit width on total backscatter gain in LeakyScatter.

Eq. (4) and Eq. (3), we can write:

$$P_{bsc}(f) \propto \frac{G^2(f,\phi_{TRX})}{c_1 + c_2|\frac{2L}{W}tan(2\theta_{rot})|} \quad (5)$$

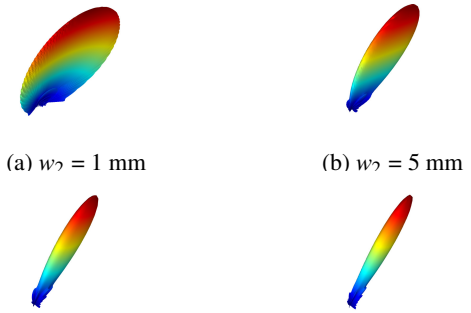
Putting all pieces together, Fig. 6 depicts the block diagram of an TRX-LeakyScatter link. As shown, the modulator at LeakyScatter sets the voltage values at the micro-mirror MEMS driver according to the bit stream. The orientation of the mirror would then change the trajectory of guided waves imposing amplitude modulations. The modulated signal is steered back toward the TRX. At the TRX, the demodulation block retrieves the data bits from analyzing the measured power spectrum.

3.4 Design Optimization for Maximizing Reflection Gain

Given the link budget scarcity in backscatter networks, we aim to maximize reflection gain (i.e., backscatter power) in LeakyScatter. Here, we introduce the underlying optimizations and trade-offs that achieve this goal.

3.4.1 Slit Geometry

The slit geometry plays a key role in the coupling efficiency and directivity gain in LeakyScatter. In principle, a wider slit is desirable as it allows for a better coupling between guided waves and free-space waves. Yet, the angle-frequency relation in leaky-wave antennas only holds for very thin slits. Indeed, widening the aperture of a rectangular slit quickly



(a) w_2 = 1 mm (b) w_2 = 5 mm

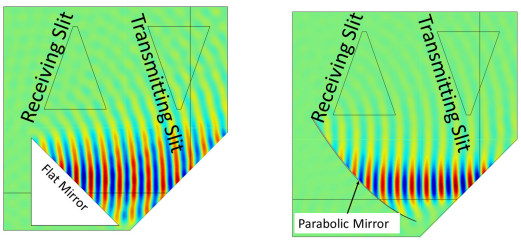
(c) w_2 = 10 mm (d) w_2 = 15 mm

Figure 8: Far field emission pattern at various slit widths, showing higher directivity gains achieved with wider slits.

invalidates the monotonic angle-frequency coupling due to the increasingly non-uniform electric field distribution along the slit. To circumvent this issue, we employ trapezoidal apertures (also explored in [16]) to increase aperture size while maintaining the coupling relations. Such slits enable us to increase the captured energy from the receiving slit as well as the out-coupled energy that leaks back out into space. Larger slits also yield smaller diffraction, thus, ray optics models are more accurate with larger slits.

We have simulated LeakyScatter with two symmetric trapezoid slits with varying slit width. Fig. 7 presents the simulated S_{21} parameter against slit widths. The smaller width of these trapezoid-shaped slits is set at $w_1 = 1$ mm while the larger width is varied from $w_2 = 10$ mm to 20 mm in COMSOL. These results were presented in the form of S-parameters, which is a metric to describe the energy propagation across different input/output ports. Here, we define S_{21} as the ratio between the reflected energy from a LeakyScatter compared to incident energy. Hence, S_{21} is a good representative of the backscattering gain. It should be noted that the S_{21} values do not directly translate to the real-world measurements; yet, the general trend can be successfully predicted by analyzing S-parameters.

From Fig. 7, we observe a clear rise in the radiated output power (by 10dB) above its initial starting point at 10 mm. We also look into these simulations in the angular space and plot the directional far-field pattern of the backscattered signal in Fig. 8. As shown, the beams become more pencil-shaped and narrow, leading to higher directivity gains. Therefore, from both simulations, we expect a higher link budget with wider trapezoid slits. However, we emphasize an important tradeoff here: intuitively, with a wider aperture, imposing a fixed power/amplitude fluctuation of P would require a greater mirror rotation angle (θ_{rot}). Unfortunately, our embedded MEMS mirrors are extremely small (mm-sized) and thus limited in their maximum rotation angle (e.g., 5° in our setup). Further, in amplitude shift keying (ASK), the number of symbols (or modulation order) depends on our flexibility to



(a) (b)

Figure 9: E-Field simulation of (a) a flat mirror and (b) an off-axis parabolic mirror showing the impact of mirror geometry on dispersion inside the waveguide.

create sufficiently distinct amplitude levels. Hence, we argue that thinner slits are more sensitive to small mirror rotations, so they support higher modulation orders. We carefully take into account the two sides of this tradeoff and fabricate LeakyScatter with the optimal slit width of $w_2 = 15$ mm.

3.4.2 Parabolic Mirror vs. Flat Mirror

The fixed mirror in LeakyScatter provides a 90-degree rotation in the trajectory of guided waves. The guided waves dispense as they travel inside the waveguide demanding a large electronically-controlled mirror to collect and rotate these waves. In order to keep the mirror size to mm-scale, we explore other possibilities for the fixed mirror. In particular, unlike flat mirrors, parabolic mirrors can focus EM radiation to a focal point. Hence, we use an 90° Off-Axis-Parabolic (OAP) mirror instead of a flat mirror to decrease dispersion in LeakyScatter. An OAP mirror is a segmented part of a parabolic mirror that diverts the incident signal by a specific angle while focusing it at the same time.

Fig. 9 illustrates the simulation results of our OAP mirror in comparison with a flat reflected surface at 173 GHz. This plot demonstrates the E-field magnitude only between the two mirrors to highlight the impact of the mirror shape. While the flat mirror successfully reflects the incident wave, it causes outward radiations as opposed to the parabolic mirror that focuses the beam to the center of the MEMS mirror. LeakyScatter thus takes advantage of an OAP mirror to accurately re-direct guided waves toward the transmitting slit and thereby increase the output power.

3.5 Concurrent Backscatter Links

The directional and wideband operation in LeakyScatter networks allows for multi-node concurrent transmissions. In particular, LeakyScatter supports a wide range of frequencies (e.g., from 100 GHz to 500 GHz). Albeit, the correct spectral band should be selected based on the angular configuration of LeakyScatter relative to the active TRX. When multiple LeakyScatter devices are sufficiently separated in the angular

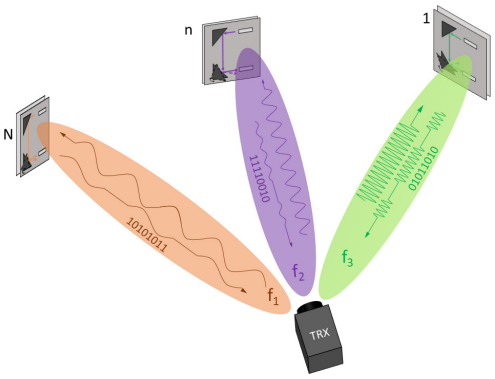


Figure 10: Concurrent transmissions of multiple LeakyScatter devices where backscattered signals can be separated in spatial and/or spectral space.

domain, they can simultaneously and independently modulate a different sub-band of the ambient wideband spectrum. This separation in the angular domain can be seen with distributed nodes at different locations around the TRX (as shown in Fig. 10) or even in co-located nodes that have different orientations. A straightforward strategy is to make TRX broadcast a pseudo-pulse in all directions to collect data from all existing nodes.[3] In such a case, spatially-separated LeakyScatters will modulate different sub-bands; thus, a simple power detection across spectrum can retrieve the information of multiple backscatter nodes. We emphasize that such a scheme relies solely on power measurements. Each LeakyScatter initiates its packet with a pre-known preamble that is utilized at TRX for node identification, localization, and steady-state power calibrations.

Interestingly, the bandwidth of a backscatter link would also depend on the incident angle of THz signals on LeakyScatter. Indeed, the non-linear angle-frequency function causes the in-coupled/out-coupled spectral range (and therefore bandwidth) to also change with angle. Particularly, the operating bandwidth for a receiver located at the far-field of a typical leaky waveguide device at an angle ϕ relative to the waveguide can be described by

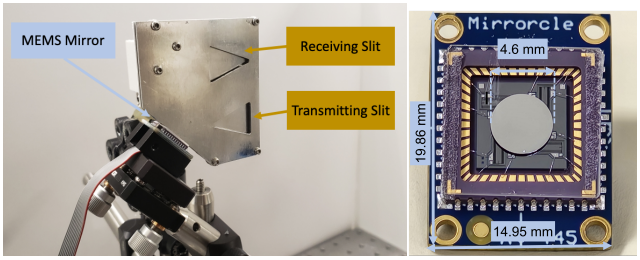
$$BW(\phi) = \frac{df}{d\beta}\frac{d\beta}{d\phi}\Delta\phi = \frac{c_0}{2b \cdot \sin\phi \tan\phi}\Delta\phi, \qquad (6)$$

where $\Delta\phi$ is the effective angular aperture subtended by the receiver and β is the wave number of guided waves.

Eq. (6) indicates that the bandwidth is wider for lower emission angles. Hence, given a fixed application-driven bandwidth per node, a non-uniform distribution of concurrent LeakyScatters is expected across the entire angular space.

In summary, the Thz LeakyScatter networks offer a two-layer protection against inter-user interference. Namely,

[3] Designing efficient medium access protocols for multi-node backscattering is out of the scope of this work and is a subject of future studies.



(a) (b)

Figure 11: Our (a) fabricated LeakyScatter with (b) a 4.6 mm electrostatic MEMS mirror.

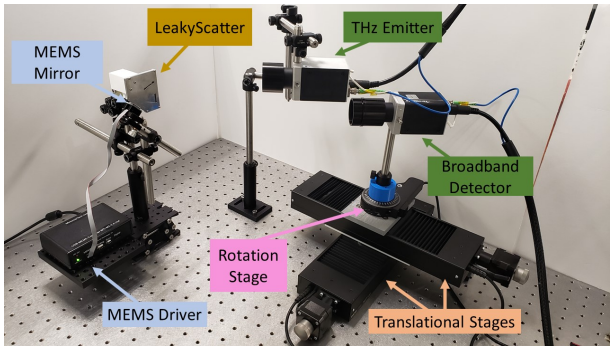


Figure 12: Our experimental setup.

the directionality of the reflected signals offers opportunities for space division multiple access and the inherent spatial-angular correlations allow the leaky-wave backscatter nodes to operate on different sub-bands according to their angular settings. This flexible multi-band operation would be self-regulated by the LeakyScatter's architecture at zero power costs, allowing scalability to dense user populations.

4 Experimental Platform and Methodology

We evaluate the performance of our LeakyScatter through extensive COMSOL simulations and over-the-air experiments. Our custom waveguide-based backscatter device, as illustrated in Fig. 11, is constructed with two thin metal plates held in parallel at a separation of 1 mm (i.e., $b = 1$ mm). The overall dimension is 60 mm $\times$ 54 mm. On the front metal plate, we cut two trapezoidal shape slits with longer and shorter widths of 1 mm and 15 mm, respectively. Both slits are 20 mm long. We embed a reflecting OAP mirror that has a smooth metal surface in between the two plates (hence, the thickness of the mirror is also 1 mm). We emphasize that building such a device is cheap (only a few cents) and can be done in a machine shop. To modulate the amplitude of backscattered signals, we employ a mm-sized MEMS mirror, offering continuous rotation in both its x and y directions (with a max rotation angle of $\pm 5^\circ$). The average power consumed by the MEMS mirror is less than 1 mW for continuous full-speed operation. The mirror is controlled via a low-profile driver with max voltage of 5 V. Resonant frequencies of these mirrors range from a few kHz up to tens of kHz and change with mirror size [38].

Our system architecture assumes a tunable wideband transceiver that is able to transmit signals above 100 GHz and steer them toward LeakyScatter nodes. Hence, we use a time-domain broadband system (TeraMetrix T-Ray [1]) that produces an ultra-short pulse with a flat frequency response between 100 GHz to 400 GHz. The detector measures the electric field magnitude at a wide range of frequencies with a spectral resolution of 1.22 GHz (sampling rate of 10 THz). The emitter and detectors are both linearly polarized with a polarization extinction ratio (the ratio of transmitted/received power between the intended and orthogonal polarization modes) $<$ 20:1. We collect raw time-domain samples and apply conventional signal processing techniques (smoothing, filtering, FFT, etc.) to isolate the signal at the spectral range of interest. Note that even though our testbed provides time-domain waveforms, LeakyScatter only uses power measurements (albeit across multiple frequencies) for data demodulation and detection.

Fig. 12 demonstrates our experimental setup consisting of the LeakyScatter and our THz emitter and detector. The emitter is configured at various settings. The detector is placed on a motorized rotation stage on top of a 2D motorized translation stage providing high-precision movement around the waveguide at various configurations. The emitter is equipped with a collimating lens that directs the THz signal into the upper slit of the waveguide.

Our evaluations are limited to an angular range of 20-60 degrees due to our transmitter's low output power. Specifically, the transmit power above 400 GHz, which corresponds to angles below 20 degrees, is always about 6 dB or more weaker, making the reflected signal harder to observe. Additionally, signals are very noisy at $<$ 170 GHz due to a combination of higher dispersion inside the waveguide, getting closer to the cutoff frequency (which is currently 150 GHz but can be changed by modifying the plate separation) and limited spectral resolution of the setup. A higher transmit power is needed to compensate for these losses and will be addressed with future THz transmitters. Nevertheless, we emphasize that our observations and conclusions presented in the paper hold true for all other angles.

5 Evaluation

In this section, we discuss our over-the-air experiments and evaluate the key design components of LeakyScatter.

5.1 Reciprocity

First, we experimentally characterize and compare the transmission and reception characteristics of leaky

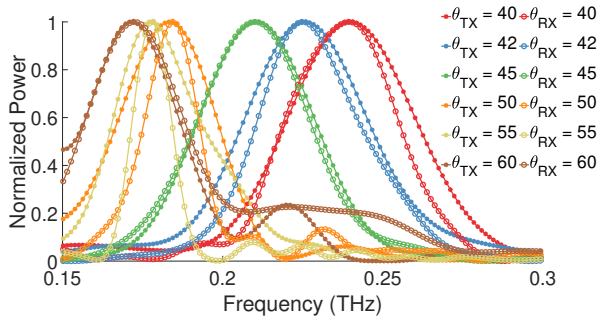


Figure 13: Experimental evaluation of the leaky-wave antenna reciprocity: The angle-frequency correlation holds in both transmission and reception modes.

waveguides to validate the reciprocity in angle frequency coupling. This is a crucial building block for designing a waveguide-based retrodirective backscatter device.

Setup. To this end, we employ two different configurations, as also shown in Fig. 3. In particular, for measuring the emission radiation pattern of a leaky-wave antenna, we focus a THz pulse into a single-slit waveguide with an external lens of focal length 75mm. The THz detector is placed on a 2D translation stage at a fixed distance where it can be configured at multiple angle directions relative to the waveguide. Similarly, our setup for measuring the reception pattern of the leaky-waveguide switches the source and the detector, i.e., the wideband source is now placed on a 2D translation stage and the detector focused into the waveguide. In both setups, the same exact waveguide was used. We configured the detector in transmission and source in reception at different angles, each corresponding to θ_{RX} and θ_{TX} from Fig. 4 respectively, ranging from 30° to 63°.

Fig. 13 shows the normalized power spectrum measurements from both reception and transmission for a few configurations, namely, 40°, 42°, 45°, 50°, 55°, and 60°. As the angle changes, we see a clear shift in the peak frequency and bandwidth in both modes (reception and transmission), obeying both equations Eq. (2) and Eq. (1). More importantly, the peak frequencies in the reception and transmission modes are in agreement, with the greatest discrepancy being a 2 GHz difference at 60°. There is, however, a discrepancy in the bandwidths of the reception and transmission measurements with the transmission bandwidths being generally larger than the reception. This is largely a result of the different distances of the detector-waveguide in transmission mode (38cm) and of the source-waveguide in reception mode (29.5cm). The shorter distance in transmission allows for a larger acceptable range of in-coupling frequencies, as opposed to the reception setup, and hence a wider bandwidth. Nevertheless, the overall spectral profiles closely follow each other.

We have experimentally illustrated that leaky-wave antennas are reciprocal, i.e., the angle-frequency relationship holds true when the antenna used in transmission and reception modes. LeakyScatter leverages this property to enable retrodirective scattering with zero power consumption.

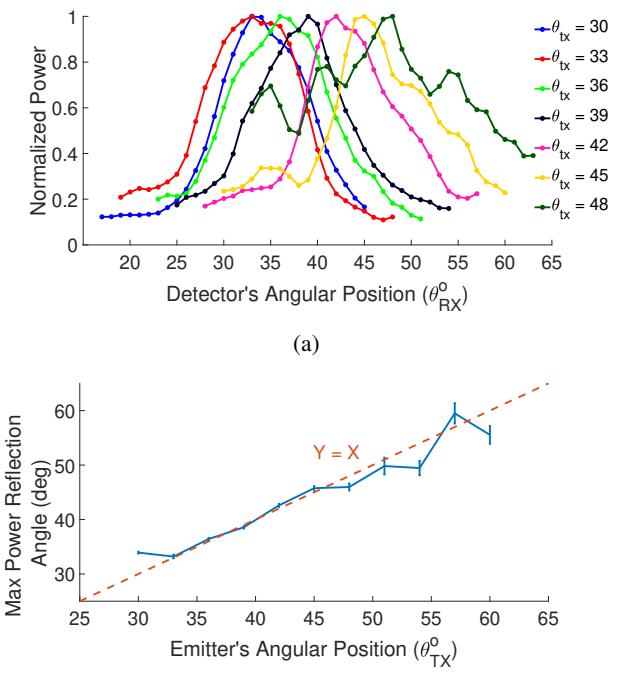


Figure 14: Experimental characterization of retrodirectivity in LeakyScatter: (a) The measured radiation patterns at different settings; (b) the max-power backscatter angle vs. the ground truth impinging angle.

5.2 Retrodirectivty

Next, we experimentally evaluate the retrodirectivity in LeakyScatter, which is the ability to steer the backscatter signals toward the transceiver. Such directional communications are essential for closing the link given the high path loss at 100 GHz.

Setup. To assess this, we use the setup illustrated in Fig. 12. We integrate a fixed mirror (instead of a rotating mirror) inside the waveguide to direct the guided waves from the receiving aperture toward the transmitting aperture. We try multiple impinging angles between the THz emitter and LeakyScatter (θ_{TX}). Each time, we move the THz detector on a 2D stage to measure the radiation pattern of directional backscatter signals. Recall that given the reciprocal angle-frequency coupling of leaky waveguides, we expect to observe the backscattered signal being strongest when the angle of reflection is the same as the impinging angle.

Fig. 14a presents the normalized power distribution of backscattered signal across space given several THz source configurations (θ_{TX}): 30°, 33°, 36°, 39°, 42°, 45°, and 48°. Given that our source is wideband, evaluation for each of the backscattering transmissions were restricted to spectral band where Eq. (5) gave minimal losses. As expected, in each of

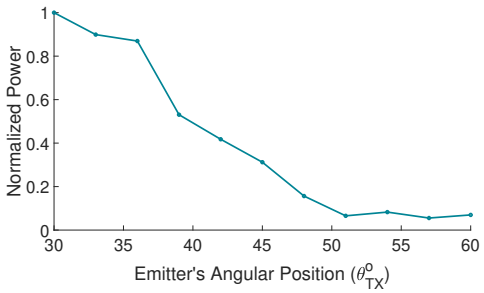


Figure 15: Normalized received power at different reflection angles.

the corresponding curves, the peak reception occurs when the reflection angle is equal to the transmission angle with a half-power beamwidth of 10.56° on average.

Fig. 14b presents these results against the expected behavior demonstrating that LeakyScatter achieves a mean error of 1.911°. We observe an increase in the overall error and fluctuations at higher impinging angles (i.e., when LeakyScatter is placed at larger angles relative to the THz emitter). We find two underlying reasons for this observation: (*i*) the inherent angle-frequency relationship in leaky waveguides is a nonlinear function such that the spectral profile is much more similar (less differential) at larger emission/acceptance angles. This implies that the main lobe of backscatter radiation is prone to slight misalignment when LeakyScatter is positioned at larger angles; and (*ii*) the overall backscattered power is not uniform at different configurations. We show the dependency of maximum reflected power with the angular location of LeakyScatter in Fig. 15. As we approach higher angles, the received power begins to drop (e.g., a drop of 10 dB is observed when the location changes from 30° to 51°). However soon after, the maximum received power begins to be less location-dependent. Although our results show that retrodirective beams can be formed, they also reveal that the directivity gain is not uniform across space. In other words, the amount of reflected power would depend on the LeakyScatter's location relative to the TRX: those positioned at smaller angles have the advantage of forming more directive/high-power backscattered links. This is to some extent a direct impact of our design specs (e.g., OAP mirror) that will be discussed later.

We have experimentally validated that our proposed backscaterring architecture is retrodirective and can establish and maintain directional connectivity with an external transceiver regardless of its location.

5.3 Data Modulation

Next, we experimentally evaluate the ability of LeakyScatter to modulate the impinging signal for data transmission.

LeakyScatter modulates the reflection path within the waveguide via an electronically-controlled mirror, thus enabling ASK by imposing different reflection power (or amplitude). Specifically, we use the same setup as in 5.2 except that we employ a mm-sized MEMS mirror for digital data encoding. As an example ASK, we map the base orientation of the Mirror ($\theta_{rot} = 0°$) to a "0" bit and the rotation angle of 5 degrees ($\theta_{rot} = 5°$) to a "1" bit. At the detector, different power levels are translated to their corresponding bit. We emphasize that this binary ASK is evaluated as an example scheme. Higher order modulation is feasible with arbitrary encoding of symbols to the embedded mirror's orientation (i.e., reflected amplitude).

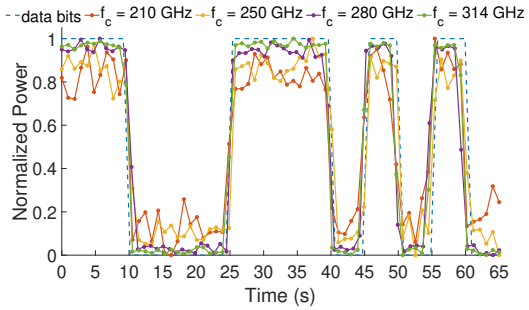


Figure 16: Measured modulated backscattered signal at various center frequencies, showing the spectral agility of LeakyScatter.

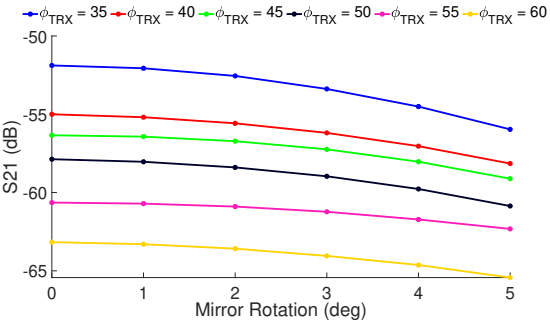
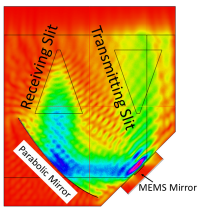


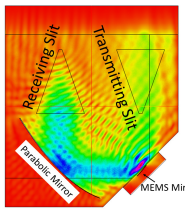
Figure 17: The sensitivity of backscatter gain (captured by S_{21} simulations) to mirror rotation in LeakyScatter.

Fig. 16 shows the successfully transmitted bit stream 0011100010101 with a symbol time of 5 seconds. We repeated this experiment at different center frequencies f_c = *210* GHz, 250 GHz, 280 GHz, and 314 GHz (corresponding to reflection angles of 45°, 37°, 32°, and 28° respectively). Our results demonstrate a successful demodulation/detection regardless of the carrier frequency or LeakyScatter's relative angular location. However, we observe that the difference between the two power levels (representing 0 and 1) gets smaller at lower frequencies. This implies that the ASK modulation order is limited for lower frequencies.

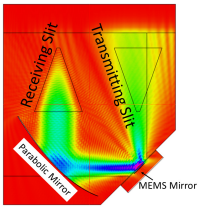
To better understand this experimental observation, we have simulated the LeakyScatter structure in COMSOL and looked into the S-parameters of this device. Specifically, Fig. 17 plots S_{21} as a measure of the backscatter power at different mirror rotations. We repeat the simulations for several impinging angle configurations. Surprisingly, at any mirror orientation, we observe a larger S_{21} at lower impinging/reflecting angles.



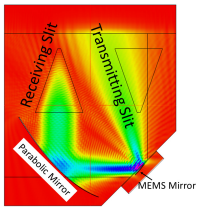
(a) $f_c = 173$ GHz, $\theta_{rot} = 0$



(b) $f_c = 173$ GHz, $\theta_{rot} = 5$



(c) $f_c = 261$ GHz, $\theta_{rot} = 0$



(d) $f_c = 261$ GHz, $\theta_{rot} = 5$

Figure 18: E-field simulations of guided waves in LeakyScatter.

Further, the power drop caused by the mirror's rotation is generally more pronounced at smaller angles (e.g., an S_{21} drop of 2.25 dB at $\theta_{TRX} = 60°$ vs. a 4.1 dB drop at $\theta_{TRX} = 35°$. Note that this trend holds for negative rotations (not shown).

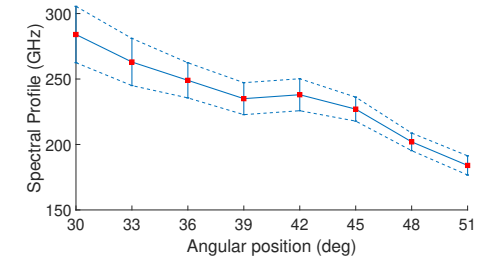
To find the underlying reason, we looked further into the simulated E-field inside the waveguide at the same device configurations. Fig. 18 depicts this with an in-coupled signal of 173 GHz and 261 GHz signal at rotation angles $\theta_{rot} = 0$ and $\theta_{rot} = 5$. Interestingly, we observe less dispersion for the higher frequency tone; i.e., the guided waves are much more directed (laser-like) at 261 GHz, especially as they interact with the embedded parabolic mirror. The propagation path of guided waves can thus be more precisely controlled via a rotating mirror. There is a clear drop in the amount of power at the second (transmitting) slit due to the 5° mirror rotation. This implies that the ASK modulation order (i.e, the number of symbols which here correlate with the number of distinct power levels using an electronically-controlled rotating mirror) is frequency-dependent with higher frequencies offering higher modulation orders.

We have experimentally demonstrated that we can piggyback information bits on the backscattered signal with LeakyScatter. Interestingly, a higher modulation order can be achieved at higher frequencies due to their small dispersion and laser-like behavior in guided mode.

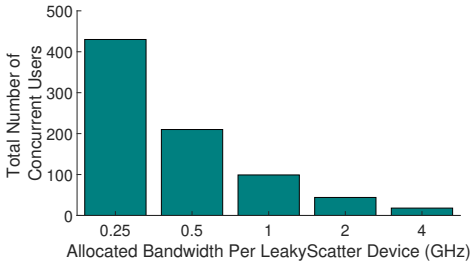
5.4 Multiple Concurrent LeakyScatters

Finally, we investigate the user capacity in LeakyScatter networks. We consider a single transceiver and emulate multi-node backscattering by placing the LeakyScatter at various locations and collecting the raw power spectrum.

First, we experimentally evaluate the half-power bandwidth at various TRX-LeakyScatter angular configurations. As discussed in Sec. 3.5, we expect larger half-power bandwidths



(a)



(b)

Figure 19: Experimental characterization of user density: (a) Spectral band of operation in LeakyScatter when configured at various impinging angles; (b) Total concurrent LeakyScatter nodes given their application-driven bandwidth requirements.

at lower incident angles. Fig. 19a confirms this claim through experimental analysis where the bandwidth of backscatter links are captured across different angular positions (all at a fixed distance of 34 cm from LeakyScatter). This observation suggests that the concurrent user capacity (given a per-user bandwidth) is also non-uniform across space, and dependent on the incident angle with a similar trend.

Fig. 19b shows the emulated total number of concurrent users across the angular domain. We assume directional pencil-shaped beams (1° of beamwidth) at TRX[4]. Further, we emulate five different bandwidths for backscatter communication. Note that in practice, this number is application-dependent. According to Fig. 19b, LeakyScatter can establish 430 simultaneous non-overlapping links with a spectral allocation of 0.25 GHz per user. Similarly, 18 users can be supported at a spectral allocation of 4 GHz each. We emphasize that different backscatter nodes modulate different portions of the spectrum (according to their location and orientation); thereby, having multiple nodes is a simple extension of a single LeakyScatter as the signals are orthogonal in frequency. In the future, we will experimentally implement a multi-backscatter network using a multi-beam multi-frequency emitter.

We have experimentally demonstrated that by taking advantage of LeakyScatter's retrodirectivity and spectral agility, we can support multiple concurrent backscatter nodes in dense user settings.

[4]For simplicity, we consider the far-field scenario in which the backscatter range is much larger than the dimension of the physical LeakyScatter nodes.

6 Discussion and Limitations

In this section, we discuss the opportunities and limitations in THz backscatter networks as well as our future directions.

Mobility Support with LeakyScatter. One of the main challenges in directional communication is maintaining beam alignment despite nodal mobility. LeakyScatter offers a retrodirective link which means the backscatter beam will always point back toward the broadband TRX regardless of device motion. Interestingly, this will be accompanied by a change in the spectral profile observed at the TRX. This opens the door for simultaneous data transmission and localization, an emerging paradigm for 6G wireless systems. Note that leaky-wave antennas have been used recently for angular positing in unmodulated and active settings (non backscattering) [11, 13, 28]. This paper is the *first* effort that exploits the angle-frequency coupling in leaky-wave devices for enabling low-power frequency-agile backscattering at the sub-THz regime.

Communication Range and Coverage. A common concern in all backscattering networks is the limited communication range, i.e., the range of a backscatter link is always smaller than an active TX-RX link due to lack of amplification at the backscatter. This issue worsens in sub-THz and THz frequencies that suffer from increased propagation attenuation. Yet, our experimental results demonstrate the feasibility of closing a link at frequencies as high as 314 GHz without any amplifications and via an ultra-low-power broadband emitter/detector. Our source has an average output power of 400 nW across 5 THz non-uniformly. Our current implementation of LeakyScatter, however, has an operating bandwidth of approximately 400 GHz which upon calculation yields a 260 nW average power output. Even under such stringent power budgets, a communication range of half a meter was achieved. This is owed to the directionality at LeakyScatter which compensates for the higher path loss. Note that our communication range is similar to the numbers reported by prior work that uses an active leaky-wave antenna [13, 28] which implies that the limitation in range is a direct result of the employed low-power source and not the architecture of LeakyScatter. Employing more realistic TRX can scale up the transmitter-receiver distance as links at WLAN-scale distances (100+) meters have already been demonstrated above 100 GHz [59, 60].

We note that LeakyScatter is limited in angular coverage of 90°. Prior work proposed a periodic arrangement of slits to extend the coverage of leaky-wave antennas to 180° [32]. To extend the coverage to 360°, open slits on both ends are needed. One potential design is to use three aluminum plates such that two of the plates contain periodic slits and the 'center plate' acts as the common 'back plate'. We will investigate these architectures in the future.

Fabrication Cost and Power Consumption. LeakyScatter can be easily fabricated with two thin metal plates, spacers, and an electronically-controlled mirror. Prior work has realized leaky-wave antennas via integrated circuits, i.e., in 65nm CMOS with an area size of $3mm^2$ [50]. The small form factor, ease of fabrication, and low cost make such designs suitable for various IoT applications. The integrated broadband emitter/detector, however, can be power demanding. We emphasize that such asymmetry between the transceiver and the backscatter tag is typical in these networks [34]. Nevertheless, ongoing research continues to develop efficient low-power THz transmitter and detectors suitable for future handheld devices [5, 49, 51].

Higher Order Modulation. We have introduced a novel low-power physical layer architecture for THz communication by showing the feasibility of passive THz backscattering; nevertheless, further extensive study is needed for a full-stack demonstration. Additionally, the current achievable data-rates fall behind the speeds promised within the THz regime. The underlying bottleneck is the speed of the electro-static MEMS mirrors which can be addressed through novel modulation strategies. Building high-speed THz modulators is itself an emerging field of research, with recent advances in materials (e.g., carbon nanotubes and graphene) showing promise [4, 7, 39]. In this paper, we consider a noncoherent on–off keying modulation scheme, which is one of the two identified modes in the recent standardization of the IEEE 802.15.3d task group [42]. Nevertheless, in the future, we will explore not only coherent modulation techniques and higher-order amplitude-phase schemes, but also optical modulation techniques, such as via variation in waveguide's electromagnetic permittivity, for larger data-rates.

7 Conclusion

In this paper, we present LeakyScatter, a novel structure that enables low-power directional backscattering above 100 GHz. LeakyScatter adopts the inherent dependency of emission angle on frequency in LWAs to passively redirect the signals back in the same direction from which they were transmitted. The proposed architecture utilizes a metal parallel-plate waveguide with two open slits. Data transmission is facilitated via a mm-sized MEMS mirror that modulates the reflection loss (i.e., amplitude of backscattered link) according to the bit stream. Our over-the-air experiments show that LeakyScatter is ultra-wideband and spectrally-agile operating at few 100s of GHz, opening unique opportunities for dense user implementations.

8 Acknowledgments

We appreciate the valuable comments and feedback from the anonymous reviewers. This research was supported by the US Air Force, and NSF grants CNS-2145240 and CNS-2148271.

References

[1] The TeraMetrix T-Ray® 5000 Series Intelligent Terahertz Control Unit. Available at https://lunainc.com/blog/terametrix-t-rayr-5000-series-intelligent-terahertz-control-unit, 2018.

[2] BALANIS, C. A., AND BIRTCHER, C. R. Antenna measurements. *Modern Antenna Handbook* (2008), 977–1033.

[3] BOYER, C., AND ROY, S. Backscatter Communication and RFID: Coding, Energy, and MIMO Analysis. *IEEE Transactions on Communications 62*, 3 (2014), 770–785.

[4] BURDANOVA, M., TSAPENKO, A., SATCO, D., KASHTIBAN, R., MOSLEY, C., MONTI, M., STANIFORTH, M., SLOAN, J., GLADUSH, Y. G., NASIBULIN, A. G., AND LLOYD-HUGHES, J. Efficient Ultrafast THz Modulators Based on Negative Photoconductivity in Controllably Doped Carbon Nanotubes. In *International Conference on Infrared, Millimeter, and Terahertz Waves (IRMMW-THz)* (2019), pp. 1–1.

[5] CHI, T., HUANG, M.-Y., LI, S., AND WANG, H. 17.7 A Packaged 90-to-300GHz Transmitter and 115-to-325GHz Coherent Receiver in CMOS for Full-Band Continuous-wave mm-Wave Hyperspectral Imaging. In *In Proc. of IEEE International Solid- State Circuits Conference (ISSCC)* (2017).

[6] CORREIA, R., DING, Y., DASKALAKIS, S. N., PETRIDIS, P., GOUSSETIS, G., GEORGIADIS, A., AND CARVALHO, N. B. Chirp Based Backscatter Modulation. In *IEEE MTT-S International Microwave Symposium* (2019), pp. 279–282.

[7] DOCHERTY, C. J., STRANKS, S. D., HABISREUTINGER, S. N., JOYCE, H. J., HERZ, L. M., NICHOLAS, R. J., AND JOHNSTON, M. B. An Ultrafast Carbon Nanotube Terahertz Polarisation Modulator. *Journal of Applied Physics 115*, 20 (2014), 203108.

[8] DUTTA, S., BARATI, C. N., RAMIREZ, D., DHANANJAY, A., BUCKWALTER, J. F., AND RANGAN, S. A Case for Digital Beamforming at mmWave. *IEEE Transactions on Wireless Communications 19*, 2 (2019), 756–770.

[9] EID, A., HESTER, J., AND TENTZERIS, M. M. A Scalable High-gain and Large-beamwidth mm-wave Harvesting Approach for 5G-powered IoT. In *IEEE MTT-S International Microwave Symposium* (2019), pp. 1309–1312.

[10] FEDERAL COMMUNICATIONS COMMISSION. The Commission Seeks Comment on Spectrum for the Internet of Things. *ET Docket*, 21-353 (2021).

[11] GHASEMPOUR, Y., AMARASINGHE, Y., YEH, C.-Y., KNIGHTLY, E., AND MITTLEMAN, D. M. Line-of-Sight and Non-Line-of-Sight Links for Dispersive Terahertz Wireless Networks. *APL Photonics 6*, 4 (2021), 041304.

[12] GHASEMPOUR, Y., SHRESTHA, R., CHAROUS, A., KNIGHTLY, E., AND MITTLEMAN, D. M. Single-Shot Link Discovery for Terahertz Wireless Networks. *Nature Communication 11*, 1 (2020), 2017.

[13] GHASEMPOUR, Y., YEH, C.-Y., SHRESTHA, R., AMARASINGHE, Y., MITTLEMAN, D., AND KNIGHTLY, E. W. LeakyTrack: Non-Coherent Single-Antenna Nodal and Environmental Mobility Tracking with a Leaky-Wave Antenna. In *Proc. of ACM SenSys* (2020), pp. 56–68.

[14] GHASEMPOUR, Y., YEH, C.-Y., SHRESTHA, R., MITTLEMAN, D., AND KNIGHTLY, E. Single Shot Single Antenna Path Discovery in THz Networks. In *Proc. of ACM MobiCom* (2020), pp. 317–327.

[15] GROSS, F. B. *Frontiers in Antennas: Next Generation Design & Engineering*. McGraw-Hill Education, 2011.

[16] GUERBOUKHA, H., SHRESTHA, R., NERONHA, J., RYAN, O., HORNBUCKLE, M., FANG, Z., AND MITTLEMAN, D. M. Efficient Leaky-Wave Antenna for Terahertz Wireless Communications. In *Conference on Lasers and Electro-Optics* (2021), Optical Society of America.

[17] GUO, X., SHANGGUAN, L., HE, Y., JING, N., ZHANG, J., JIANG, H., AND LIU, Y. Saiyan: Design and implementation of a low-power demodulator for LoRa backscatter systems. In *Proc. of USENIX NSDI* (Apr. 2022), pp. 437–451.

[18] HESSAR, M., NAJAFI, A., AND GOLLAKOTA, S. NetScatter: Enabling Large-Scale backscatter networks. In *Proc. of USENIX NSDI* (2019), pp. 271–284.

[19] HESTER, J. G., AND TENTZERIS, M. M. A Mm-wave Ultra-long-range Energy-autonomous Printed RFID-enabled Van-atta Wireless Sensor: At the Crossroads of 5G and IoT. In *IEEE MTT-S International Microwave Symposium* (2017), pp. 1557–1560.

[20] ILIE-ZUDOR, E., KEMÉNY, Z., VAN BLOMMESTEIN, F., MONOSTORI, L., AND VAN DER MEULEN, A. A survey of Applications and Requirements of Unique Identification Systems and RFID Techniques. *Computers in Industry 62*, 3 (2011), 227–252.

[21] JACKSON, D. R., AND OLINER, A. A. Leaky-Wave Antennas. *Modern Antenna Handbook* (2008), 325–367.

[22] JAMEEL, F., DUAN, R., CHANG, Z., LILJEMARK, A., RISTANIEMI, T., AND JANTTI, R. Applications of Backscatter Communications for Healthcare Networks. *IEEE Network 33*, 6 (2019), 50–57.

[23] JIANG, J., XU, Z., DANG, F., AND WANG, J. Long-range Ambient LoRa Backscatter with Parallel Decoding. In *Proc. of ACM MobiCom* (2021), pp. 684–696.

[24] KARL, N. J., MCKINNEY, R. W., MONNAI, Y., MENDIS, R., AND MITTLEMAN, D. M. Frequency-division Multiplexing in the Terahertz Range using a Leaky-wave Antenna. *Nature Photonics 9*, 11 (2015), 717.

[25] KELLOGG, B., PARKS, A., GOLLAKOTA, S., SMITH, J. R., AND WETHERALL, D. Wi-Fi Backscatter: Internet Connectivity for RF-powered devices. In *Proc. of ACM SIGCOMM* (2014), pp. 607–618.

[26] KIMIONIS, J., GEORGIADIS, A., DASKALAKIS, S. N., AND TENTZERIS, M. M. A Printed Millimetre-wave Modulator and Antenna Array for Backscatter Communications at Gigabit Data Rates. *Nature Electronics 4*, 6 (2021), 439–446.

[27] KLUDZE, A., AND GHASEMPOUR, Y. Towards Terahertz Wireless Authentication with Unique Aperture Fingerprints using Leaky-Wave Antennas. In *International Conference on Infrared, Millimeter, and Terahertz Waves (IRMMW-THz)* (2022).

[28] KLUDZE, A., SHRESTHA, R., KNIGHTLY, E., MITTLEMAN, D., AND GHASEMPOUR, Y. 3D Localization via a Single Non-Coherent THz Antenna. In *Proc. of ACM MobiCom* (2022).

[29] LI, S., ZHENG, H., ZHANG, C., SONG, Y., YANG, S., CHEN, M., LU, L., AND LI, M. Passive DSSS: Empowering the Downlink Communication for Backscatter Systems. In *Proc. of USENIX NSDI* (Apr. 2022), pp. 913–928.

[30] LI, Z., CHEN, B., YANG, Z., LI, H., XU, C., CHEN, X., WANG, K., AND XU, W. FerroTag: A Paper-Based MmWave-Scannable Tagging Infrastructure. In *Proc. of ACM SenSys* (2019), p. 324–337.

[31] LIANG, Q., DURRANI, T. S., GU, X., KOH, J., LI, Y., AND WANG, X. Guest Editorial Special Issue on Spectrum and Energy Efficient Communications for Internet of Things. *IEEE Internet of Things Journal 6*, 4 (2019), 5948–5953.

[32] LIU, J., JACKSON, D. R., AND LONG, Y. Substrate integrated waveguide (SIW) Leaky-Wave Antenna with Transverse Slots. *IEEE Transactions on Antennas and Propagation 60*, 1 (2011), 20–29.

[33] LIU, V., PARKS, A., TALLA, V., GOLLAKOTA, S., WETHERALL, D., AND SMITH, J. R. Ambient backscatter: Wireless communication out of thin air. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 39–50.

[34] LIU, W., HUANG, K., ZHOU, X., AND DURRANI, S. Next Generation Backscatter Communication: Systems, Techniques, and Applications. *EURASIP Journal on Wireless Communications and Networking 2019*, 1 (2019), 1–11.

[35] LONG, S., AND MIAO, F. Research on ZigBee Wireless Communication Technology and its Application. In *IEEE IAEAC* (2019), vol. 1, pp. 1830–1834.

[36] LUETH, K. L. State of the IoT 2020: 12 billion IoT Connections, Surpassing Non-IoT for the First Time. Available at https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/, 2020.

[37] MAZAHERI, M. H., CHEN, A., AND ABARI, O. MmTag: A Millimeter Wave Backscatter Network. In *Proc. of ACM SIGCOMM* (2021), p. 463–474.

[38] MIRRORCLE. MEMS Mirrors - Mirrorcle Technologies Inc. Available at https://www.mirrorcletech.com/wp/products/mems-mirrors/ (2022/04/20), 2022.

[39] MITTENDORFF, M., LI, S., AND MURPHY, T. E. Graphene-based Waveguide-Integrated Terahertz Modulator. *ACS Photonics 4*, 2 (2017), 316–321.

[40] NADERIPARIZI, S., HESSAR, M., TALLA, V., GOLLAKOTA, S., AND SMITH, J. R. Towards Battery-free HD Video Streaming. In *Proc. of USENIX NSDI* (2018), pp. 233–247.

[41] PENG, Y., SHANGGUAN, L., HU, Y., QIAN, Y., LIN, X., CHEN, X., FANG, D., AND JAMIESON, K. PLoRa: A Passive Long-Range Data Network from Ambient LoRa Transmissions. In *Proc. of ACM SIGCOMM* (2018), pp. 147–160.

[42] PETROV, V., KURNER, T., AND HOSAKO, I. IEEE 802.15.3d: First Standardization Efforts for Sub-Terahertz Band Communications toward 6G. *Comm. Mag. 58*, 11 (Nov 2020), 28–33.

[43] POLEMI, A., AND MACI, S. On the Polarization Properties of a Dielectric Leaky Wave Antenna. *IEEE Antennas and Wireless Propagation Letters 5* (2006), 306–310.

[44] PRADHAN, S., CHAI, E., SUNDARESAN, K., QIU, L., KHOJASTEPOUR, M. A., AND RANGARAJAN, S. Rio: A Pervasive RFID-based Touch Gesture Interface. In *Proc. of ACM MobiCom* (2017), pp. 261–274.

[45] QIN, Z., LI, F. Y., LI, G. Y., MCCANN, J. A., AND NI, Q. Low-power Wide-area Networks for Sustainable IoT. *IEEE Wireless communications 26*, 3 (2019), 140–145.

[46] RANGAN, S., RAPPAPORT, T. S., AND ERKIP, E. Millimeter-Wave Cellular Wireless Networks: Potentials and Challenges. *Proceedings of the IEEE 102*, 3 (2014), 366–385.

[47] RAPPAPORT, T. S., HEATH JR, R. W., DANIELS, R. C., AND MURDOCK, J. N. *Millimeter Wave Wireless Communications*. Pearson Education, 2015.

[48] RAPPAPORT, T. S., SUN, S., MAYZUS, R., ZHAO, H., AZAR, Y., WANG, K., WONG, G. N., SCHULZ, J. K., SAMIMI, M., AND GUTIERREZ, F. Millimeter Wave Mobile Communications for 5G Cellular: It Will Work! *IEEE Access 1* (2013), 335–349.

[49] REYNAERT, P., STEYAERT, W., STANDAERT, A., SIMIC, D., AND KAIZHE, G. mm-Wave and THz Circuit Design in Standard CMOS Technologies: Challenges and Opportunities. In *IEEE Asia Pacific Microwave Conference* (2017), pp. 85–88.

[50] SAEIDI, H., VENKATESH, S., LU, X., AND SENGUPTA, K. 22.1 THz Prism: One-Shot Simultaneous Multi-Node Angular Localization Using Spectrum-to-Space Mapping with 360-to-400GHz Broadband Transceiver and Dual-Port Integrated Leaky-Wave Antennas. In *IEEE International Solid- State Circuits Conference (ISSCC)* (2021), vol. 64, pp. 314–316.

[51] SENGUPTA, K., AND HAJIMIRI, A. A 0.28 THz Power-Generation and Beam-Steering Array in CMOS Based on Distributed Active Radiators. *IEEE Journal of Solid-State Circuits 47*, 12 (2012), 3013–3031.

[52] SKRIMPONIS, P., DUTTA, S., MEZZAVILLA, M., RANGAN, S., MIRFARSHBAFAN, S. H., STUDER, C., BUCKWALTER, J., AND RODWELL, M. Power Consumption Analysis for Mobile mmWave and Sub-THz Receivers. In *6G Wireless Summit* (2020), IEEE, pp. 1–5.

[53] SOLTANAGHAEI, E., PRABHAKARA, A., BALANUTA, A., ANDERSON, M., RABAEY, J. M., KUMAR, S., AND ROWE, A. Millimetro: MmWave Retro-Reflective Tags for Accurate, Long Range Localization. In *Proc. of ACM MobiCom* (2021), p. 69–82.

[54] SORNIN, N., LUIS, M., EIRICH, T., KRAMP, T., AND HERSENT, O. LoRaWAN Specifications, LoRa Alliance, San Ramon.

[55] SUTINJO, A., OKONIEWSKI, M., AND JOHNSTON, R. H. Radiation from fast and slow traveling waves. *IEEE Antennas and Propagation Magazine 50*, 4 (2008), 175–181.

[56] TALLA, V., HESSAR, M., KELLOGG, B., NAJAFI, A., SMITH, J. R., AND GOLLAKOTA, S. Lora Backscatter: Enabling the vision of Ubiquitous Connectivity. In *Proc. of ACM IMWUT* (2017), vol. 1, pp. 1–24.

[57] TONG, X., ZHU, F., WAN, Y., TIAN, X., AND WANG, X. Batch Localization Based on OFDMA backscatter. In *Proc. of the ACM IMWUT* (2019), vol. 3, pp. 1–25.

[58] WANG, Q., YU, J., XIONG, C., ZHAO, J., CHEN, S., ZHANG, R., AND GONG, W. Efficient Backscatter with Ambient WiFi for Live Streaming. In *IEEE Global Communications Conference* (2020), pp. 1–6.

[59] Y. YANG, M. MANDEHGAR, AND D. R. GRISCHKOWSKY. THz-TDS Characterization of the Digital Communication Channels of the Atmosphere and the Enabled Applications. *Journal of Infrared, Millimeter, and Terahertz Waves 36*, 1 (2015), 97–129.

[60] YANG, Y., MANDEHGAR, M., AND GRISCHKOWSKY, D. R. Understanding THz Pulse Propagation in the Atmosphere. *IEEE Transactions on Terahertz Science and Technology 2*, 4 (2012), 406–415.

[61] YEH, C.-Y., GHASEMPOUR, Y., AMARASINGHE, Y., MITTLEMAN, D. M., AND KNIGHTLY, E. W. Security in Terahertz WLANs with Leaky Wave Antennas. In *Proc. of ACM WiSec* (2020), pp. 317–327.

[62] ZHANG, J., SOLTANAGHAI, E., BALANUTA, A., GRIMSLEY, R., KUMAR, S., AND ROWE, A. PLatter: On the Feasibility of Building-Scale Power Line Backscatter. In *Proc. of USENIX NSDI 22* (Apr. 2022), pp. 897–911.

RF-Bouncer: A Programmable Dual-band Metasurface for Sub-6 Wireless Networks

Xinyi Li[1,*], *Chao Feng*[1,*], *Xiaojing Wang*[1], *Yangfan Zhang*[1], *Yaxiong Xie*[2], *Xiaojiang Chen*[1,†]
[1]*Northwest University,* [2]*University at Buffalo SUNY*

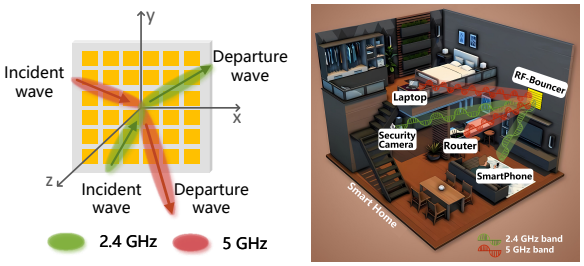
Abstract

Offloading the beamforming task from the endpoints to the metasurface installed in the propagation environment has attracted significant attention. Currently, most of the metasurface-based beamforming solutions are designed and optimized for operation on a single ISM band (either 2.4 GHz or 5 GHz). In this paper, we propose RF-Bouncer, a compact, low-cost, simple-structure programmable dual-band metasurface that supports concurrent beamforming on two Sub-6 ISM bands. By configuring the states of the meta-atoms, the metasurface is able to simultaneously steer the incident signals from two bands towards their desired departure angles. We fabricate the metasurface and validate its performance via extensive experiments. Experimental results demonstrate that RF-Bouncer achieves 15.4 dB average signal strength improvement and a 2.49× throughput improvement even with a relatively small 16 × 16 array of meta-atoms.

1 Introduction

It is a common practice for wireless communication systems to leverage beamforming technique to improve the throughput and extend the communication range. Higher beamforming gain requires a larger number of antennas installed on the communication endpoints. Two practical challenges hinder the deployment of radio systems with a large antenna array. First, the majority of today's IoT devices have to be small in size due to cost and form factor constraints, leaving no space for a large array. Second, the radio chains connected to each antenna increase hardware costs and power consumption.

Recently, offloading the beamforming from the communication endpoints to a metasurface deployed in the propagation environment has attracted significant attention [4, 14]. RFocus [4] leverages a metasurface that consists of thousands of simple 2-way RF switches to beamform the incoming signal towards the receiver. Due to the limited programmability of the RF switch, *i.e.*, switch on for reflecting and switch off for no reflection, RFocus needs huge number of meta-atoms to work efficiently and robustly. RFlens [14] upgrades the metasurface with a dedicated meta-atom that resembles a 1-bit phase shifter and thus achieves reasonable beamforming performance with only 256 functional meta-atoms.



(a) RF-Bouncer principle. (b) 3D scenario.

Figure 1: RF-Bouncer's metasurface simultaneously steers the incident signal towards the target directions at two bands, improving the overall performance of dual-band concurrent communication in a complex 3D indoor environment.

We observe that most of the current metasurface-based beamforming solutions focus on optimizing communication performance on single frequency band, for example, RFocus works for frequencies below 3 GHz while RFlens is optimized for 5 GHz band. But, these two Sub-6 ISM bands at 2.4 GHz and 5 GHz accommodate three wireless protocols widely used for communication between IoT devices: Wi-Fi, Bluetooth and ZigBee. Due to the densely deployed IoT devices, concurrent wireless transmissions over two Sub-6 ISM bands are very common. A naive solution to extend existing solutions to support dual-band operation is to install two meta-surfaces, one for a single band. Such a solution not only requires more deployment space to accommodate the extra metasurfaces but also results in higher costs. Stacking one metasurface on top of another [42] is another option to support the dual-band operation, resulting in a complicated circuit design. More recent attempts [11, 14, 26, 32] employ varactor to adjust phase, which incurs a high insertion loss [26, 32] and require a precise

[*]Co-primary authors, both authors contributed equally to this research.
[†]Corresponding author.

and sophisticated DC voltage control backend [11, 14].

In this paper, we propose to design an area-efficient, low-cost, and simple-structure programmable dual-band metasurface that supports concurrent beamforming on two ISM bands. By concurrent beamforming we mean the metasurface is able to simultaneously steer the two incident signals from two ISM bands towards their desired departure angles, just as shown in Figure 1. The locations of the two pairs of transceivers are random in practice, so our metasurface should work with a arbitrary combination of two incident angles and two departure angles in 3-D space, as shown in Figure 1 (b).

Designing a dual-band metasurface is challenging. Generally, to maximize the communication efficiency, the electric length of the RF components (meta-atoms) should match the wavelength of the signal. There exists a large discrepancy in the wavelength of signals from two widely separated bands, for example, the wavelength is 12 cm and 6 cm for signals at frequency 2.4 GHz and 5 GHz, respectively. Therefore, it is difficult to fabricate hardware with fixed physical dimensions but multiple resonance frequencies.

To solve the problem, we propose a novel meta-atom that has two resonant frequencies (bi-resonant), by integrating two antenna structures. The basic structure of our meta-atom is a metal-backed patch square structure. By adjusting its physical dimensions, we successfully fix its resonant frequency to the first ISM band (2.4 GHz). To generate an additional resonant frequency, we propose to etch slots on the patch, since the slots impact the path of the stimulated current and thus the resonant frequency, according to antenna theory [5]. By fine-tuning the location, the number and the physical dimensions of the slots on the patch, we successfully generate the second resonant frequency at the second ISM band (5 GHz) without affecting the first resonant frequency.

To empower the meta-atoms with dual-band programmability, we embed two PIN diodes into carefully selected positions of the patch on the meta-atom. Each PIN diode functions similarly to an RF switch, and two PIN diodes provide four "on/off" states. Depending on the state of the PIN diodes, the meta-atom introduces different amount of phase shifts to its reflected signal, resembling a 2-bit phase shifter.

Based on our programmable dual-band metasurface, we implement a dual-band beamforming algorithm that can quickly configure the states of all meta-atoms to accurately steer the incident signal towards the desired departure angle. We also design a beam alignment algorithm to adjust the configurations of meta-atoms in real-time to handle user mobility.

We build a prototype of RF-Bouncer's metasurface by embedding 16×16 meta-atoms inside an area of $0.35 \times 0.35m^2$. Owing to its small form factor, RF-Bouncer's metasurface can be attached to the facades of the ambient environment such as walls, furniture, and advertisement boards. Hence, RF-Bouncer can easily cope with complex indoor environments, as shown in Figure 1. Extensive experiments demonstrate that even with the small-size prototype, RF-Bouncer enables 15.4 dB average signal strength improvement and a 2.49× throughput improvement. RF-Bouncer also works robustly across protocols (e.g., Bluetooth, Zigbee and Wi-Fi), and in complex radio environments (3D and even NLoS).

Contributions. The main contributions of RF-Bouncer are: *(i)* We design a programmable dual-band metasurface that supports concurrent beamforming over two ISM bands[1]. *(ii)* We implement a dual-band beamforming algorithm that can quickly configure the metasurface to simultaneously steer the incident signals of two bands towards their desired departure directions. *(iii)* We fabricate RF-Bouncer's metasurface and validate its effectiveness in a wide range of practical scenarios.

2 Related Work

Metasurfaces and smart surfaces. Metasurfaces are three-dimensional, periodic, and artificial structures [6, 10, 17, 30]. By manipulating the phase/ amplitude of electromagnetic waves, it can beamform or re-steer the signals towards an intended direction, so as to extend the network coverage. MilliMirror [28] utilizes a 3D printed metasurface to re-steer mmWave beams to illuminate coverage blind spots. Although promising, such metasurfaces are not configurable. To enable programmability, prior studies focus on adding electronic components (i.e., varactors [8, 12, 15] or PIN diodes [35, 40, 41]) into the metasurface. Another line of literature improves indoor network coverage by designing and deploying smart surfaces in the environment to manipulate wireless channels. These smart surfaces generally consist of non-periodic but adjustable electronic components [4, 13, 39]. While the above methods have shown great promise, they mostly focus on single-link optimizations and are not yet optimized for dual-band concurrent links or 3D coverage. Unlike them, RF-Bouncer aims to simultaneously support dual-band wireless links (e.g., 2.4 GHz and 5 GHz) and targets indoor 3D network coverage improvement.

Expanding indoor wireless coverage. To expand the wireless coverage, several systems [9, 16, 20, 27, 36] deploy passive reflectors near the AP to reflect the incident signal to enhance the link SNR. Such reflectors, however, cannot be reconfigured, resulting that they cannot adapt to dynamic indoor environments. Instead, RF-Bouncer can dynamically configure the metasurface to reshape incident beams, thus adapting to dynamic indoor environments. Alternatively, some studies improve indoor wireless coverage by installing multiple APs [24, 25, 29, 34] or RF relays [2, 7, 18] in the environment. Yet, when applying to a new wireless standard or working frequency band, these approaches require updating protocols or hardware, which is cumbersome and high implementation cost. In contrast, RF-Bouncer is a standard-agnostic and cost-effective solution to enhance indoor wireless coverage.

[1]The design of RF-Bouncer is available at: https://github.com/ZYF-PhD/RF-Bouncer_OpenSource

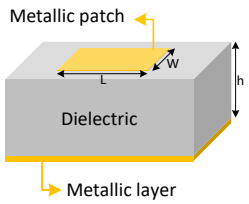


Figure 2: The incipient structure of meta-atom.

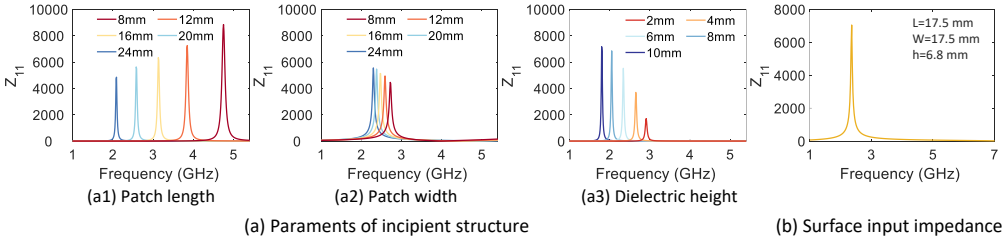


Figure 3: Impact of the paraments of the incipient structure and the surface input impedance across operating frequencies.

Dual-band design. Many efforts have been devoted to designing dual-band metasurfaces. Stacking two metasurfaces with two resonant frequencies is a straightforward solution [42], which however results in complicated circuit design. Some studies [22, 37] exploit polarization orthogonality to enable dual-band reflectarray antenna, one polarization for each band, while RF-Bouncer supports dual-band with the same polarization, without requiring low-cost IoT devices to rotate when switching the working frequency bands. [23] propose a metasurface structure that supports dual-band but only produces two symmetrical reflected beams, lacking programmability in controlling the departure direction. Some recent attempts [11, 32] employ varactor to adjust phase, which incurs a high insertion loss [26, 32] and requires a precise and sophisticated DC voltage control backend [11, 14]. [3] also supports dual-band operation, but focuses on blocking the signal from one band (either 2.4 GHz or 5 GHz) from penetrating through, whose purpose is entirely different from RF-Bouncer. The most relevant work is [31], which proposes a similar structure to support dual-band frequencies of the same polarization, but it only focuses on simulation and merely fabricates two meta-atoms as a proof-of-concept prototype. Compared with the existing dual-band design, RF-Bouncer leverages PIN diodes and a simple square patch to achieve a programmable dual-band metasurface that supports dual-band operation with the same polarization. Due to the simplicity of its structure, the proposed metasurface is easy to fabricate and thus can be easily embedded into the environment to support various IoT devices. Furthermore, RF-Bouncer designs a dual-band beamforming algorithm that can quickly configure the states of all meta-atoms to accurately steer the incident signal towards the desired departure angle, which has not been implemented by any of the prior works.

3 Hardware Design of the Metasurface

In this section, We introduce the design of the meta-atom followed by a description of the overall architecture of the whole metasurface.

3.1 Design Goal and Challenges

Design goals. To support diverse IoT devices in complex indoor environments, we have the following two design goals for RF-Bouncer's metasurface:

Goal 1: Concurrent dual-band communication. The metasurface must support concurrent wireless communication over two widely separated frequency bands, *e.g.,* 2.4 GHz and 5 GHz in our current implementation.

Goal 2: Dual-band programmability. The metasurface should have dual-band programmability to facilitate concurrent beamforming for communication at two frequency bands.

Design challenges. To achieve of design goals, we also face the following design challenges:

Challenge 1: Discrepancy in electric length. To maximize the communication efficiency, the electric length (physical size) of the meta-atoms depends on the operating frequency, *i.e.,* the electric length should be half of the signal wavelength. There exists a large discrepancy in the electric length of two widely separated bands, for example, the electric length is 6 cm and 3 cm for signals at frequency 2.4 GHz and 5 GHz, respectively. Therefore, it is difficult to fabricate hardware with a fixed physical dimension but multiple resonance frequencies.

Challenge 2: Enabling programmability. Empowering the meta-atoms with programmability without affecting the reflection efficiency is the second challenge.

3.2 Design of Meta-Atoms

In this section, we first introduce the hardware architecture of the meta-atoms followed by the description of the programmability of meta-atoms.

3.2.1 Dual-band Meta-Atoms

The basic structure. We propose to build our meta-atom based on the *metal-backed patch square structure*, which consists of three tightly connected layers: a metallic square patch on the top, a dielectric cuboid in the middle, and a metallic sheet at the bottom, just as shown in Figure 2. According to the cavity model theory [19], the resonant frequency of such a patch structure is given as:

$$f = \frac{c}{2\sqrt{\varepsilon_{re}}} \cdot \frac{1}{l_e} \tag{1}$$

where c is the free-space speed of light. The parameter ε_{re} is effective dielectric constant of the dielectric cuboid, which is

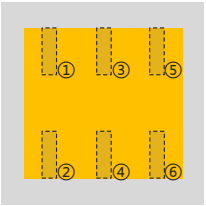


Figure 4: Different locations of slots in the meta-atom.

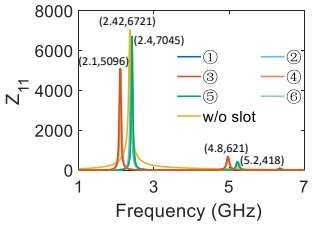


Figure 5: Surface impedance under different single slots.

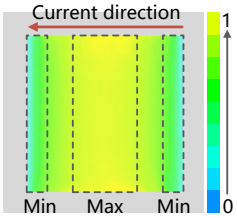


Figure 6: Current distribution of the incipient patch.

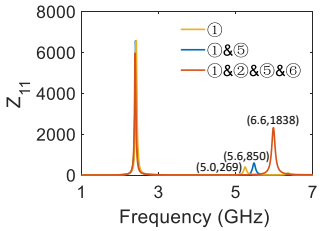


Figure 7: Surface impedance with different number of slots.

given as:

$$\varepsilon_{re} = \frac{\varepsilon_r + 1}{2} + \frac{\varepsilon_r - 1}{2}\left(1 + \frac{12h}{w}\right)^{-\frac{1}{2}} \tag{2}$$

where ε_r is the fundamental dielectric constant of the material that makes up the dielectric cuboid, w is the width of the patch, and h is the height of the dielectric cuboid. The parameter l_e is the effective length of the patch, which is given as:

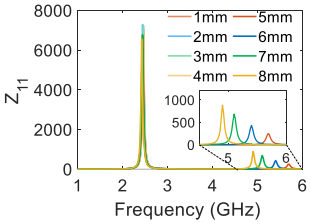
$$l_e = l + 0.824h \cdot \frac{(\varepsilon_{re} + 0.3)(\frac{w}{h} + 0.264)}{(\varepsilon_{re} - 0.258)(\frac{w}{h} + 0.8)} \tag{3}$$

where l is the length of the patch. We know from Eqn. 1, 2, and 3 that the resonate frequency of the structure is determined by the length l, the width w of the patch, and the height h of the dielectric cuboid.

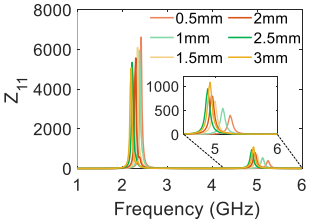
To embed more meta-atoms within one metasurface, we prefer area-efficient design, *i.e.*, smaller width w and length l. We leverage High Frequency Structure Simulator (HFSS) to conduct comprehensive simulations to quantitatively examine the relationship between the resonant frequency and the physical dimension of the meta-atoms. From the result in Figure 3 (a), we observe that, for a fixed dielectric cuboid height h, decreasing the w and l results in increased resonant frequency. Therefore, to maintain the resonant frequency at 2.4 GHz, an area-efficient meta-atom inevitably leads to a thick dielectric cuboid. To balance the size of meta-atom and thickness of the metasurfaces, we conduct extensive off-line simulations and chose the combination of $w = 17.5mm$, $l = 17.5mm$ and $h = 6.8mm$ for our meta-atom. The final result is illustrated in Figure 3 (b), from which we see that the resonance frequency is indeed at 2.4 GHz.

Dual-band operation. To empower the meta-atom with a dual-band capability, we propose to fine-tune the metal-backed patch square structure to generate a second resonant frequency at 5 GHz while keeping the first resonant frequency at 2.4 GHz. Inspired by the theory of slot antenna [5], we propose to etch slots on the patch to generate additional resonant frequency, since the slots on the patch would change the path of the stimulated current and thus the resonant frequency. The final patch structure and thus the resonant frequency depends on the location and the number of slots we etch to the patch.

To study the relationship between the location of the slot and the resonant frequency, we pick six candidate slot positions on the patch, as shown in Figure 4 and leverage HFSS



(a) The length of slot.



(b) The width of slot.

Figure 8: The impact of different slot lengths and widths.

simulation to calculate the resonant frequency. We plot the simulation results in Figure 5, from which we observe that the slots located at the edge of the patch, *i.e.*, the slots ①, ②, ⑤, and ⑥, have minimum impact on the first resonant frequency, but indeed generate the second resonant frequency. The slots at the center, *i.e.*, the slots ③ and ④, however, significantly change the first resonant frequency (shifting it from 2.4 GHz to 2.1 GHz). To explain the rational behind such a phenomenon, we plot the current distribution of the original frequency (2.4 GHz) is shown in Figure 6. We see that the distribution is highly unbalanced: the current at the edge is much weaker than the current at the center of the patch. According to [21], narrow slots located close to the current minima have a minor perturbation to the original resonant frequency. Consequently, we should etch the slots at the edge of the patch to maintain the first resonant frequency at 2.4 GHz.

Even though a new resonant frequency is successfully excited, the reflected signal by the meta-atom is weak at the new resonant frequency, since the impedance between the meta-atom and the free space is close to each other. Specifically, the reflection coefficient Γ of an antenna measures the portion of re-radiated signal, whose value is given as $\Gamma = \frac{Z_{11} - Z_0}{Z_{11} + Z_0}$, where $Z_0 = 120\pi\Omega$ is the impedance of free space. We can see that a larger difference between Z_{11} and Z_0 means more power of the incident signal re-radiates. To obtain more energy from reflective signal, we propose to increase the surface impedance and thus increase the impedance difference between Z_{11} and Z_0. Our solution is to etch multiple slots at locations with small current to form an antenna array. Figure 7 depicts the Z_{11} of the meta-atom, with the number of etched slots varying from one to four. We see that the surface impedance indeed increases with the number of slot increases, but the second resonance frequency also diverges from the desired 5 GHz.

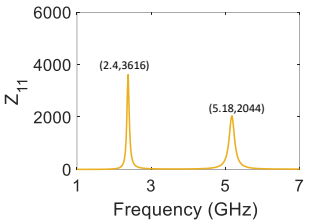


Figure 9: Surface impedance of final slot size.

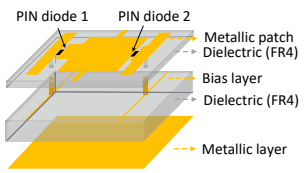


Figure 10: Modified meta-atom structure.

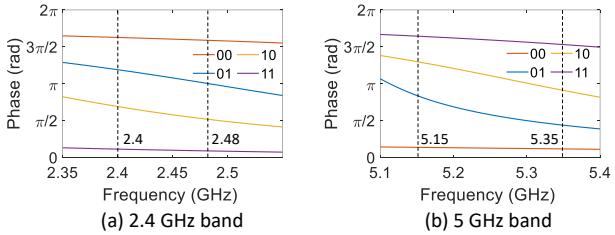


Figure 11: Reflection phase of different frequency bands.

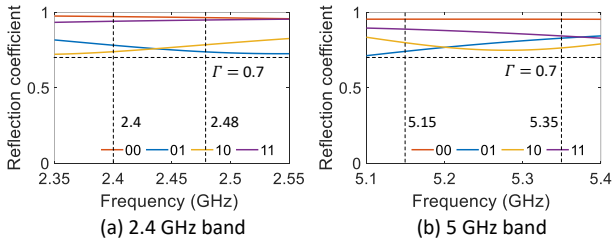


Figure 12: Reflection coefficient of 2.4 GHz and 5 GHz.

We, therefore, further fine-tune the physical dimensions of the slots to shift the second resonant frequency back to 5 GHz. We use HFSS to calculate the impedance of the meta-atom with varying slot length l_s and width w_s, and plot the results in Figure 8. We see that the increase of both l_s and w_s leads to a decrease in second resonant frequency. We choose the combination of $l_s = 7mm$ and $w_s = 0.5mm$ as our final solution. The final impedance of the meta-atom is plotted in Figure 9, from which we see that our optimized meta-atom not only shifts the resonant frequency back but also maximizes the reflection efficiency at two operating frequency bands.

3.2.2 Empowering Programmability for Meta-Atoms

To empower the meta-atom with phase-shifting capability, our basic solution is to embed tunable electronic components into the metallic patch. By programming the state of the electronic components, we change the surface impedance of meta-atom and thus the introduced phase shifts. Specifically, we select PIN diodes as our basic tunable electronic component. We select PIN diodes over varactors because PIN diode only requires two different DC voltage levels rather than precise and continuous voltage values, significantly reducing the design complexity and insertion loss [14]. We etch a rectangle slot under "U" slots and embed two PIN diodes into each slot, just as shown in Figure 10 (Please refer to Appendix A for the detailed design). By controlling the DC voltage, we obtain four

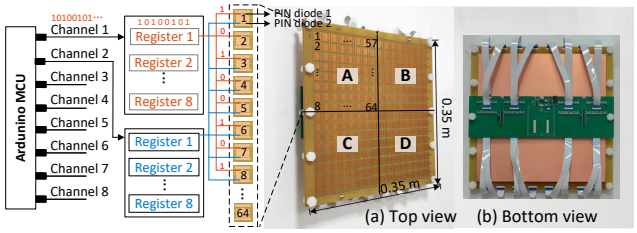


Figure 13: The control architecture of the RF-Bouncer.

stages for each meta-atom, resulting in four phase shifts. We employ HFSS to simulate the phase and reflection coefficient of each stage, and plot the results in Figure 11. We see that the phase difference between each state is about $\pi/2$ at 2.4 GHz and 5 GHz frequency bands. In addition, from Figure 12, we find that the reflection coefficient is stably higher than 0.7 in each stage at two separated frequency bands, implying each state has small impact on the power of the reflective signal. Thus, we can use the final meta-atom as a 2-bit phase shifter.

3.3 Metasurface by Assembling Meta-Atoms

RF-Bouncer's metasurface is designed by assembling multiple optimized meta-atoms, We build a prototype of RF-Bouncer's metasurface that consists of 16×16 meta-atoms. All the meta-atoms are evenly distributed inside an area of $0.35 \times 0.35m^2$, with a distance of $19.5mm$ between adjacent meta-atoms, as shown in Figure 13. To reconfigure the PIN diode states of each meta-atom, we embed a bias layer to transmit DC bias voltage to each PIN diode (SMP1340-040LF PIN diodes [1]).

The controller. To configure the whole metasurface, we design a control circuit module consisting of a Arduino DUE controller and 64 SN74LV595 shift registers to provide different DC voltages (0 V or 5 V) for each meta-atom. Specifically, we divide the entire MTS board into 4 zones, as shown in Figure 13. For each zone, we use two channels in the Arduino MCU to transmit a data stream with 128 bits to control 128 PIN diodes. Due to limited GPIO pins, each channel connects 8 registers to store 64 bits. Once the enabled port is triggered, each resister transmits 8 different DC voltages to respectively control 8 PIN diodes in each meta-atom. Via the above set up, the controller is now able to independently configure the state of each meta-atom's PIN diode. In our system, RF-Bouncer's power consumption is only at the level of mW since the metasurface itself does not emit any power.

4 Beamfomring Through RF-Bouncer

4.1 Problem Formulation

RF-Bouncer supports dual-band beamforming in 3-D space. Given the angle $\theta_i = (\alpha_i, \beta_i)$ of the incident signal, by applying appropriate phase shift $\gamma_{m,n}$ on a matrix of $M \times N$

meta-atoms, RF-Bouncer's metasurface beamforms the incident signal towards arbitrary angle $\theta_d = (\alpha_d, \beta_d)$ in the 3-D space, where α and β represent the azimuth and elevation angle, respectively, just as shown in Figure 14. RF-Bouncer has the following two working modes:

Single-band mode. In this mode, RF-Bouncer focuses on finding the phase shift $\gamma_{m,n}$ of every meta-atom that enables the metasurface beamforming the incident signal at a single frequency band (either 2.4 GHz or 5 GHz) towards an arbitrary angle in 3-D space. RF-Bouncer works in this mode when there only exists wireless communication over a single frequency band.

Dual-band mode. In this mode, RF-Bouncer must finds the optimal phase shift $\gamma_{m,n}$ for every meta-atom so that the metasurface simultaneously beamforms the wireless signals at 2.4 GHz and 5 GHz with incident angle $\theta_i^{2.4G}$ and θ_i^{5G}, towards the departure angle of $\theta_d^{2.4G}$ and θ_d^{5G}, respectively. RF-Bouncer works in this mode if there exist concurrent dual-band transmissions.

4.2 Single Band Beamforming

For the purpose of illustration, we begin the introduction of RF-Bouncer's single-band beamforming algorithm with a simple case where we beamforming in 2-D space using a metasurface consists of two meta-atoms. We then generalize the algorithm to beamforming in 3-D space with a metasurface consisting of a matrix of $M \times N$ equally spaced meta-atoms.

A two-element linear array in 2-D space. We use the example of a two-element linear array to illustrate our beamforming algorithm. As shown in Figure 15 (a), the signal travels different distances before reaching two meta-atoms of the array, resulting in phase differences. Supposing the phase of signal received by the first meta-atom is 0, then the phase vector induced by the incident path is given as:

$$\phi^I(\theta_i, f_c) = \frac{2\pi f_c}{c} \cdot d \begin{bmatrix} 0 \\ \cos\theta_i \end{bmatrix} \tag{4}$$

where f_c is the central frequency of the wireless signal and d is the distance between two meta-atoms. Similarly, the signal departure also results in phase difference. The phase vector induced by the departure path is given as:

$$\phi^T(\theta_d, f_c) = \frac{2\pi f_c}{c} \cdot d \begin{bmatrix} \cos(\pi - \theta_d) \\ 0 \end{bmatrix} \tag{5}$$

The meta-atoms shift the signal by $\gamma = [\gamma_1, \gamma_2]^T$ before reflecting the signal, just as shown in Figure 15 (b). Therefore, the phase of the signals reflected by two meta-atoms along the wavefront at the departure angle θ_d is given as:

$$\phi(\theta_i, \theta_d, f_c, \gamma) = \phi^I + \gamma + \phi^T = \frac{2\pi f_c}{c} \cdot d \begin{bmatrix} \cos(\pi - \theta_d) \\ \cos\theta_i \end{bmatrix} + \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} \tag{6}$$

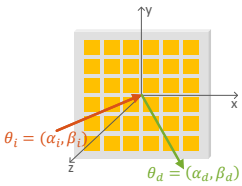


Figure 14: A metasurface in 3D space.

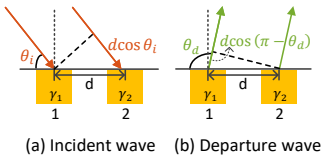
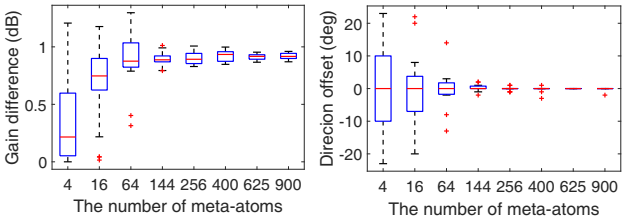


Figure 15: A linear array in 2D space.



(a) Gain difference. (b) Direction offset.

Figure 16: Beamforming gain difference and direction offset between the continuous solution $\gamma_{m,n}$ and the discrete solution $Q_{2-\text{bit}}(\gamma_{m,n})$ after quantization.

Beamforming the signal towards departure angle θ_d requires signals adding constructively, *i.e.*, the phase of signal reflected by all the meta-atoms must be the same. In the above two-element case, the phase shifts γ applied to two meta-atoms should satisfy the following equation:

$$\gamma_2 - \gamma_1 = \frac{2\pi f_c}{c} \cdot d(\cos(\pi - \theta_d) - \cos\theta_i) \tag{7}$$

If we set the phase shift of the first meta-atom to zero, *i.e.*, $\gamma_1 = 0$, the phase shift of the second meta-atom can be directly calculated according to the above equation.

Generalization. We now generalize the above beamforming algorithm to a metasurface embedded with a matrix of $M \times N$ meta-atoms in 3-D space. The phase vector of the incident path in 3-D space becomes:

$$\phi^I(\theta_i, f_c) = \frac{2\pi f_c}{c} \cdot d \begin{bmatrix} 0 & \dots & (N-1)v_i \\ \dots & \dots & \dots \\ (M-1)u_i & \dots & (M-1)u_i + (N-1)v_i \end{bmatrix} \tag{8}$$

where $u_i = \cos\alpha_i \sin\beta_i$ and $v_i = \sin\alpha_i \sin\beta_i$. Similarly, the phase vector of the departure path is given as

$$\phi^T(\theta_i, f_c) = \frac{2\pi f_c}{c} \cdot d \begin{bmatrix} (M-1)u_d + (N-1)v_d & \dots & (N-1)v_d \\ \dots & \dots & \dots \\ 0 & \dots & (M-1)u_d \end{bmatrix} \tag{9}$$

where $u_d = \cos\alpha_d \sin\beta_d$ and $v_d = \sin\alpha_d \sin\beta_d$. Combing Eqn 8 and 9 with Eqn 6, we have enough equations to derive the phase shifts $\gamma_{m,n}$ that we should apply to each meta-atom of the metasurface.

Discrete phase shifts of meta-atoms. The optimal phase shift $\gamma_{m,n}$ we calculate according to above section is continuous. Recall that each meta-atom in the metasurface is essentially a 2-bit phase shifter that only provides four possible phase

shifts: 0, $\frac{\pi}{2}$, π and $\frac{3\pi}{2}$. We apply the following quantization rule to find the discrete solution of phase shifts $\gamma_{m,n}$:

$$Q_{2-\text{bit}}(\gamma_{m,n}) = \begin{cases} 0, & otherwise \\ \pi/2, & if\ \pi/4 \le \gamma_{m,n} < 3\pi/4 \\ \pi, & if\ 3\pi/4 \le \gamma_{m,n} < 5\pi/4 \\ 3\pi/2, & if\ 5\pi/4 \le \gamma_{m,n} < 7\pi/4 \end{cases} \quad (10)$$

Phase quantization brings phase error and inevitably degrades the beamforming performance. To quantitatively investigate the impact of phase quantization, we traverse all the possible combinations of incident angle θ_i and departure angle θ_d and calculate one continuous solution $\gamma_{m,n}$ and one discrete solution $Q_{2-\text{bit}}(\gamma_{m,n})$ for each combination (θ_i, θ_d). We calculate the gap of the beamforming gain and beamforming direction between the continuous solution and the discrete solution and plot the results in Figure 16. We clearly see that with the number of meta-atoms in the metasurface increases, the gap of beamforming gain stabilizes at 1 dB, while the direction offset decreases and eventually gets very close to zero. This result is in line with the prior work [35]. RF-Bouncer's metasurface has $16 \times 16 = 256$ meta-atoms so the degradation of the beamforming performance becomes negligible.

4.3 Dual-band Beamforming

Challenge. For single band beamforming, we are able to calculate the optimal phase shifts $\hat{\gamma}_{m,n}$ of every meta-atom that strictly meeting the requirement of the beamforming: the signal reflected by all meta-atom has exact the same phase so they superimpose constructively at the receiver. Ideally, if each meta-atom is able to compensate the signal with two arbitrary phase shifts at two operating frequency band, then a naive solution would be separately finding the optimal phase shifts for two frequency band, *i.e.*, $\hat{\gamma}^{2.4G}_{m,n}$ for 2.4 GHz band and and $\hat{\gamma}^{5G}_{m,n}$ for 5 GHz band, and then applying the optimal phase shifts to each meta-atom. Our meta-atom, however, only has 2-bit programmability (four states) and thus provides four fixed combination of phase shifts at two frequency band. Specifically, for each state $\eta_{m,n}$, the phase shifts at two frequency band can be derived via a known mapping:

$$\gamma^{2.4G}_{m,n} = P_{2.4G}(\eta_{m,n})$$
$$\gamma^{5G}_{m,n} = P_{5G}(\eta_{m,n}) \quad (11)$$

We list the mapping between the state of meta-atom and the phase shifts introduced by the meta-atom at that state at both 2.4 GHz and 5 GHz band in Table 1.

Due to each meta-atom's limited phase combinations at two frequency bands, it is impossible to simultaneously implement the optimal phase shifts $\hat{\gamma}^{2.4G}_{m,n}$ and $\hat{\gamma}^{5G}_{m,n}$ on our metasurface. Consequently, it is also impossible to find analytical solutions that strictly meet the phase requirement of the beamforming. Instead, we turn to search for the optimal combination of states $\eta_{m,n}$ of meta-atom that maximize the total gain of the main lobes of the metasurface's beamforming patterns at two frequency bands.

$\eta_{m,n}$	$P_{2.4G}$	P_{5G}	$\eta_{m,n}$	$P_{2.4G}$	P_{5G}
00	$3\pi/2$	0	10	$\pi/2$	π
01	π	$\pi/2$	11	0	$3\pi/2$

Table 1: The mapping between the state of meta-atom and the phases shift the meta-atom at that state introduces to signals with central frequency of 2.4 GHz and 5 GHz.

Dual-band link optimization. Given the incident angle $\theta^{2.4G}_i$, the strength of the 2.4 GHz signal reflected by the metasurface along the departure angle $\theta^{2.4G}_d$ is given as:

$$S_{2.4G}(\theta^{2.4G}_i, \theta^{2.4G}_d, \eta) = a_{2.4G}(\theta^{2.4G}_d) \sum_1^M \sum_1^N e^{j\varphi\left(\theta^{2.4G}_i, \theta^{2.4G}_d, f_{2.4G}, \gamma\right)}$$
$$= a_{2.4G}(\theta^{2.4G}_d) \sum_1^M \sum_1^N e^{j\varphi\left(\theta^{2.4G}_i, \theta^{2.4G}_d, f_{2.4G}, P_{2.4G}(\eta_{m,n})\right)} \quad (12)$$

where $a_{2.4G}(\theta^{2.4G}_d)$ represents the amplitude of 2.4 GHz signal reflected by each meta-atom along direction $\theta^{2.4G}_d$, whose value is identical across all identical meta-atoms. Similarly, the strength of 5 GHz signal reflected by the metasurface can be represented as:

$$S_{5G}(\theta^{5G}_i, \theta^{5G}_d, \eta) = a_{5G}(\theta^{5G}_d) \sum_1^M \sum_1^N e^{j\phi\left(\theta^{5G}_i, \theta^{5G}_d, f_{5G}, P_{5G}(\eta_{m,n})\right)}. \quad (13)$$

Our goal is to search for the optimal meta-atom states η^* that maximizes the total signal strength along direction $\theta^{2.4G}_d$ and θ^{5G}_d, given incident signal angle $\theta^{2.4G}_i$ and θ^{5G}_i:

$$\eta^* = \arg\max_{\eta} \left(\left| S_{2.4G}(\theta^{2.4G}_i, \theta^{2.4G}_d, \eta) \right| + \left| S_{5G}(\theta^{5G}_i, \theta^{5G}_d, \eta) \right| \right). \quad (14)$$

To prevent over-optimizing single band and thus guarantee the fairness between two bands, we further adjust our objective function to:

$$\eta^* = \arg\min_{\eta} \left(\left(\left| S^*_{2.4G} \right| - \left| S_{2.4G}(\theta^{2.4G}_i, \theta^{2.4G}_d, \eta) \right| \right) + \left(\left| S^*_{5G} \right| - \left| S_{5G}(\theta^{5G}_i, \theta^{5G}_d, \eta) \right| \right) \right) \quad (15)$$

where $S^*_{2.4G}$ and S^*_{5G} are the theoretical maximum signal strength achieved when we single-band beamform on 2.4 GHz and 5 GHz band using continuous phase shifters, respectively. We employ genetic algorithm (GA) algorithm [33] to solve our optimization problem described in Eqn 15. To speed up the search, instead of generating a random initial population, we use the coding patterns optimized for each single frequency band as a set of initial chromosomes in the initial population of the GA algorithm.

We conduct an experiment to verify the effectiveness of the proposed dual-band links optimization algorithm. In this

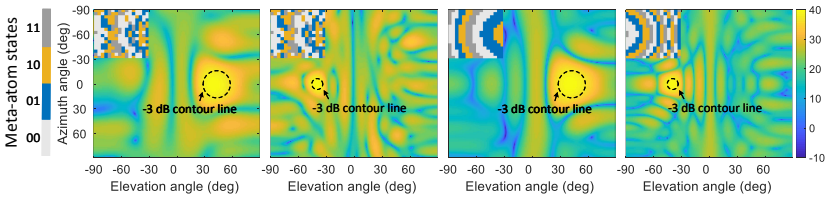


Figure 17: The beam patterns of single-band links and dual-band links, respectively.

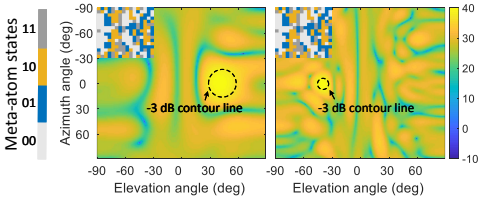


Figure 18: The beam patterns when changing negative meta-atoms.

experiment, we set the incident angle of 2.4 GHz and 5 GHz signal to $(\alpha_i^{2.4G}=0°, \beta_i^{2.4G}=0°)$ and $(\alpha_i^{5G}=0°, \beta_i^{5G}=0°)$, respectively. The desired beamforming direction of 2.4 GHz and 5 GHz signal are set to $(\alpha_d^{2.4G}=0°, \beta_d^{2.4G}=40°)$ and $(\alpha_d^{5G}=0°, \beta_d^{5G}=-40°)$, respectively. After applying the optimal state $\eta_{m,n}^*$ to each meta-atom, the beam pattern of the whole metasurface is plotted in Figure 17, from which we could see that: 1) the two mainlobes obtained by dual-band beamforming well match the desired angles; 2) the sidelobe levels are much lower than the mainlobe levels. In addition, the mainlobe beamwidth of the 2.4 GHz band is wider than the 5 GHz band. The reason is that the size of the meta-atom is more suitable for 5 GHz band, but this issue can be easily solved by increasing the number of meta-atoms to generate a narrow beam of mainlobe [38]. 3) dual-band beamforming has a lower gain about 3 dB than single-band beamforming, while the -3 dB beamwidth is only slightly wider than single-band beamforming (Figure 17 (c) and (d)).

4.4 Harnessing the Ambient Multipath

We observe that there exists some meta-atoms that contribute negligible power or even have negative impact on the main lobe, but significantly affect the distribution of the side lobes, especially when the metasurface is configured for dual-band operation. To demonstrate such a phenomenon, we change the states of a small group of meta-atoms in Figure 17 (a) and (b), and plot the 3-D beam patterns of the new metasurface configuration in Figure 18. Comparing the beam patterns in these two figures, we see that the direction and gain of main lobes still retain, but the side lobes change dramatically. The signal of the side lobes does not travel directly towards the receiver, but may still reach the receiver after being reflected by diverse objects in the propagation environment. We propose to further improve the signal strength by adjusting the pattern of side lobes of the metasurface.

The key challenge we face is to select the group of meta-atoms that mainly affects the side lobes. Since the target meta-atoms have negligible or even negative impact on the main lobe, the phase of the signal reflected by the target meta-atoms must be misaligned (difference larger than $\pi/2$) with the phase of the main lobes. Without loss of generality, we denote the desired beamforming directions at 2.4 GHz and 5 GHz bands are $(\alpha_{2.4G}, \beta_{2.4G})$ and $(\alpha_{5G}, \beta_{5G})$, respectively. Then, the phase of mainlobe is given as:

$$\mathrm{PM}_{2.4G} = \angle \sum_{m=1}^{M}\sum_{n=1}^{N} e^{j\phi(\theta_i^{2.4G}, \theta_d^{2.4G}, f_{2.4G}, P_{2.4G}(\eta_{m,n}^*))} \tag{16}$$

$$\mathrm{PM}_{5G} = \angle \sum_{m=1}^{M}\sum_{n=1}^{N} e^{j\phi(\theta_i^{5G}, \theta_d^{5G}, f_{5G}, P_{5G}(\eta_{m,n}^*))} \tag{17}$$

where $\eta_{m,n}^*$ is the optimal states of meta-atoms we calculated according to Eqn 15. Then, we find separate sets of target meta-atoms at two frequency bands, respectively:

$$TU_{2.4G} = \left\{ \{m,n\} \,\middle|\, \left|\mathrm{PM}_{2.4G} - \phi(\theta_i^{2.4G}, \theta_d^{2.4G}, f_{2.4G}, \gamma_{m,n})\right| \geq \frac{\pi}{2} \right\}, \tag{18}$$

$$TU_{5G} = \left\{ \{m,n\} \,\middle|\, \left|\mathrm{PM}_{5G} - \phi(\theta_i^{5G}, \theta_d^{5G}, f_{5G}, \gamma_{m,n})\right| \geq \frac{\pi}{2} \right\}, \tag{19}$$

where m and n vary from 1 to 16 in our system. Finally, we select the intersection of two sets of target meta-atoms as the final solution:

$$TU = TU_{2.4} \cap TU_5. \tag{20}$$

The intersection TU includes meta-atoms that have negligible or even negative impact for 2.4 GHz and 5 GHz band simultaneously, so we are safe to change the states of the meta-atoms in TU to adjust the side lobes while at the same time guarantees minimum impact on main lobes of two band. We iterate all possible combinations of state and choose the one that provides best signal quality. It is worth noting that when the number of variable meta-atoms is large, it could take a long time for exhaustive search. To reduce the search time, one potential solution is to divide the whole metasurface into several parts. For each part, the variable meta-atoms change their state in the same way. Therefore, the number of exhaustive search will be reduced to a small number. For example, we divide the whole metasurface into 4 parts. Therefore, all possible combinations of state will be reduced to 512 (i.e., 4^4). Assuming Wi-Fi packets are collected at a rate of 1,000 packets per second, the search time will be 0.5 seconds.

4.5 Beam Alignment

To accurately beam the reflected signal towards the receiver, we need to know the signal incident angle θ_i and departure

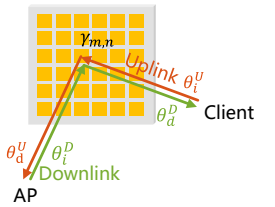


Figure 19: Illustration of uplink and downlink.

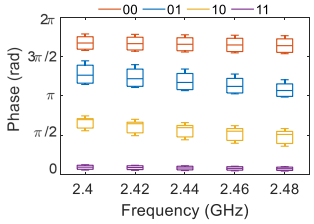


Figure 20: Phase offset under different incident angles.

angle θ_d. We observe that in a typical Wi-Fi system, the access point is static for most of the cases. Therefore, the signal incident and departure angle is fixed and known for downlink and uplink Wi-Fi communication, respectively (Figure 19). We propose to search for the unknown angle, *i.e.*, incident angle for uplink and departure angle for downlink.

Beam search for downlink. For downlink communication, we need to search for the departure angle θ_d. In n-th round of beam search, the metasurface configures its meta-atoms to point the main lobe towards a specific angle θ_d^n. After receiving one packet under such a configuration, the receiver embeds one bit inside its ACK to inform the metasurface whether the received signal quality has increased or not comparing with the previous configuration, *i.e.*, bit 1 represents increase and bit 0 means decrease. We equip the metasurface with a Wi-Fi receiver to overhear the ACK. After iterating all possible departure angles, we select the θ_d that provides the highest received signal quality as our results.

To speed up the searching process, we implement a two-stage searching algorithm. In the first stage, we search with a relatively large step size of 20°. After obtaining the rough direction, we then search with a small step size of 5° to fine-tune the results. A step size of 5° is fine-grain enough since the beamforming gain only decreases less than 1 dB at directions that are 2.5° apart from the beamforming direction, according to both our simulation and empirical results.

Bidirectional communication. We observe that, due to reciprocity, we only need to perform beam search in one direction. First, according to *channel reciprocity*, the phase shifts that should be applied to each meta-atom for beamforming is the same even when we swap the value of the incident angle θ_i and departure angle θ_d. As shown in Figure 19, the signal travels exactly the same distance no matter the AP or the client is the sender, introducing the same amount of phase variations. Therefore, the phase shifts required to meet the phase requirement of beamforming is also the same.

Second, the optimized meta-atom introduces the same amount of phase shifts to the signal, regardless of the signal incident angle. To verify that, we use HFSS to calculate the phase shifts introduced by meta-atom by varying the incident angle, operating frequency and state of meta-atom. We plot the distribution of phase shifts in Figure 20, from which we see that the phase variations introduced by the meta-atom is stable. According to the above analysis, the configuration of the metasurface used for beamforming in one direction also works in the opposite direction. Such an observation significantly accelerates convergence of the beam alignment algorithm, especially when the client moves.

5 Evaluation

Experimental setup. For controlled experiments, we use USRP N210 software-defined radios with a UBX-40 daughterboard as the radio transmitter (Tx) and receiver (Rx). We conduct extensive experiments in three indoor environments to evaluate the performance of RF-Bouncer: a 140 m^2 duplex with two-bedroom, a 160 m^2 apartment and a spacious corridor environment with corner. In the default experimental setting, 2.44 GHz and 5.25 GHz are selected as the operating frequency of the 2.4 GHz band and 5 GHz band, respectively. The Tx is deployed in the normal direction of the metasurface.

5.1 Hardware Verification

Dual-band beamforming verification. This experiment compares beamforming results between single-band and dual-band coding patterns. We configure the metasurface using single-band coding patterns of 2.4 GHz and 5 GHz, and optimization dual-band coding patterns, respectively. The Rx moves along a semicircle (3 m radius) from -90° to 90° with a step of 10°, while the Tx stays in the center. The Tx-metasurface distance sets as 0.5 m. Figure 21 demonstrates that both single-band and dual-band coding pattern can achieve effective beamforming results, but dual-band beamforming results come at the cost of slight decreases of signal strength or slight shift of the direction on the mainlobe.

The performance of beamforming in dual-band. To evaluate the beamforming performance of dual-band, we keep the Tx-metasurface distance sets as 0.5 m. We default $\alpha = 0°$ and only vary β in the following experiments. We move the Rx along a semicircle (3 m radius) from $-90°$ to 90° with a step of 10°. The results are shown in Figure 22, we can clearly see that the effective beamforming ranges of 2.4 GHz and 5 GHz are both $[-60°, 60°]$. Although the $-3dB$ beamwidth becomes wider and beamforming gain becomes lower when the beamforming direction is towards the boundary, the correct directionality is retained. It is worth noting that the beamforming gain of 0° is slightly decreased since the Tx blocks the link between metasurface and Rx. In addition, the performance will be significantly dropped when the beamforming direction is over the boundary. In conclusion, the effective field-of-view (FoV) of beamforming is $[-60°, 60°]$.

The effective incident angles. We conduct experiments in the corridor to explore the effective range of incident angles. For the convenience of expression, we only mention β in the following and default $\alpha = 0°$. To determine the range, we first vary incident angles by changing the direction of Tx from

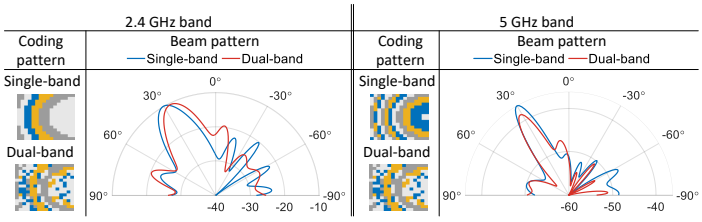
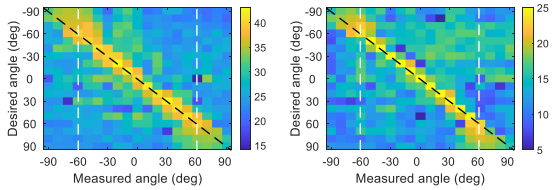
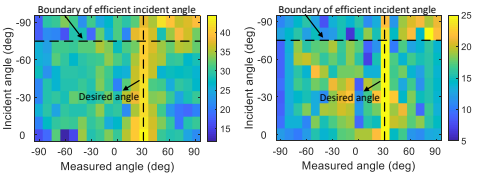


Figure 21: Results of single and dual band coding patterns.



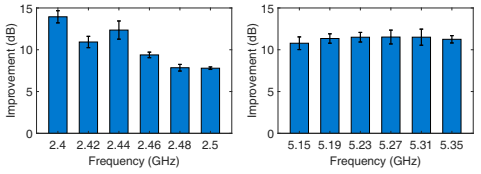
(a) 2.4 GHz band. (b) 5 GHz band.

Figure 22: The results of beam steering from $-90°$ to $90°$.



(a) 2.4 GHz band. (b) 5 GHz band.

Figure 23: The performance under different incident angles.



(a) 2.4 GHz band. (b) 5 GHz band.

Figure 24: SNR improvement across operating frequencies.

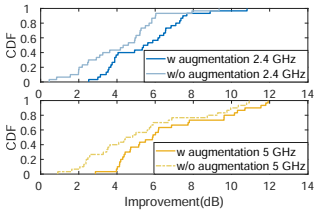
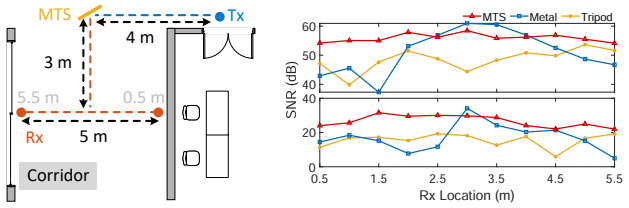


Figure 25: Results of multipath augmentation scheme.



(a) Deployment layout. (b) Power improvement.

Figure 26: The performance of different reflectors.

$-90°$ to $0°$ with a step of $10°$ and set the Tx-metasurface distance as 0.5 m. Then, we move Rx along a semicircle (3 m radius) from $-90°$ to $90°$ with a step of $10°$ to obtain different beam patterns under different incident angles. The desired beamforming direction is set towards $30°$. The results are shown in Figure 23, from which we see that RF-Bouncer can achieve beamforming effectively in the desired direction when the incident angle varies from $-70°$ to $0°$. However, the beamforming gain and direction of the mainlobe can not be guaranteed due to the incident wave being almost parallel (i.e., from $-90°$ to $-80°$) to the metasurface. In addition, because of the symmetry of metasurface, the same experimental results will appear in $[0°, 70°]$. To summarize, RF-Bouncer can work well as long as Tx is located in $[-70°, 70°]$.

Performance across different spectrums. In this experiment, we validate the performance of RF-Bouncer across different operating frequency bands. The distance of Tx-metasurface is 0.5 m, and the direction of the incident wave is perpendicular to the metasurface. The distance between Rx and metasurface is set to 3 m. The direction of the emergent wave focuses on $(30°, 0°)$ and Rx is located in the same direction. One case is from 2.4 GHz to 2.5 GHz with a step of 0.02 GHz. Figure 24(a) shows the SNR can be increased by over 7.79 dB (up to 13.94 dB). Another case is from 5.15 GHz to 5.35 GHz with a step of 0.04 GHz. Figure 24(b) shows the SNR can be improved by over 10.78 dB (up to 11.5 dB). Therefore, RF-Bouncer can be applied to ubiquitous commercial IoT devices working in 2.4 GHz and 5 GHz bands.

5.2 Communication Performance

Multipath augmentation verification. To examine the effectiveness of multipath augmentation described in Section 4.4, we conduct experiments in a representative 3D scenario (Figure 28 (a)). Specifically, we fix the location of the transmitter and randomly move the receiver to 30 locations. Then, for each location, we respectively collect the measurements with/without multipath augmentation at 2.4 GHz and 5 GHz. Figure 25 plots the CDF of signal strength improvement at two frequency bands. We can see that the median improvements with/without multipath augmentation at 2.4 GHz and 5 GHz are respectively 4.32 dB, 5.38 dB and 4.43 dB, 5.89 dB. These results demonstrate that our multipath augmentation scheme can effectively harness the ambient multipath to improve the link SNR.

The performance of different reflectors. By placing RF-Bouncer at a corridor intersection, the blind spot around the corner can be illuminated. We conduct an experiment in a spacious corridor environment, as shown in Figure 26(a). The metasurface is placed at the corner, receiving signals from $(-45°, 0°)$, and reflecting a fan beam from $(-85°, 0°)$ to $(5°, 0°)$. The Tx is 4 m away from the metasurface; whereas Rx is 3 m away moved across a 5 m distance. Figure 26(b) compares the RSS with metasurface, a metal plane reflector with the same size as the metasurface, and the tripod without reflectors. While the metal plane creates a stronger mainlobe towards the specular direction, the RSS drastically drops as the Rx is moved to anomalous directions. In contrast, RF-Bouncer reshapes the incidental beam to cover a wider angular range and thus a larger region around the corner.

Multi-bit beamforming verification. The 2-bit metasurface

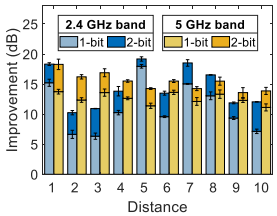
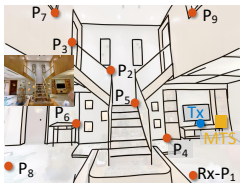
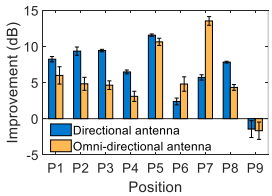


Figure 27: 1-bit V.S. 2-bit.

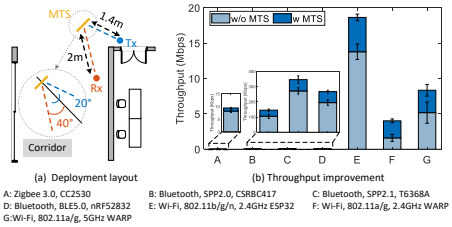


(a) 3D scenario Layout

Figure 28: Experimental results under 3D scenario.



(b) Power Improvement

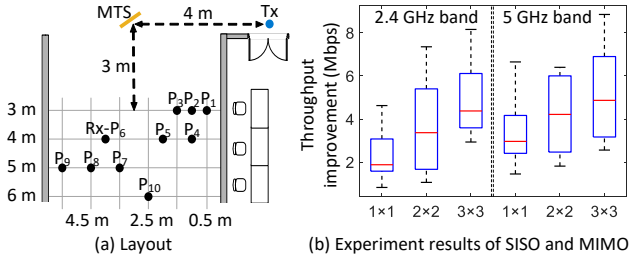


(a) Deployment layout

(b) Throughput improvement

A: Zigbee 3.0, CC2530
B: Bluetooth, SPP2.0, CSRBC417
C: Bluetooth, SPP2.1, T6368A
D: Bluetooth, BLE5.0, nRF52832
E: Wi-Fi, 802.11b/g/n, 2.4GHz ESP32
F: Wi-Fi, 802.11a/g, 2.4GHz WARP
G:Wi-Fi, 802.11a/g, 5GHz WARP

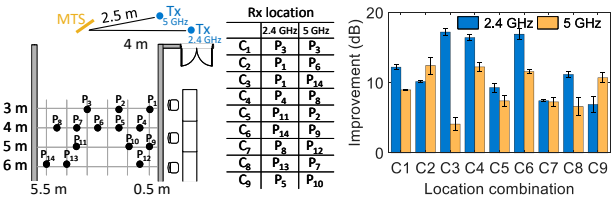
Figure 29: Experimental results of throughput improvement across different IoT devices.



(a) Layout

(b) Experiment results of SISO and MIMO

Figure 30: Throughput improvement of SISO and MIMO.



(a) Layout.

(b) SNR improvement.

Figure 31: Performance under concurrent transmissions.

designed by RF-Bouncer can be backward compatible to realize the 1-bit programmable function. In this experiment, we compare the performance between 2-bit, 1-bit, and without metasurface (referred to w/o MTS). The Tx-metasurface distance is set to 0.5 m and Rx is located at (30°,0°) of metasurface. We then vary the Rx-metasurface distance from 1 m to 10 m by the step of 1 m to measure the SNR improvement. Figure 27 demonstrates that compared w/o MTS, 1-bit and 2-bit programmable functions both can significantly enhance the SNR, but the improvement of 2-bit is larger than 1-bit. For example, when compared between 2-bit and 1-bit, the minimum, median, and maximum SNR increase by 1.22 dB, 3.52 dB, and 4.91 dB across the 2.4 GHz band, and 1.29 dB, 2.81 dB, and 4.54 dB across the 5 GHz band, respectively.

Performance under 3D scenario. We test the SNR improvement achieved by RF-Bouncer in a representative 140 m^2 3D scenario(Figure 28(a)). Due to the deployment limitation, we mount the Tx and metasurface on tripods and place them at the same height up the ground. The distance of Tx-metasurface is 0.5 m. Both Rx and Tx work in the 5 GHz band. The Rx is located at 9 different locations. The height of the Rx from the ground varies from 10 cm to 5 m. The elevation angle of Rx varies from $-30°$ to 30° and the azimuth angle of Rx varies from $-30°$ to 40°. We measure the SNR improvement by using directional and omnidirectional antenna, respectively. Figure 28(b)shows the improvement in different channel conditions. Almost all signal strength improvements under different positions are above 2.5 dB and up to 13.5 dB. Furthermore, due to indoor multipath, the improvement of omni-directional antenna at some positions (i.e., P_6 and P_7) in the 3D scenario is higher than directional antenna. In contrast, the improvement of P_9 is negative due to the following reasons: 1) the azimuth angle between P_9 and metasurface exceeds the effective beamforming FoV of metasurface, leading RF-Bouncer can not provide beamforming gain to it; 2) the reinforced concrete between floors blocks the LoS between Tx and Rx, causing most energy of the incident signal reflects to other directions. This issue can be easily solved by deploying multiple metasurfaces.

Throughput across different IoT devices. RF-Bouncer aims to enhance the signal energy of IoT devices in the NLoS scenario. Hence, in this experiment, we test IoT devices (i.e., CC2530, CSRBC417, KT6368A, nRF52832, ESP32, WARPv3) operating different frequency bands (i.e., 2.4 GHz and 5 GHz) with various protocols (i.e., Zigbee 3.0, SPP 2.0, SPP 2.1, BLE 5.0, 802.11 b/g/n, and 802.11 a/g) in a corner NLoS scenario. We use *iperf* to measure TCP throughput for ESP32 and use the WARPLab environment for the WARPv3 boards. The NLoS deployment layout is shown in Figure 29(a). We set the Tx-metasurface distance and Rx-metasurface distance as 1.4 m and 2 m, respectively. The incident angle of metasurface is $(20°,0°)$ and the emerging angle is $(-40°,0°)$. The results with metasurface and without metasurface (referred to w MTS and w/o MTS) are shown in Figure 29(b). We can see that the minimum, median, and maximum throughput gain are 115%, 137%, and 249%, respectively. These results imply that RF-Bouncer is transparent to the working protocols and frequencies.

SISO and MIMO links. We now evaluate the throughput performance in SISO and MIMO communication systems. Specifically, we use two laptops equipped with AR9580 wireless cards as the transmitter and receiver, each of them has three antennas and works on the 801.11n protocol. Then, we fix the transmitter location and move the receiver to 10 locations, as shown in Figure 30(a). At each location, we respectively change the communication mode, varying from 1×1, 2×2 and 3×3, and use *iperf* toolbox to collect throughput measurements. Figure 30(b) shows that as the number of antennas used in communication system increases, the

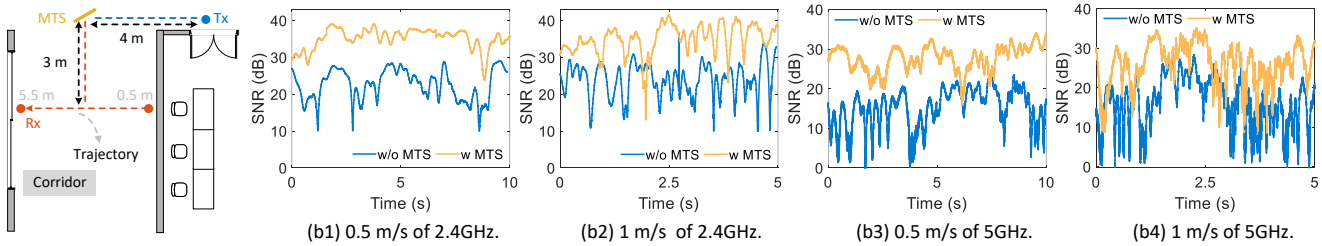


Figure 32: Experimental results of different walking speeds under different frequency bands.

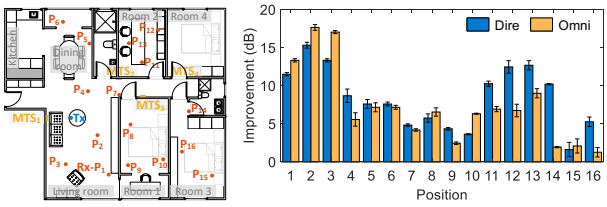


Figure 33: Whole-house coverage under cooperative work.

throughput improvement increases in both frequency bands. For example, RF-Bouncer can achieve a median throughput improvement of 1.88Mbps, 3.37Mbps, 4.38Mbps, and 2.96Mbps, 4.22Mbps, 4.87Mbps for 1×1, 2×2 and 3×3 at 2.4 GHz and 5 GHz, which corresponds to an improvement of up to 50% compared with the baseline in each case.

Concurrent transmissions. In this experiment, we evaluate RF-Bouncer's performance in the presence of concurrent wireless links at 2.4 GHz and 5 GHz. Specifically, we fix two transmitters working at 2.4 GHz and 5 GHz in two different locations, and move two corresponding receivers to nine different location combinations. The detailed deployment is presented in Figure 31(a). In each location combination, the coding pattern of metasurface is obtained based on Sec. 4.3, and we then collect the measurements to calculate the SNR improvement when there is no metasurface. The results in Figure 31(b) shows that RF-Bouncer can simultaneously improve the SNR of two concurrent wireless links. For example, RF-Bouncer can achieve an average SNR improvement of 9.01 dB and 12.08 dB for 2.4 GHz and 5 GHz, respectively. This demonstrates that RF-Bouncer can work well for dual-band concurrent wireless transmissions.

5.3 Performance under Mobility

In this section, we examine the performance of RF-Bouncer in a mobile environment. We place the transmitter at a fixed location and move the receiver along a predefined trajectory with two constant speeds: 0.5 m/s (slow) and 1.0 m/s (normal). In each speed case, RF-Bouncer controller configures the metasurface in real time to accurately beam the reflected signal towards the receiver. The detailed sweep mechanism is referred to Sec. 4.5. Then, we collect the measurements to calculate the SNR during the receiver's movement. Figure 32 shows the real-time SNR measurements with/without metasurface (referred to w MTS and w/o MTS) in 2.4 GHz and 5 GHz. We can see that RF-Bouncer can consistently achieve a SNR improvement compared to the case without metasurface at both different speeds and different frequency bands. These results demonstrate that RF-Bouncer can work well in mobile environments. In addition, we can observe that the performance of RF-Bouncer in slow speed works better than high speed. This is because RF-Bouncer has more time to beam the signal towards the receiver. We thus will explore the high speed scenario of RF-Bouncer in future work.

5.4 Coverage Extension

Whole-house coverage under cooperative work. By placing multiple metasurfaces (MTSs) in the complex whole-house scenario, the signal coverage can be efficiently expanded. We conduct an experiment in a 160 m^2 place with four rooms (Figure 33(a)). Four MTSs are cooperating. MTSs independently control and the working range of each MTS is disjoint. We note that this experiment does not consider how to select a good MTS route in the central control end to achieve good performance. Instead, we manually selected a routing route to perform beamforming for each location. Specifically, the transmitter in this experiment is fixed at one location and four MTSs are also pre-fixed at different locations. The route starts at MTS_1, goes through MTS_2 and MTS_3, and ends at MTS_4. Coverage includes the living room, the dining room, and three rooms. Note that missing areas that are not currently covered - such as the kitchen, room 4, and the bathrooms - can be easily covered by deploying more MTSs in the future. We set the Tx-MTS distance to 1 m and measure the SNR improvement in different locations. RSS from P_1 to P_6 is controlled by MTS_1; from P_7 to P_{10} is controlled by MTS_2; from P_{11} to P_{13} is controlled by MTS_3; and from P_{14} to P_{16} is controlled by MTS_4. The results of using the omnidirectional/directional antenna (referred to Omni and Dire) and with/without the MTS (referred to w MTS and w/o MTS) are shown in Figure 33(b). We can clearly see that 1) the signal coverage can be efficiently expanded by leveraging multiple MTSs collaborative with each other; 2) the SNR is generally improved (above 1.26 dB and up to 17.65 dB) in NLoS envi-

ronments by the MTS, while the enhancements of different distances are different due to multipath; 3) MTS can achieve good performance even without LoS path between MTS and Rx (i.e., through the wall); 4) regardless of the antenna pattern, RF-Bouncer can improve the SNR.

Corner coverage expansion. In this experiment, we show how RF-Bouncer expands the wireless coverage at the corner scenario. As shown in Figure 34, we collect measurements with/without metasurface in both downlink and uplink for each location. From Figure 34, we can see that without metasurface, most locations around the corner have a lower SNR, especially for 5 GHz links, which only have an average SNR of 4.24 dB and 3.67 dB in downlink and uplink, respectively. This is because a signal of higher frequency has more severe path attenuation. In contrast, with the help of RF-Bouncer metasurface, most locations around the corner significantly improved SNR in both downlink and uplink at 2.4 GHz and 5 GHz. For example, RF-Bouncer can achieve an average SNR improvement of 10.30 dB (up to 26.16 dB) and 8.10 dB (up to 24.19 dB) for 2.4 GHz and 5 GHz downlinks, while an average SNR improvement of 8.78 dB (up to 25.27 dB) and 8.54 dB (up to 17.01 dB) for 2.4 GHz and 5 GHz uplinks. These results demonstrate that 1)RF-Bouncer can well expand the wireless coverage at the corner scenario; 2) RF-Bouncer can achieve good performance when Tx and Rx locations vary over a wide range of angles; 3) RF-Bouncer can work well for downlink and uplink simultaneously.

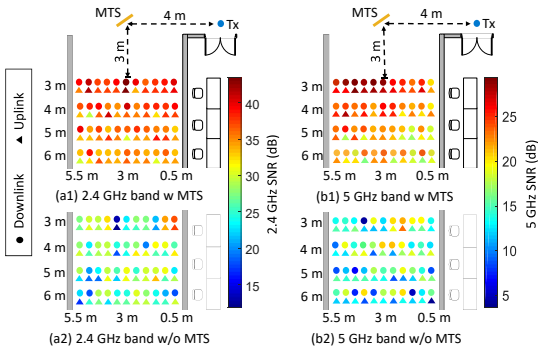


Figure 34: Corner coverage extension.

6 Discussion

Multiple metasurfaces cooperation. Cooperating multiple metasurfaces across rooms can effectively expand the wireless coverage, which is an interesting and challenging direction. In our current implementation, we manually calculate the configuration of each metasurface offline (which cannot guarantee the optimal performance) and send the configuration to each controller. In the future work, we will design an algorithm that can automatically configure the networked metasurfaces. In addition, current metasurface only consists of 256 meta-atoms with a size of $0.35 \times 0.35m^2$, such a small aperture would lead to a wide beam at 2.4 GHz, which results in a worse coverage between different metasurfaces in the case of installing multiple metasurfaces. This is a limitation of our current version. Thus, to avoid this issue, one possible method is to design a larger aperture metasurface to generate a narrow beam of mainlobe.

Operationg frequency. Our current design has a bandwidth of 200 MHz (from 5.15 GHz to 5.35 GHz), which covers 17 WiFi channels at 5 GHz. Since commercial devices go all the way to 5.8 GHz of spectrum, we thus will optimize our meta-atom's design to enlarge the effective working band to cover additional 5 GHz channels in the follow-up work.

Unwanted interference induced by metasurface. Deploying a smart surface to amplify some of the signals could possibly lead to interference, especially when there are multiple concurrent wireless links. But the possibility is small since the metasurface is beamforming the signal towards a specific direction, instead of omnidirectional reflection. Also, if the whole area is covered with metasurfaces, we could minimize interference by coordinating the metasurfaces.

Practicality and scalability of RF-Bouncer. The current version of RF-Bouncer needs to deploy a metasurface for each room, causing a huge cost. Fortunately, its cost can be minimized through mass fabrication. Meanwhile, due to the thin surface nature of metasurface, it can potentially be embedded into the environment (e.g., furniture and walls) to reduce the footprint, promoting its widespread deployment. In addition, since each metasurface has a FoV (i.e., $[-60°, 60°]$ for RF-Bouncer's metasurface), by deploying a metasurface in the public area (e.g., the corridor), we can only use a single metasurface to reflect signal into many rooms, so as to avoid each room requires installing a metasurface.

7 Conclusion

We have designed, fabricated, and validated RF-Bouncer, a 2-bit dual-band reflecting metasurface to expand indoor wireless coverage. By encoding the phase shifting values, RF-Bouncer can simultaneously manipulate electromagnetic waves in two ISM bands. In addition, RF-Bouncer is transparent to protocols, so as to support diverse commercial IoT devices. Field study shows that RF-Bouncer can enable 15.4 dB average signal strength improvement.

Acknowledgment

Thanks the anonymous shepherd and reviewers for their valuable comments. This work is supported in part by National Natural Science Foundation of China under Grants (62272388, 61972316, 62061146001) and the Shaanxi International Science and Technology Cooperation Program (2023-GHZD-04, 2023-GHZD-06).

References

[1] Smp1340-040lf. https://www.skyworksinc.com/Products/Diodes/SMP1340-Series.

[2] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Enabling high-quality untethered virtual reality. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 531–544, 2017.

[3] Haider Ali, Laeeq Riaz, Syed Abdul Mannan Kirmani, Shahid A Khan, and M Farhan Shafique. Dual band-stop reconfigurable (switchable) frequency selective surface for wlan applications at 2.4 and 5 ghz. *AEU-International Journal of Electronics and Communications*, 143:154038, 2022.

[4] Venkat Arun and Hari Balakrishnan. Rfocus: Beamforming using thousands of passive antennas. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1047–1061, 2020.

[5] Constantine A Balanis. *Antenna theory: analysis and design*. John wiley & sons, 2015.

[6] Ertugrul Basar, Marco Di Renzo, Julien De Rosny, Merouane Debbah, Mohamed-Slim Alouini, and Rui Zhang. Wireless communications through reconfigurable intelligent surfaces. *IEEE ACCESS*, 7:116753–116773, 2019.

[7] Dinesh Bharadia and Sachin Katti. Fastforward: Fast and constructive full duplex relays. *ACM SIGCOMM Computer Communication Review*, 44(4):199–210, 2014.

[8] Michael Boyarsky, Timothy Sleasman, Mohammadreza F Imani, Jonah N Gollub, and David R Smith. Electronically steered metasurface antenna. *Scientific reports*, 11(1):1–10, 2021.

[9] Justin Chan, Changxi Zheng, and Xia Zhou. 3d printing your wireless coverage. In *Proceedings of the 2nd International Workshop on Hot Topics in Wireless*, pages 1–5, 2015.

[10] Hou-Tong Chen, Antoinette J Taylor, and Nanfang Yu. A review of metasurfaces: physics and applications. *Reports on progress in physics*, 79(7):076401, 2016.

[11] Kun Woo Cho, Yasaman Ghasempour, and Kyle Jamieson. Towards dual-band reconfigurable metamaterial surfaces for satellite networking. *arXiv preprint arXiv:2206.14939*, 2022.

[12] Kun Woo Cho, Mohammad H Mazaheri, Jeremy Gummeson, Omid Abari, and Kyle Jamieson. mmwall: A reconfigurable metamaterial surface for mmwave networks. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 119–125, 2021.

[13] Manideep Dunna, Chi Zhang, Daniel Sievenpiper, and Dinesh Bharadia. Scattermimo: Enabling virtual mimo with smart surfaces. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 1–14, 2020.

[14] Chao Feng, Xinyi Li, Yangfan Zhang, Xiaojing Wang, Liqiong Chang, Fuwei Wang, Xinyu Zhang, and Xiaojiang Chen. Rflens: metasurface-enabled beamforming for iot communication and sensing. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 587–600, 2021.

[15] Kai Guo, Qun Zheng, Zhiping Yin, and Zhongyi Guo. Generation of mode-reconfigurable and frequency-adjustable oam beams using dynamic reflective metasurface. *IEEE Access*, 8:75523–75529, 2020.

[16] Sihui Han and Kang G Shin. Enhancing wireless performance using reflectors. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[17] Hui-Hsin Hsiao, Cheng Hung Chu, and Din Ping Tsai. Fundamentals and applications of metasurfaces. *Small Methods*, 1(4):1600064, 2017.

[18] Kai-Cheng Hsu, Kate Ching-Ju Lin, and Hung-Yu Wei. Full-duplex delay-and-forward relaying. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 221–230, 2016.

[19] James R James, Peter S Hall, and Colin Wood. *Microstrip antenna: theory and design*, volume 12. Iet, 1986.

[20] Wahab Khawaja, Ozgur Ozdemir, Yavuz Yapici, Fatih Erden, and Ismail Guvenc. Coverage enhancement for nlos mmwave links using passive reflectors. *IEEE Open Journal of the Communications Society*, 1:263–281, 2020.

[21] Girish Kumar and Kamala Prasan Ray. *Broadband microstrip antennas*. Artech house, 2003.

[22] Teng Li, Hongfu Meng, and Wenbin Dou. Design and implementation of dual-frequency dual-polarization slotted waveguide antenna array for ka-band application. *IEEE Antennas and Wireless Propagation Letters*, 13:1317–1320, 2014.

[23] Hai Lin, Wen Yu, Rongxin Tang, Jing Jin, Yumei Wang, Jie Xiong, Yanjie Wu, and Junming Zhao. A dual-band reconfigurable intelligent metasurface with beam steering. *Journal of Physics D: Applied Physics*, 55(24):245002, 2022.

[24] Allen Miu, Hari Balakrishnan, and Can Emre Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *Proceedings of the 11th annual international conference on Mobile computing and networking*, pages 16–30, 2005.

[25] Rohan Murty, Jitendra Padhye, Ranveer Chandra, Alec Wolman, and Brian Zill. Designing high performance enterprise wi-fi networks. In *NSDI*, volume 8, pages 73–88, 2008.

[26] Binh Duong Nguyen and Christian Pichot. Unit-cell loaded with pin diodes for 1-bit linearly polarized reconfigurable transmitarrays. *IEEE Antennas and Wireless Propagation Letters*, 18(1):98–102, 2018.

[27] Zhangyou Peng, Linxiao Li, Miao Wang, Zhonghao Zhang, Qi Liu, Yang Liu, and Ruoran Liu. An effective coverage scheme with passive-reflectors for urban millimeter-wave communication. *IEEE Antennas and Wireless Propagation Letters*, 15:398–401, 2015.

[28] Kun Qian, Lulu Yao, Xinyu Zhang, and Tina Ng. Millimirror: 3d printed reflecting surface for millimeter-wave coverage expansion. In *Proceedings of the 28th Annual International Conference on Mobile Computing and Networking*, 2022.

[29] Hariharan Rahul, Haitham Hassanieh, and Dina Katabi. Sourcesync: A distributed wireless architecture for exploiting sender diversity. *ACM SIGCOMM Computer Communication Review*, 40(4):171–182, 2010.

[30] Marco Di Renzo, Merouane Debbah, Dinh-Thuy Phan-Huy, Alessio Zappone, Mohamed-Slim Alouini, Chau Yuen, Vincenzo Sciancalepore, George C Alexandropoulos, Jakob Hoydis, Haris Gacanin, et al. Smart radio environments empowered by reconfigurable ai metasurfaces: An idea whose time has come. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):1–20, 2019.

[31] Yasir Saifullah, Qinzhuo Chen, Guo-Min Yang, Abu Bakar Waqas, and Feng Xu. Dual-band multi-bit programmable reflective metasurface unit cell: design and experiment. *Optics Express*, 29(2):2658–2668, 2021.

[32] Amin Tayebi, Junyan Tang, Pavel Roy Paladhi, Lalita Udpa, Satish S Udpa, and Edward J Rothwell. Dynamic beam shaping using a dual-band electronically tunable reflectarray antenna. *IEEE Transactions on Antennas and Propagation*, 63(10):4534–4539, 2015.

[33] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[34] Grace R Woo, Pouya Kheradpour, Dawei Shen, and Dina Katabi. Beyond the bits: cooperative packet recovery using physical layer information. In *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, pages 147–158, 2007.

[35] Qingqing Wu and Rui Zhang. Towards smart and reconfigurable environment: Intelligent reflecting surface aided wireless network. *IEEE Communications Magazine*, 58(1):106–112, 2019.

[36] Xi Xiong, Justin Chan, Ethan Yu, Nisha Kumari, Ardalan Amiri Sani, Changxi Zheng, and Xia Zhou. Customizing indoor wireless coverage via 3d-fabricated reflectors. In *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments*, pages 1–10, 2017.

[37] Hongjing Xu, Shenheng Xu, Fan Yang, and Maokun Li. Design and experiment of a dual-band 1 bit reconfigurable reflectarray antenna with independent large-angle beam scanning capability. *IEEE Antennas and Wireless Propagation Letters*, 19(11):1896–1900, 2020.

[38] Fan Yang, Ruyuan Deng, Shenheng Xu, and Maokun Li. Design and experiment of a near-zero-thickness high-gain transmit-reflect-array antenna using anisotropic metasurface. *IEEE transactions on antennas and propagation*, 66(6):2853–2861, 2018.

[39] R Ivan Zelaya, William Sussman, Jeremy Gummeson, Kyle Jamieson, and Wenjun Hu. Lava: fine-grained 3d indoor wireless coverage for small iot devices. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 123–136, 2021.

[40] Lei Zhang, Ming Zheng Chen, Wankai Tang, Jun Yan Dai, Long Miao, Xiao Yang Zhou, Shi Jin, Qiang Cheng, and Tie Jun Cui. A wireless communication scheme based on space-and frequency-division multiplexing using digital metasurfaces. *Nature electronics*, 4(3):218–227, 2021.

[41] Lei Zhang, Xiao Qing Chen, Shuo Liu, Qian Zhang, Jie Zhao, Jun Yan Dai, Guo Dong Bai, Xiang Wan, Qiang Cheng, Giuseppe Castaldi, et al. Space-time-coding digital metasurfaces. *Nature communications*, 9(1):1–11, 2018.

[42] Na Zhang, Ke Chen, Yilin Zheng, Qi Hu, Kai Qu, Junming Zhao, Jian Wang, and Yijun Feng. Programmable coding metasurface for dual-band independent real-time

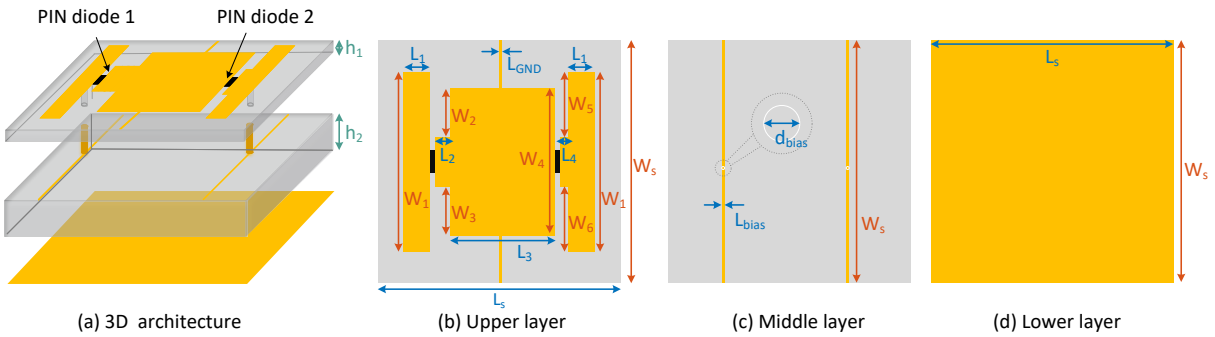


Figure 35: The structure of optimal unit-cell.

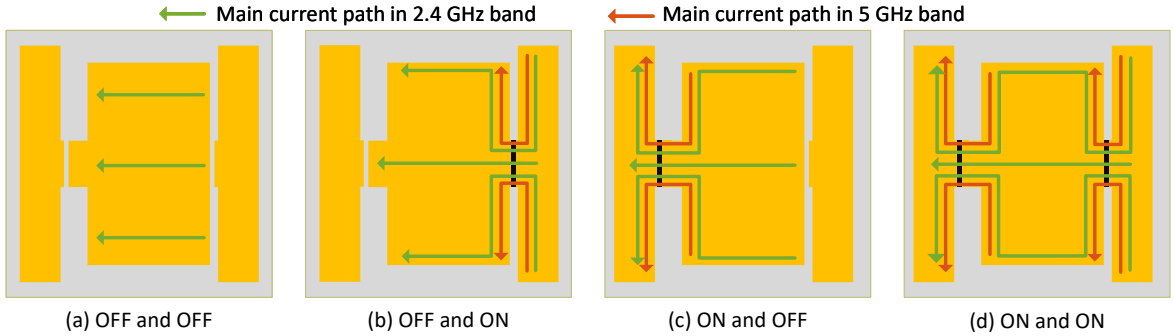


Figure 36: The meta-atom as a 2-bit phase shifter.

beam control. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(1):20–28, 2020.

A Appendix

Figure 35 illustrates the structure of optimal unit-cell and Table 2 summarizes the optimal unit-cell parameter configurations. We place two PIN diodes with the opposite orientation on the upper patch layer as shown in Figure 35(b). The bias current supported by a DC voltage regulator flows from the bias line and flows to the patch through two vertical via-holes. Then, it flows to the GND line after passing through PIN diodes. Under the different DC voltage levels, each PIN diode switches to "ON" or "OFF" state and thus the opening directions of each patch have four states, as shown in Figure 36. Depending on the sign of the bias current, the meta-atom introduces a phase shifting of 0. $\pi/2$, π, or $3\pi/2$. In addition, in order to decouple the influence between each bias line, we partition the whole metasurface into 4 areas (e.g., A, B, C, and D in Figure 13, so the maximum number of bias lines passing through a unit-cell is reduced from 16 to 8.

Therefore, we can consider the meta-atom as a 2-bit phase shifter, corresponding to four electromagnetic responses. In order to independently adjust each meta-atom's phase, we employ a bias line layer to control the states of the PIN diode within each meta-atom.

Table 2: The parameters of the unit-cell.

Parameter	Value (mm)	Parameter	Value (mm)
L_s	19.5	W_s	19.5
L_1	3	L_2	1.4
L_3	9.1	L_4	0.2
L_{bias}	0.2	L_{GND}	0.4
W_1	17.5	W_2	5.8
W_3	5.8	W_4	15
W_5	7	W_6	7
h_1	0.3	h_2	6.5
d_{bias}	0.8		

Scalable Distributed Massive MIMO Baseband Processing

Junzhi Gong
Harvard University

Anuj Kalia
Microsoft

Minlan Yu
Harvard University

Abstract

Massive MIMO (multiple-in multiple-out) is a key wireless technique to get higher bandwidth in modern mobile networks such as 5G. The large amount of computation required for massive MIMO baseband processing poses a challenge to the ongoing softwarization of radio access networks (RAN), in which mobile network operators are replacing specialized baseband processing chips with commodity servers. Existing software-based systems for massive MIMO fail to scale to increasingly larger MIMO dimensions with an ever-increasing number of antennas and users. This paper presents a new scalable distributed system called Hydra, designed to parallelize massive MIMO baseband processing while minimizing the overhead of distributing computation over multiple machines. Hydra's high scalability comes from reducing inter-server and inter-core communication at different stages of baseband processing. To do so, among other techniques, we take advantage of hardware features in modern commodity radios in novel ways. Our evaluation shows that Hydra can support over four times larger MIMO configurations than prior state-of-the-art systems, handling for the first time, 150×32 massive MIMO with three servers.

1 Introduction

Massive MIMO is a key wireless technique to increase spectral efficiency in modern mobile networks such as 5G. Massive MIMO refers to using a large number of radio antennas to simultaneously serve a large number of users on the same frequency resources. Mobile network operators today are deploying multi-user massive MIMO to handle the increasing demand from mobile users [16]. For example, T-Mobile recently demonstrated the benefits of massive MIMO in a setup with 64 antennas serving eight concurrent users [12], achieving an impressively high total downlink bandwidth of 5.6 Gbps. A promising way to handle the demand for higher spectral efficiency and mobile bandwidth is to increase the massive MIMO dimensions: the number of radio antennas, and the number of users served simultaneously [17, 26, 28].

While the previous-generation LTE networks typically used small MIMO configurations (e.g., four antennas), massive MIMO deployments with 64 antennas are already commonplace in 5G, and future deployments could use hundreds of antennas [16]. For example, AirSpan's Air5G 7200 already supports 128 transmit and 128 receive antennas [2].

This paper tackles the challenge of scalably supporting increasing massive MIMO dimensions in *virtualized* RANs (vRAN). With vRANs, mobile network operators are replacing specialized RAN hardware, such as ASICs and DSPs for wireless signal processing, with commodity x86 servers [5, 10, 11, 13, 15]. RAN virtualization offers important benefits, such as mitigating vendor lock in and increasing RAN flexibility and feature velocity. However, massive MIMO remains a challenge for software-based RANs [8]. This is due to the extremely high computational requirements of massive MIMO, in the presence of tight millisecond-scale latency deadlines. For example, the largest massive MIMO configuration considered in this paper—150 antennas and 32 users—requires our system (Hydra) to use 71 CPU cores, cumulatively handling 80.6 Gbps of fronthaul traffic, within a latency deadline of 2.5 ms.

Our goal is to design a system that can efficiently scale to increasing massive MIMO dimensions by using the resources of more servers, to handle the requirements of 5G and future radio technologies. Key to Hydra's scalability is a set of new techniques that we design to scalably distribute massive MIMO computation among a pool of servers while minimizing the distribution overhead from inter-server and inter-core communication. Existing projects that implement massive MIMO baseband processing in software, such as Agora [17] and BigStation [28], lack a path for scaling to increasing MIMO dimensions. One the one hand, single-machine systems like Agora and Intel's FlexRAN [3] are limited to the CPU and network bandwidth resources of only one machine. One the other hand, the BigStation project studies the opportunities for distributing multi-user MIMO computation, but does not seek to optimize the distribution overhead, which is the focus of this work.

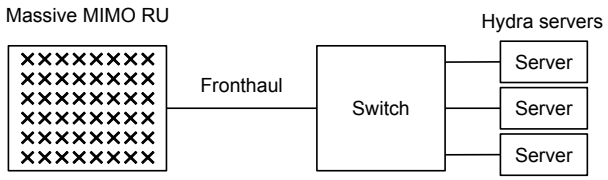


Figure 1: The architecture of a virtualized and distributed massive MIMO baseband processing system.

A massive MIMO baseband processing system (also called a baseband unit, or BBU) connects to a multi-antenna radio unit (RU) over a wired fronthaul link (Figure 1). In our setup, the BBU consists of one or more servers in a datacenter, connected to the fronthaul via an Ethernet switch. The RU and the BBU exchange packets containing in-band and quadrature (IQ) samples for hundreds or thousands of Orthogonal Frequency Division Multiplexing (OFDM) subcarriers. The BBU's computation consists of a pipeline of stages, each with a different type of parallelism [18, 28]. Hydra's design includes three key new ideas (summarized below) to map these BBU computation stages and the inter-stage shuffling of intermediate data to different hardware components, while minimizing inter-server and intra-server communication for scalability.

Taking the uplink direction as an example, the computation stages are as follows (Figure 2). First, antenna-parallel processing converts each antenna's time-domain IQ samples into the frequency domain using a fast Fourier transform (FFT). Second, subcarrier-parallel processing converts each subcarrier's per-antenna IQ sample streams into demodulated per-user streams. Third, user-parallel processing runs forward error correction on the per-user streams, converting them to user bit streams. The BBU connects these stages using communication mechanisms that shuffle the outputs of one stage into the inputs of the next stage.

Our first two ideas reduce inter-server communication (compared to BigStation), and the third reduces intra-server communication (compared to Agora).

1. **BBU-RU interface.** We identify that an existing hardware feature in modern RUs—the ability to perform FFT and generate separate packets for configurable subcarrier ranges—can be used to build a scalable distributed system for massive MIMO. Note that these RU abilities were not originally intended to build scalable distributed systems (Section 3.1); instead, we found a novel use case of these abilities. These FFT-capable RUs exchange frequency-domain IQ samples with the BBU, instead of time-domain IQ samples like in BigStation. Hydra routes packets for different subcarrier ranges to different servers, partitioning the fronthaul traffic and feeding the subcarrier-parallel pipeline stage with near-zero overhead. Compared to BigStation's design in which BBU servers run FFT and shuffle subcarrier ranges in software, our approach reduces inter-server communication by up to 66.4%.

2. **Within the BBU cluster.** Due to abundant parallelism, massive MIMO BBU processing offers many options to distribute computation within the server pool, at the cost of inter-server communication. For example, BigStation shuffles the BBU's intermediate data over the network within the subcarrier-parallel stage, and predicts benefits from splitting individual matrix inverse operations across servers. Such inter-server communication limits BBU scalability. To minimize inter-server communication in Hydra, we observe that the subcarrier-parallel stage transforms the data dimension from antennas to users, and massive MIMO by nature uses much fewer users than antennas. Therefore, we delay inter-server communication until after the subcarrier-parallel stage, shuffling only a small fraction of the BBU's input data rate among its servers.

3. **Within one server.** Within a machine, Hydra affinitizes the processing of an OFDM subcarrier to a CPU core. This ensures that the same CPU cores process a subcarrier through multiple subcarrier-parallel BBU sub-stages, eliminating inter-core shuffling of intermediate outputs. Hydra also avoids centralized scheduling of BBU tasks, which reduces inter-synchronization overhead, and prevents a single thread from becoming a bottleneck. This allows Hydra to use up to 47% fewer CPU cores than Agora's design (Section 5.6).

Besides the above key ideas, we also dynamically increase or decrease the number of CPU cores used, to efficiently handle the varying demands in mobile networks and reduce the energy consumption. We build Hydra starting from Agora's open source implementation. Our evaluation with an RU emulator shows that the number of antennas and users that Hydra can handle scales with the number of servers. With three servers, Hydra handles 150×32 MIMO, which has 2.3x more antennas and 2x more users than the prior state of the art single-machine system (Agora). With a larger 18-node cluster of old servers, Hydra handles 256×32 MIMO. Our evaluation also shows Hydra reduces CPU use by 46% when the traffic demand is low, compared to the corresponding system without dynamic core scaling.

2 Background and motivation

Antennas at a multi-user massive MIMO RU receive wireless signals that are a combination of several users' transmissions. Each antenna has associated hardware that digitizes these signals into per-subcarrier IQ samples (typically represented as fixed-point complex values), assembles the IQ samples into packets, and transmits them to the BBU over a wired fronthaul link. The BBU's task is to recover the bits

transmitted by each user from these jumbled complex numbers.

Doing so requires a huge amount of signal processing on the IQ samples, including matrix operations and forward error correction. This high computation cost is justified by the corresponding improvements in spectral efficiency. Note that although server hardware is an important contributor to operating expenditure, spectrum is often the most valuable resource in mobile networks.

2.1 Massive MIMO basics

Traditional single-user base stations allows at most one user to communicate with the base station on a given frequency resource (i.e., a subcarrier), avoiding inter-user interference. Multi-user MIMO uses interference canceling to allow a mobile base station to serve multiple users concurrently on the same subcarrier. Multi-user MIMO exploits *spatial* diversity, which means that different users are separated in physical space and therefore have different channels to the RU. Massive MIMO refers to multi-user MIMO with a large number (typically 32 or more) of base station antennas.

An M×N massive MIMO configuration uses M RU antennas to simultaneously serve N user antennas. On a given subcarrier, we can represent the signals transmitted by the N user antennas using an N×1 complex-valued column vector $x_{N\times1}$. The signal $y_{M\times1}$ received by the RU is a mixture of all users' transmissions. y can be modeled as $\approx W_{M\times N} \times x_{N\times1}$, where W is the "channel matrix", i.e., $W_{i,j}$ is the wireless channel between antenna i and user j.

Zero-forcing receivers. The BBU's main task then is to jointly process the signals y from all RU antennas to recover the users' signals x. Importantly, the BBU can do this joint processing for each subcarrier in parallel. The joint processing consists of two steps. First, the BBU estimates the channel matrix W by using "pilot" transmissions from the users that have well-known numerical values, and are separated in time or frequency to avoid inter-user interference. Second, with the common "zero-forcing" approach, the BBU then computes the pseudo-inverse of the channel matrix W, as $H = (W^*W)^{-1}W^*$. For subsequent non-pilot data transmissions, the BBU recovers an approximation of x by computing $H \times y$, in a process called *equalization*.

After reconstructing x, the BBU performs *demodulation* to map the complex numbers to bits. The demodulated bits contain both user data bits and parity bits, appended by the radio protocol.

Finally, the BBU *decodes* the demodulated output via a forward-error correction (FEC) algorithm to produce the users' bits. Similar to Agora [17], Hydra uses 5G's Low Density Parity Check (LDPC) algorithm for decoding.

2.2 Massive MIMO baseband processing

Our goal in Hydra is to distribute massive MIMO BBU processing among a pool of servers using the fewest number

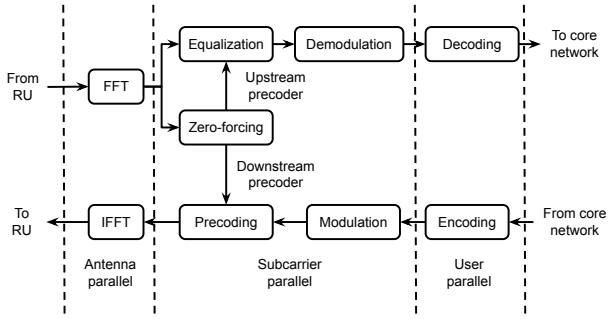


Figure 2: Massive MIMO processing pipeline.

of CPU cores and servers, achieved by minimizing distribution overheads from inter- and intra-server communication. We next discuss the two aspects of the massive MIMO processing pipeline that are crucial for designing a scalable distributed system: (1) the opportunity for distribution presented by the different types of parallelism in each stage, (2) and the scalability challenge posed by the need to shuffle data from one pipeline stage's output to the next stage's input (which we colloquially call the "data shuffling overhead").

Through the paper, we use the largest massive MIMO configuration supported by Hydra as a running **example** for exposition: 150×32 MIMO; with a typical 20 MHz configuration: 2048 subcarriers, out of which 1200 subcarriers carry data and the rest are used for guard bands; and 1 ms "slots" (discussed next).

Slots and symbols. Wireless protocols such as 4G and 5G divide time into slots. Each slot duration is typically further subdivided into 14 *symbol* durations. In each symbol duration, each RU antenna sends packets to the BBU containing IQ samples for all subcarriers. The radio protocol reserves pre-configured symbols for pilot signals from users, which are used for channel estimation. Similar to prior work [18, 28], we use the first symbol for pilots.

2.2.1 Types of parallelism

As noted by prior work [18, 28], massive MIMO baseband processing exhibits parallelism in different dimensions at different stages of the processing pipeline. This allows BBUs to divide the processing among multiple workers (i.e., CPU cores or servers). Figure 2 shows the three dimensions of parallelism for both uplink and downlink.

In the first antenna-parallel stage, the BBU performs FFT on the 150 antenna streams in parallel, converting time-domain IQ samples into frequency-domain samples. This step also eliminates the guard subcarriers and retains the 1200 subcarriers. The second frequency-parallel stage consists of three sub-stages: the BBU performs channel inversion, equalization, and demodulation for the 1200 subcarriers in parallel. The BBU may amortize the high cost of

matrix inversion by assuming that the channel matrix for some small configurable number of consecutive subcarriers is the same. In the third user-parallel stage, the BBU performs FEC decoding for each user independently.

2.2.2 Challenge: Inter-stage data shuffling

Massive MIMO processing is not perfectly parallelizable because the type of parallelism changes at each stage of massive MIMO processing, requiring shuffling the output of one stage to the input of another stage. One of our design goals in Hydra is to minimize this shuffling overhead.

Antenna-parallel to subcarrier-parallel shuffling. Consider a thread T_{fft} that has computed the FFT for a fronthaul packet in the antenna-parallel stage. At this point, T_{fft} has data for all 1200 subcarriers. T_{fft} must then transmit data from different subcarrier ranges to the subcarrier-parallel stage threads processing the corresponding subcarrier ranges. This transmission uses shared-memory for destination threads on the same machine, and over-the-network transmission for remote threads.

Subcarrier-parallel to user-parallel shuffling. Consider a thread T_{sc} that has finishes the subcarrier-parallel stage (i.e., demodulation) for its partition of subcarriers. At this point, T_{sc} has data for all users, and must transmit different user ranges to threads running user-parallel processing.

In Section 3, we discuss how Hydra avoids the overhead of both explicit and implicit *intra*-stage data shuffling in prior BBU designs.

2.3 The need for distributed computing

Our goal for building a scalable distributed design for massive MIMO BBUs is to provide a path for scaling to massive MIMO's ever-increasing computational demands. If massive MIMO vRANs are limited to a single server, they will suffer from limited mobile bandwidth, spectral efficiency, and have a less competitive feature set compared to traditional BBUs based on specialized hardware. Note that while this paper focuses on increasing antennas and users as the main driver for higher computation requirements, other important factors such as increasing the frequency bandwidth (e.g., 20 MHz to 100 MHz) and decreasing the slot size (e.g., 1 ms to 0.5 ms) also substantially increase the computation resources required and fronthaul traffic bandwidth.

PHY latency deadlines. The radio protocol's NACK (negative acknowledgment) turnaround time imposes a latency deadline on BBU processing. For example, in 4G and 5G, in case of an irrecoverable bit error on the uplink, the BBU must send a downlink NACK to the user within four slots (i.e., within 4 ms). In this work, we set Hydra's latency deadline to 2.5 ms at the 99.99-th percentile, to allow 1.5 ms for the MAC to schedule the downlink NACK.

High computational requirements. The number of CPU cores required to meet the BBU's latency deadline increases with MIMO dimensions, eventually exceeding the capacity of a single machine and necessitating a distributed design. For example, even after our optimizations, Hydra requires two servers to support 128×32 MIMO, and three servers to support 150×32 MIMO. Our 150×32 massive MIMO configuration requires 71 CPU cores. Note that although servers with very large numbers (100+) of high performance cores are available today, vRAN operators typically deploy smaller servers due to constraints such as power draw and fleet homogeneity. We explain these factors in detail next.

Limitation of single-machine systems. The CPU requirement of large MIMO configurations such as 150×32 (71 cores) is too high for a single vRAN server. This is because vRAN servers are deployed in small edge datacenters that have limited energy and space budgets, which precludes using beefy servers (e.g., quad-socket servers with 100+ cores). vRAN servers are typically single-socket or mid-range dual-socket servers. For example, HPE's servers targeted for vRAN have at most 28 CPU cores [14]. Similarly, Dell's reference architecture for vRAN has 40 cores per server [4].

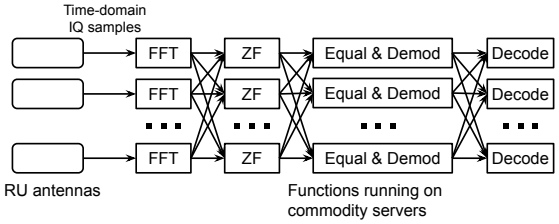
In addition, massive MIMO servers co-exist with vRAN servers handling other workloads, such as BBUs for non-massive RUs, and virtualized implementations of higher cellular protocol layers (e.g., MAC). Datacenter operators prefer maintaining a uniform fleet of servers, i.e., it is uncommon to deploy special high core-count servers for just one workload. Therefore, a distributed design that can support massive MIMO workloads in typical vRAN servers is useful.

Another advantage of not relying on high-end beefy servers, which we have currently not explored in this work, is cheaper fault tolerance. vRAN deployments must provide extremely high availability since they are part of the critical phone infrastructure. One way to limit vRAN downtime is to deploy some servers as hot backups. Maintaining a beefy backup server to guard against the failure of a single beefy server is more expensive compared to maintaining a smaller backup server to guard against failure of one of Hydra's servers.

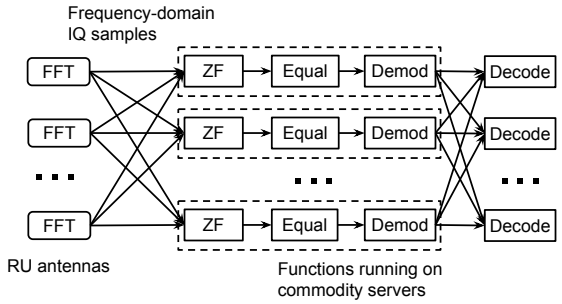
2.4 Limitations of prior distributed designs

BigStation [28] is the state-of-the-art design for virtualized distributed massive MIMO baseband processing. BigStation was designed around a decade ago for 4G MIMO processing, supporting up to 12 antennas and 12 users. BigStation's design (Figure 3) has two limitations:

High inter-core and inter-server communication. BigStation was designed for relatively small MIMO configurations, and aimed to meet latency deadlines with the weaker CPU cores available in 2012. Thus, BigStation aggressively distributes decomposable BBU tasks among CPU cores in the cluster (Section 3.2). As the MIMO dimension



(a) BigStation architecture



(b) Hydra architecture

Figure 3: Architecture comparison between BigStation and Hydra (uplink). Unlike BigStation, Hydra shuffles data only after the subcarrier-parallel stage. "ZF", "Equal", "Demod", and "Decode" are short for zero-forcing, equalization, demodulation, and LDPC decoding, respectively.

increases, the communication overhead in BigStation becomes significant and limits its scalability. In contrast, today we have higher MIMO dimensions and more powerful CPU cores that can individually complete the required MIMO operations within the radio protocol's deadline. Therefore, Hydra's design centers on keeping computation local on a CPU core to the extent possible.

Time-domain fronthaul traffic. BigStation was designed for older RUs that lacked FFT support (Section 3.1). To handle the large number of antennas in massive MIMO, BigStation's servers spend a correspondingly large amount of CPU cycles in (a) performing FFT and IFFT, and (2) sending, receiving, and shuffling IQ samples (Section 4). For example, in the uplink direction, the servers first receive time-domain IQ samples from antennas, perform FFT, and then shuffle the frequency-domain IQ samples among each other to enter the subcarrier-parallel processing stage.

2.5 Motivation and challenges for Hydra

The above limitations of single-server and distributed MIMO BBUs motivated us to create a new distributed design. The key challenge in Hydra is: how can we distribute massive MIMO BBU processing among a pool of workers (servers or CPU cores) with minimal overhead? An ideal design is to perfectly parallelize the workload among the BBU servers without any inter-server communication.

When splitting massive MIMO BBU processing for an RU with fronthaul traffic F_{bw} Gbps among N servers, the minimal network communication each server must handle is $C_{min} = F_{bw}/N$ Gbps. Our design achieves close to perfect parallelism with only 20% additional inter-server communication compared to C_{min} (Section 5.2).

3 Design

We next describe Hydra's three main design components. Our design focuses on reduces communication overhead, with three approaches: (1) using the ability of modern RUs to run FFT and split packets into subcarrier ranges (Section 3.1), (2) shuffling data between servers at a stage that minimizes inter-server traffic (Section 3.2), and (3) avoiding inter-core data movement and coordination within a server (Section 3.3).

For brevity, similar to prior work [18, 24, 29] we primarily focus on uplink processing, which is often more computationally intensive than downlink processing due to the higher cost of channel decoding compared to encoding. Interestingly, we find that downlink processing can be costlier than uplink on some server architectures (Section 5.2).

3.1 Scalable fronthaul traffic partitioning

Massive MIMO radios generate a high rate of fronthaul traffic. For a scalable design with minimal overhead, it is critical to not re-shuffle any substantial fraction of the fronthaul traffic between servers. Doing so adds overhead in terms of CPU cycles, latency, and datacenter network bandwidth use. We show that the new ability in modern RUs to run FFT and generate separate packets for different subcarrier ranges allows partitioning the fronthaul traffic among Hydra's servers with zero overhead. This reduces the amount of traffic that each server must handle before the subcarrier-parallel stage by up to 66.4% compared to BigStation.

Quantifying fronthaul bandwidth. For example, the fronthaul bandwidth in our running example configuration (Section 2.2)—150×32 MIMO, 20 MHz frequency bandwidth (2048 subcarriers, 1200 data subcarriers), 1 ms slots with 14 symbols—is 80.6 Gbps, assuming that the RU performs FFT and eliminates guard subcarriers: Each of the 150 antennas generates one packet with 1200 subcarriers (four bytes per IQ sample) for each of the 14 symbols in a slot. Therefore, the BBU receives $150 \times 1200 \times 4 \times 14$ bytes every millisecond, totaling 80.6 Gbps.

Note that most of the factors listed in Section 2.3 cause the fronthaul bandwidth to increase linearly. For example, using 100 MHz bandwidth with 0.5 ms slots is a common configuration in 5G deployments. This increases the fronthaul bandwidth by 5.5x to 444 Gbps by (1) increasing from 1200 to 3300 data subcarriers (4096 total subcarriers), and (2) doubling the rate at which the BBU receives packets.

To better describe the advantages of Hydra's fronthaul traffic partitioning approach, we begin by first discussing BigStation's approach. We compare the datacenter network

System	Receive rate	Transmit rate
BigStation	47.3 Gbps	12.9 Gbps
Hydra	20.2 Gbps	0 Gbps

Table 1: Comparison of datacenter network bandwidth used in BigStation and Hydra before the subcarrier-parallel stage. These numbers are for a 150-antenna RU and 20 MHz bandwidth with 1 ms slots.

bandwidth handled by each machine before the subcarrier-parallel stage in BigStation and Hydra, assuming each BBU design uses a cluster of four machines, and our running example MIMO configuration.

BigStation's approach BigStation uses RUs without FFT support, and therefore operates on time-domain IQ sample packets, with 2048 IQ samples each (one IQ sample per subcarrier). In our example MIMO configuration, the fronthaul bandwidth for BigStation is therefore $80.6\times2048/1200$ =137.5 Gbps. BigStation partitions the fronthaul traffic by antenna: each of the four BigStation servers receives packets for an exclusive range of 32 antennas, i.e., 137.5/4 = 34.4 Gbps.

BigStation's servers then run FFT on the time-domain IQ sample packets and drop the guard subcarriers, retaining 1200 subcarriers. To feed the next subcarrier-parallel stage (Section 2.2.2), each server must send 75% of the 1200 subcarriers to other machines. This corresponds to $32\times(0.75\times1200)\times4\times14$ bytes per millisecond, or 12.9 Gbps. Symmetrically, each server also receives 12.9 Gbps of shuffling input from the other three servers.

Hardware capabilities of modern radios The O-RAN alliance [9] defines specifications for various components and interfaces in 5G vRAN deployments. There are two hardware capabilities in modern O-RAN–compliant RUs that allow us to design new ways to partition fronthaul traffic. (1) FFT support. O-RAN's fronthaul specification [7] requires RUs to support FFT. Running FFT at the RU reduces fronthaul bandwidth requirement by dropping guard subcarriers at the RU. For example, in our 20 MHz configuration with 2048 subcarriers and 1200 data subcarriers, running FFT at the RU cuts down fronthaul bandwidth by 41%. Since FFT is cheap to implement, it is widely included in RUs. This O-RAN feature benefits massive and non-massive RUs alike (e.g., small cells in a city that have low-bandwidth connections to the BBU).

(2) O-RAN also requires the RU to support configurable "fragmentation" (Section 3.5 in the fronthaul spec [7]) of its IQ sample packets on both the uplink and the downlink. This is needed to support fronthaul networks with different MTUs (a fronthaul packet with 1200 subcarriers requires 4800 bytes, which exceeds Ethernet's typical 1500-byte MTU), and to allow the BBU to request only certain frequency ranges from the RU. The latter is useful for reducing fronthaul traffic during low load when only a few frequency resources are in use.

Hydra's approach We found a novel use case of these two RU abilities, which is different from what they were originally designed for (i.e., reducing fronthaul traffic or handling different MTUs): distributing massive MIMO fronthaul traffic among a pool of servers with zero overhead.

Fortunately, O-RAN RUs do not fragment packets at arbitrary boundaries, such as in the middle of an IQ sample. With such an implementation of fragmentation, Hydra's servers would need to re-shuffle some IQ samples between servers before the subcarrier-parallel processing stage. Instead, each fragment contains a contiguous range of subcarriers, and the BBU can configure these ranges over its control plane connection to the RU. In Hydra, we use as many equally-sized ranges as the number of servers. If the number of subcarriers is not a multiple of the number of servers, one server gets a slightly larger range than others.

For example, in a four-server Hydra cluster, we configure the RU to send four packets per antenna in each symbol duration. The four packets contain IQ samples for data subcarrier ranges 0–299, 300–599, 600–899, 900–1199. We then route the ith packet to server i. Each Hydra server therefore receives 80.6/4 =20.2 Gbps and sends no datacenter network traffic before entering the subcarrier-parallel stage.

Table 1 compares the amount of traffic received and sent by each server before the subcarrier-parallel stage in BigStation and Hydra for our example configuration. Hydra's total bi-directional bandwidth requirement per-server (20.2 Gbps) is 66.4% percent lower than BigStation's (60.2 Gbps). Our evaluation shows that using FFT-capable RUs with subcarrier range fragmentation reduces the number of CPU cores needed by Hydra by up to 46% (Section 5.5).

3.2 Scalable PHY computation partitioning

After partitioning subcarriers, Hydra still needs to run the subcarrier-parallel stage (i.e., zero-forcing, equalization, and demodulation), and the user-parallel stage (i.e., decoding). There are several possible approaches to partitioning this remaining computation among Hydra's servers. We observe that the massive MIMO processing pipeline progressively reduces the amount of data transferred between pipeline stages. Hydra minimizes the amount of inter-server data shuffling by delaying shuffling to the last stage of the MIMO processing pipeline (decoding). Compared to BigStation's design, Hydra's servers shuffle up to 42% less data (Section 3.2.2).

3.2.1 Hydra's approach

Recall that in our running example, the ith Hydra server S_i receives IQ samples for subcarriers $[300\times i, 300\times(i+1))$. S_i runs all the subcarrier-parallel processing sub-stages for these 300 subcarriers locally, i.e., without shipping any com-

putation to remote servers. After demodulation, S_i creates 32 per-user output buffers, each with 300 entries for the corresponding users' transmissions on each subcarrier. The decoding for 75% of these users happens on other servers, so S_i ships its outputs for those users over the network.

Rationale. The transition point between subcarrier-parallel and user-parallel stages offers an efficient point for shipping computation across servers. This is because the subcarrier-parallel stage reduces the amount of data flowing through the BBU's uplink pipeline: The equalization step for a subcarrier transforms per-antenna samples into per-user samples, and the number of antennas in massive MIMO is substantially larger than the number of users. For example, 64×8 is the typical massive MIMO configuration in today's 5G deployments. In our running 128×32 massive MIMO example, equalization shrinks the pipeline's flow by 4x.

3.2.2 BigStation's approach

The main alternative to our approach is computation and data shipping even within the subcarrier-parallel stage. For example, unlike Hydra, the computation for a given subcarrier in BigStation happens on different servers: BigStation reserves a set of its servers for only matrix inversion. These servers then ship the computed inverses to other servers running equalization and demodulation.

Comparison with Hydra. In our running example with 150×32 MIMO, we assume that groups of 32 subcarriers (as many as the number of users to avoid inter-user interference during pilot transmission) have the same channel matrix. Therefore, there are 1200/32 channel matrices, and each matrix is $150 \times 32 \times 8 = 38400$ bytes in size. Shipping these matrices over the network in each millisecond slot duration requires 11.5 Gbps.

Shuffling data between the subcarrier-parallel stage and the user-parallel stage for this MIMO configuration cumulatively requires 16 Gbps for a three-server BBU (Section 5.2). This shuffle is required in both Hydra and BigStation, but it is the only shuffle required in Hydra. Therefore, Hydra's inter-server shuffle bandwidth (16 Gbps) is 42% lower than BigStation's (11.5 Gbps + 16 Gbps) for this configuration.

Alternative approaches. Similarly, early versions of our system aimed to maximize parallelism in the MIMO BBU by dynamically shipping individual matrix inversion and multiply operations over the cluster. Our thinking was in line with BigStation's hypotheses [28, Section 5.3] that for very large MIMO systems, the matrix inverse and multiply operations may need to be partitioned across servers. However, such an approach is unnecessary and inefficient: distributing individual MIMO matrix operations is unnecessary because modern CPU cores are individually powerful enough to meet the BBU's deadline. For example, computing the pseudo-inverse of a 150×32 channel matrix takes only around 150 µs on our servers. Our evaluation shows that confining a subcarrier's processing to a single server (a single core) works well on today's hardware. In addition, shipping the matrix operations adds overhead by shipping a huge amount of matrix contents over the network, requiring similar bandwidth to the fronthaul bandwidth.

3.3 Scaling within a machine

In our goal to build a scalable distributed system for massive MIMO baseband processing, we also had to create new optimizations for the processing within a single machine. We found that our baseline Agora system has a large amount of overhead from inter-core data movement and synchronization. We next describe two optimizations to reduce inter-core data movement and synchronization that we made on top of Agora. Our evaluation shows that these optimizations are crucial: without them, for some large MIMO configurations, Hydra either fails to support the configuration, or requires over 2x more CPU cores (Section 5.6).

Subcarrier-to-core affinity Recognizing the high cost of inter-core communication, we design Hydra to use the same CPU core for all the subcarrier-parallel sub-stages for a given subcarrier. In contrast, Agora centers its design around fine-grained task distribution, so a random core runs any individual matrix inverse or matrix multiply. Although this is a straightforward way of parallelizing MIMO processing that provides flexibility in allocating tasks to cores, it incurs a large amount of inter-core communication.

In Agora, the CPU core that computes the channel matrix inverse for a subcarrier (C_i) is almost always different from the CPU cores that run equalization and demodulation for that subcarrier (C_e). Note that a given channel matrix inverse computed from the pilot symbols is used for equalization in 13 subsequent data symbols. This creates overhead by repeatedly moving the computed matrix inverse from C_i's private caches to C_e's caches. It also reduces the cache efficiency of all cores, since the same matrix contents are duplicated in several caches.

In Hydra, the same CPU core performs channel matrix inversion, equalization, and demodulation for a given subcarrier. This eliminates inter-core cache movement and duplication of cached data.

No central coordinator thread. Agora uses a coordinator-worker thread design, in which a single coordinator thread communicates with worker threads via shared-memory queues. The coordinator thread queues task descriptors to the workers (e.g., the address and dimension of a source matrix to invert), and receives completions from workers. We find that in Agora's design, the worker threads spend a substantial amount of time blocked and waiting for work from the coordinator. This happens because the coordinator must schedule a large number of tasks, restricting performance. In contrast, Hydra's threads use shared-memory counters to track dependencies in the

Task	Without HT	With HT
Invert a 150×32 matrix	2.9 K/sec	4.8 K/sec
Equalization (32×150 times 150×1)	0.96 M/sec	1.23 M/sec
Precoding (150×32 times 32×1)	0.67 M/sec	0.72 M/sec
LDPC decoding (one UE)	20.2 K/sec	23.3 K/sec
LDPC encoding (one UE)	48.5 K/sec	60.0 K/sec

Table 2: Comparison of single-core processing rate with and without Hyper Threading (HT) for 150×32 MIMO.

MIMO pipeline, avoiding the coordinator bottleneck. In addition, since we also use subcarrier-to-core affinity, there are over 10x fewer cross-core task dependencies in Hydra than in Agora.

3.4 Downlink processing

The BBU's downlink pipeline is the reverse of the uplink pipeline. A separate MAC layer sends per-user data to Hydra's user-parallel threads, which perform LDPC encoding, and send the corresponding subcarrier ranges to Hydra's subcarrier-parallel workers. Subcarrier-parallel workers then apply the precoder matrix that they computed during uplink processing (the precoder is the transpose of the channel matrix inverse) to transform per-user streams to per-antenna streams. The packet I/O threads on each machine then combine the outputs of each subcarrier-parallel worker to generate one packet for the machine's assigned subcarrier range, which is then sent to the RU.

4 Implementation

We implement Hydra in C++ for Linux, building on top of Agora's open-source codebase. Similar to Agora, (1) we use Intel MKL for matrix operations accelerated with AVX-512 SIMD instructions, (2) we use Intel's FlexRAN library [6] for LDPC decoding and encoding.

Thread types in Hydra. A Hydra deployment consists of one or more Hydra processes running on different servers in a cluster. Each Hydra process launches three sets of threads, each pinned to a different core: (1) packet IO threads, (2) subcarrier-parallel threads, and (3) user-parallel threads. Packet IO threads send and receive fronthaul traffic, and shuffle BBU pipeline data between servers. Each subcarrier-parallel and user-parallel thread is assigned a static range of subcarriers or range of users, respectively. For each MIMO dimension, we use the fewest number of threads for each thread type (currently determined manually) required to handle the maximum workload.

Hyper Threading. While Agora disables Hyper Threading (HT), we find that enabling it improves the performance of massive MIMO processing by reducing the negative impact of memory stalls. Massive MIMO processing generates a large memory footprint (e.g., 1200 150×32 matrices, or 11.5 MB), causing misses in the CPU's L1–L3 caches, which reduce CPU efficiency. Hyper Threading hides the impact of these memory stalls by overlapping memory accesses with compute, e.g., by allowing one logical thread to use a SIMD unit while another logical thread is stalled. Table 2 shows that for 150×32 MIMO, using HT improves a single core's throughput by 7.5%–65.5% for different PHY routines measured in isolation in micro benchmarks. For end-to-end runs, we were able to fit 150×32 MIMO processing in three servers only with HT enabled. For smaller MIMO configurations that we were able to test both with and without HT, using HT reduces the number of physical CPU cores needed by Hydra, e.g., from 68 to 53 cores for uplink processing for 128×32 MIMO.

Dynamic CPU core utilization. Since mobile networks experience highly variable workloads, it is important for the BBU to scale its energy consumption with the workload [1]. For example, cell sites in residential areas have high utilization during the day time, but almost no utilization at night. Hydra scales its CPU usage with the workload as follows.

For every slot, the MAC layer (not included in our system yet) communicates the set of users active on each subcarrier to Hydra. If during a slot with low load, the base station has fewer active users than the number of users permitted by the MIMO dimension, Hydra puts the corresponding user-parallel threads to sleep. Similarly, if a slot's MAC configuration has some subcarriers not assigned to any user, Hydra puts the corresponding subcarrier-parallel threads to sleep. While this approach is fairly simple, we believe that it works well because mobile networks experience highly bursty workload patterns, with significant periods of zero load. For example, recent measurements show that a 4G cell is fully idle in 75% of the slots [20]. During zero-load periods, Hydra disables most of its threads, keeping only the packet I/O threads active. The CPU cores yielded by Hydra may be used by other co-located latency-tolerant edge workloads such as machine learning and analytics [20].

5 Evaluation

This section presents our evaluation of Hydra's performance, the effectiveness of our design choices, and comparisons with design choices made by prior massive MIMO baseband processing systems (i.e., Agora and BigStation).

For evaluation, we created a complete version of BigStation based on the original design [29]. To focus the evaluation on the distributed system design differences between BigStation and Hydra, we also implement all of Hydra's single-machine optimizations for BigStation.

5.1 Evaluation setup

5.1.1 Server setup

We run our evaluation in two clusters. For most experiments, we use a "main" cluster of four commodity servers, with three servers running our distributed BBU, and one server acting as a fronthaul traffic generator emulating an

RU (Section 5.1.2). All servers are connected to an Arista 7060 switch with 100 GbE single-port Mellanox ConnectX-5 NICs. Each server has two Intel Xeon Silver 4216 CPUs (2.1 GHz, AVX512 support), with 16 cores per CPU. We use at most 29 cores per server to leave some cores for the OS to avoid kernel thread starvation.

To demonstrate our design's scalability to more servers than available in our main cluster, we use another cluster in CloudLab [19] consisting of 27 servers, with up to 18 servers for the BBU, and nine servers for emulating the RU. These servers are less powerful than our main cluster's servers. All servers are connected to Mellanox 2410 switches with a Mellanox ConnectX-4 25 GbE NIC. Each server has one ten-core Intel E5-2640v4 CPU (2.4 GHz, no AVX512 support).

As is typical in vRAN systems [3, 18], we configure each server to reduce jitter: we run our processes as real-time processes with the highest scheduling priority, and remap OS interrupts to an unused core. Our experiments run in a dedicated cluster without network congestion and therefore experience no packet loss. Unless specified otherwise, the experiments run in the main cluster.

5.1.2 Emulated fronthaul traffic generator

Since O-RAN-compatible massive MIMO radios are not readily available today, we emulate the fronthaul traffic with a software-based generator, using Agora's DPDK-based generator as a starting point. The generator applies a Rayleigh fading channel with Gaussian noise (25 dB signal-to-noise ratio). Agora's generator emulates a basic RU without FFT support, transmitting time-domain IQ samples; we modify it to emulate an O-RAN radio: our generator runs FFT, discards guard subcarriers, and splits packets into multiple subcarrier ranges, one per Hydra server. In addition, for the packet for a given subcarrier range, the generator uses the network address (IP and MAC address) for the corresponding Hydra server.

In the CloudLab cluster, which has 25 GbE, the fronthaul bandwidth exceeds a single server's NIC bandwidth for MIMO configurations with over 46 antennas. To overcome this, we split the traffic generation for the antennas across multiple servers. All servers are time-synchronized to a sub-microsecond accuracy with PTP, and agree on the first slot's start time during initialization. We also add generator support to change the set of active subcarriers and users to emulate high and low load scenarios.

5.1.3 Wireless parameters

Our wireless settings are similar to Agora: all experiments use a 20 MHz configuration with 1200 data subcarriers (2048 total subcarriers), 1 ms slot duration, and 64 QAM modulation. We use 1/3 LDPC code rate and base graph #1, with a LDPC lifting size ("Z") up to 104. This configuration results in 29.7 Mbps data rate per user, or 950 Mbps for 32 users. Since our primary focus is performance, our experiments use the peak load where all subcarriers and users are active, unless mentioned otherwise.

5.2 End-to-end performance

Figure 4 shows the number of CPU cores and servers needed by Hydra and BigStation to support different massive MIMO configurations. We show the numbers for both uplink and downlink processing. We run the experiment for 100 seconds, spanning 100k 1 ms slots. To support a MIMO dimension, the BBU must satisfy two constraints: (1) the BBU's 99.99th percentile latency must be below 2.5 ms (Section 2.3), and the BBU must have a throughput of one slot per slot duration (1 ms in our case) to keep up with the RU. Hydra supports up to 150×32 MIMO with three BBU servers. For 150×32 MIMO, Hydra uses 71 cores for uplink processing, or 83 cores for downlink processing.

Interestingly, we find that in our main cluster, Hydra's downlink processing is more expensive than uplink. This is the opposite of measurements in the Agora paper, as well as our CloudLab measurements for Hydra (Section 5.4.3). This happens because downlink precoding (0.72 M/s per core for 32×150 by 150×1 multiplications) is more expensive than uplink equalization (1.23 M/s per core for 150×32 by 32×1 multiplications) on our main cluster (Table 2).[1] In our CloudLab cluster, the lack of AVX512 instructions reverses this effect (i.e., downlink becomes cheaper than uplink) by making LDPC decoding far more expensive than LDPC encoding: decoding is 10x more expensive than encoding on CloudLab, compared to 2.5x more expensive on our main cluster.

5.2.1 Comparison with BigStation

BigStation supports only up to 128×16 MIMO with the three servers. For MIMO configurations that BigStation supports, Hydra uses only around half the CPU cores for uplink processing, and between 30–40% fewer CPU cores for downlink processing. BigStation's worse performance comes from two factors. First, BigStation spends additional CPU cycles for running FFT in software instead of the RU, and shuffling a larger amount of data between servers than Hydra. Second, the higher network I/O and data shuffling generates more memory pressure and inter-core communication than Hydra, reducing BigStation's compute efficiency.

We provide a detailed accounting of Hydra's and BigStation's CPU usage below, with 128×16 downlink processing (the most challenging downlink configuration supported by BigStation) as the example.

- **Packet I/O.** BigStation uses 24 cores for packet I/O, compared to only four for Hydra.
- **IFFT.** BigStation uses six cores for IFFT processing, whereas Hydra uses the RU's ability to perform IFFT.

[1] We are investigating the root cause of this difference by studying Intel MKL's implementation of matrix multiplication.

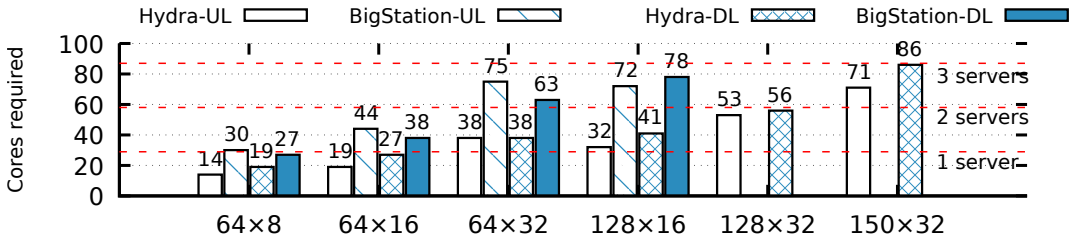


Figure 4: Number of cores and servers required to support different massive MIMO settings for Hydra and BigStation in uplink (UL) and downlink (DL) mode. BigStation supports up to 128×16, so the bars for larger configurations are not shown.

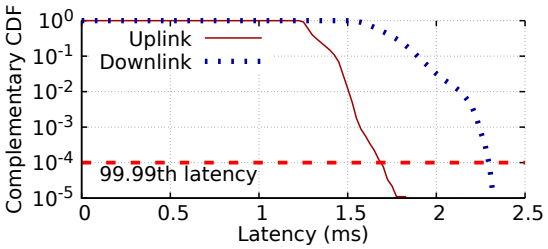


Figure 5: Complementary CDF of Hydra's latency for 150×32 uplink and downlink processing.

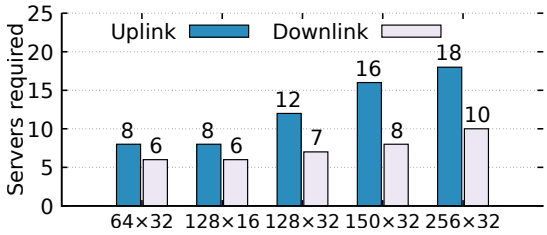


Figure 6: Number of servers required to support different massive MIMO settings in the CloudLab cluster.

- **Subcarrier processing.** BigStation uses 33 cores for subcarrier-parallel processing (six cores for zero-forcing, and 27 cores for precoding). Hydra uses 31 cores for its combined subcarrier processing stage.
- **LDPC encoding.** BigStation uses nine cores for LDPC encoding, compared to six for Hydra.

5.3 Comparison with Agora

Hydra supports Agora's largest 64×16 MIMO configuration with one server for both the uplink and downlink. Different from Agora's single-server design, Hydra allows using two servers to support 128×32 MIMO, and three servers to support 150×32 MIMO.

For a single-server performance comparison, we compare Hydra's performance with the numbers published in the Agora paper [18]. This is because we were unable to reproduce numbers comparable to those reported in Agora due to hardware differences, e.g., we use weaker CPUs (16-core Xeon Silver 4216, $900 per CPU) than those used in Agora's evaluation (16-core Xeon Gold 6130, $1900 per CPU). For 64×16 uplink processing, Hydra uses 19 CPU cores compared to Agora's 28 (including Agora's two packet I/O cores); for downlink processing, Hydra uses 27 cores compared to Agora's 23.

5.4 Hydra's performance details

5.4.1 Tail latency

Figure 5 shows that Hydra successfully meets our latency target of sub-2.5 ms 99.99th percentile latency for Hydra's largest-supported 150×32 MIMO configuration. For the uplink, Hydra's *maximum* latency is only 1.8 ms, and its 99.99-th percentile tail latency is 1.7 ms. For the downlink, Hydra's maximum and 99.99% latency are both 2.3 ms.

5.4.2 Additional network traffic

For 150×32 MIMO, Hydra processes 80.6 Gbps of fronthaul traffic, and cumulatively shuffles only 16 Gbps among servers, or 20% additional traffic. (Since there are three servers, each server transmits two-thirds of 32×400×6 bytes per data symbol. Since we use 64 QAM modulation, our demodulation stage represents each subcarrier's sample with 6 bits.)

5.4.3 Server scalability

We use the CloudLab cluster to study how Hydra's design scales with an increasing number of servers. Figure 6 shows how Hydra supports higher massive MIMO dimensions as the number of servers increases in the CloudLab cluster. Hydra supports 256×32 MIMO with 18 servers for uplink processing, or with 10 servers for downlink processing. For the regime studied, the number of servers needed for uplink processing scales roughly linearly with the number of antennas: for 32 users, Hydra needs 8, 12, and 18 servers for 64, 128, and 256 antennas, respectively. There is room for further scaling since Hydra does not hit a scalability bottleneck at 256×32; this scale was limited by only the number of CloudLab servers we managed to reserve.

Different from our main cluster, downlink processing is cheaper than uplink processing in the CloudLab cluster. This is primarily because LDPC decoding is 10x more ex-

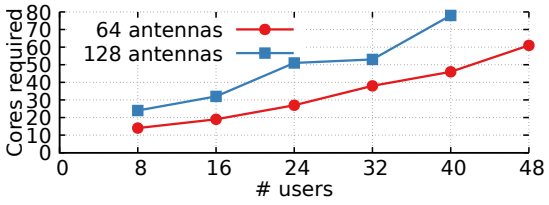


Figure 7: Minimum number of CPU cores required to support different massive MIMO settings.

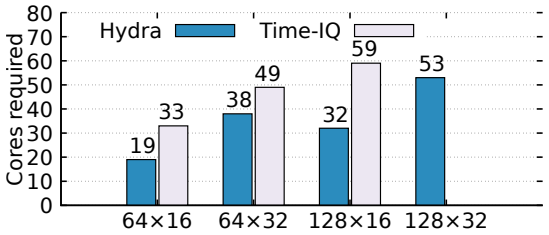


Figure 8: CPU cores needed by Hydra and Time-IQ for different massive MIMO configurations (uplink).

pensive than encoding on CloudLab servers, but only 2.5x more expensive on our main cluster.

5.4.4 User scalability

Figure 7 shows the minimum of CPU cores required to support an increasing number of users for two antenna configurations: 64 antennas and 128 antennas. We find that Hydra can scalably support more users by using more cores. Using LDPC accelerators (e.g., Intel's ACC100 accelerators) can allow Hydra to support even more users.

5.5 Benefits of leveraging RU features

To quantify the performance benefits of offloading FFT and subcarrier range splitting to the RU, we created a variant of Hydra called "Time-IQ" that works with time-domain IQ samples. Time-IQ runs FFT in software on the BBU servers to generate frequency-domain IQ samples, which it then shuffles among the servers for the subcarrier-parallel stage. Figure 8 compares the number of cores needed by Hydra and Time-IQ to support four different massive MIMO settings. Hydra uses 42%, 22%, and 46% fewer CPU cores for the 64×16, 64×32, and 128×16 configurations, respectively. Time-IQ is unable to support the 128×32 MIMO configuration with the 87 cores available in our cluster, whereas Hydra supports this configuration with 53 cores.

Next, we then run both Time-IQ and Hydra for 128×16 massive MIMO using 59 cores (the minimum CPU cores required by Time-IQ) and measure the 99.99-th tail latency breakdown. Figure 9 shows the 99.99-th percentile completion time for each of the three pipeline stages (the antenna-parallel FFT stage, the subcarrier-parallel stage, and the user-parallel decoding stage). Time-IQ has higher latency than Hydra due to the additional FFT processing in software, and a longer subcarrier-parallel stage. Time-IQ's subcarrier-parallel stage is longer because it must shuffle frequency-domain IQ samples between the antenna-parallel stage and the subcarrier-parallel stage.

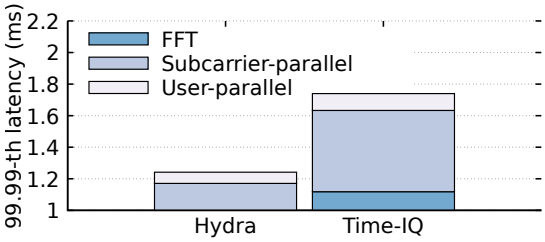


Figure 9: 99.99-th tail latency breakdown for Hydra and Time-IQ design for 128×16 MIMO (uplink) with 59 cores. The figure starts at 1 ms because it takes 1 ms to receive all IQ samples from RU antennas.

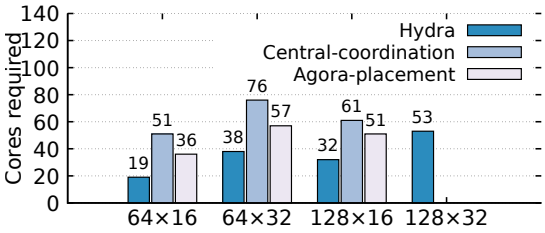


Figure 10: CPU cores needed by Hydra, Central-Coordination, and Agora-Placement for different massive MIMO dimensions (uplink).

5.6 Impact of intra-server optimizations

We next evaluate the effectiveness of our optimization to reduce inter-core communication (Section 3.3): affinitizing the processing of subcarriers to CPU cores, and avoiding a central coordinator thread for task scheduling. We created two variants of Hydra for this measurement: The first variant, called "Agora-Placement," works without a coordinator thread, but uses Agora's random assignment of tasks to CPU cores, which increase inter-core data movement and reduces cache effectiveness. The second variant, called "Central-Coordination," affinitizes subcarrier processing to CPU cores, but uses a coordinator thread to schedule tasks to workers. Figure 10 shows that reducing inter-core communication and avoiding centralization of task coordination logic is crucial for performance. In addition, the two variants are unable to support 128×32 MIMO with three servers.

For example, using a coordinator thread for task scheduling can more than double the number of CPU cores needed. We verify that this happens because of the large amount of time that worker threads in Central-Coordination spend in waiting for work from the coordinator thread. For example, with 128×32 MIMO and 53 cores (the minimum needed by Hydra to support 128×32), workers cores in the Central-

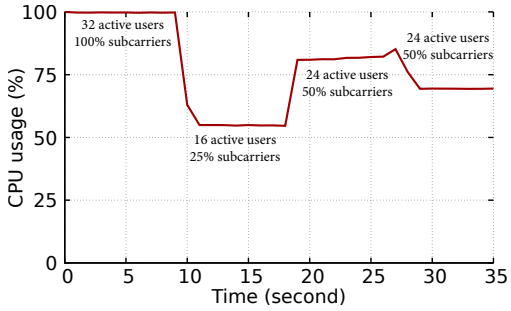


Figure 11: Hydra's CPU usage with a dynamic workload.

Coordination design spend around 0.5 ms on average in every millisecond slot waiting for the coordinator. In contrast, Hydra's workers spend only 0.14 ms on average waiting on shared-memory counters to saturate.

5.7 Dynamic CPU core scaling

We next evaluate the efficiency of Hydra's dynamic CPU core scaling mechanism. We run Hydra under 150×32 massive MIMO with 1200 data subcarriers, and dynamically change the number of active users and active data subcarriers. We change the workload in four stages, and each stage lasts for 8–10 seconds. The four stages are 1) 32 active users and 100% active subcarriers, 2) 16 active users and 25% active subcarriers, 3) 24 active users and 50% active subcarriers, and 4) 24 active users and 25% active subcarriers.

In a production RAN, the MAC layer sends commands to the PHY informing it about the number of active users and subcarriers in every slot; the PHY can use this information to adjust its CPU utilization [20]. Since we do not currently have a MAC layer, Hydra servers read per-slot configuration from a configuration file. Figure 11 shows the real-time CPU usage of Hydra over time, normalized to 71 cores at 100% load. Hydra first utilizes full CPU resource for the first stage, and reduces the CPU usage to 54% in the second stage. Hydra then dynamically changes the CPU usage to 80% and 70% in the third and the fourth stage.

6 Related Work

Software-based RAN processing. The use of commodity servers for high-performance PHY processing was pioneered by Sora [27], which demonstrated the use of modern CPU features such as SIMD for wireless signal processing for WiFi. The Sora project later led to BigStation [28], which was the first to use a distributed system to handle the high computation requirements of multiuser MIMO. Agora is a more recent project that focuses on massive MIMO processing within a single server. Hydra builds upon these designs by combining the single-machine design of Agora with ideas from BigStation, but focuses on minimizing the overheads in distributing massive MIMO computation. Intel's FlexRAN [3] is a production-grade single-machine PHY implementation is 5G NR-compliant and is used in large-scale vRAN deployments [10]. However, FlexRAN is closed-source, with a few open-source components like their LDPC encoder and decoder. Hydra's design could benefit from FlexRAN's other high-performance signal processing blocks, such as matrix inversion and demodulation.

Hardware-based RAN processing. The LuMaMi testbed [24] is a massive MIMO processing system that uses specialized hardware (e.g., FPGAs and PCIe switches). LuMaMi can handle 100×10 massive MIMO with 0.5 ms slots. While LuMaMi and Hydra cannot be compared apples-to-apples, it is interesting to note that Hydra can handle a substantially larger MIMO configuration (i.e., 150×32), although with a more relaxed latency deadline (1 ms slots). We believe that comparing software-only and hardware-based approaches for massive MIMO processing is an interesting avenue for future research.

Quantum computing approaches such as QuAMax [23] and ParaMax [22] have recently been proposed to tackle the high computational cost of massive MIMO. While our work uses linear MIMO methods (i.e., zero-forcing equalization and precoding), quantum-based approaches can handle more expensive non-linear methods like sphere decoding [21, 25]. Since sphere decoding can be too expensive for a single server, Hydra's techniques may be used to distribute the work among multiple servers.

7 Conclusion

We have presented the design of Hydra, a new distributed design for scalable massive MIMO processing in software. Hydra focuses its design on reducing the overhead in distributing massive MIMO computation among a pool of servers. Our design leverages features of modern RUs in novel ways to partition the fronthaul traffic with zero overhead, uses an efficient split for shuffling inter-server data between the MIMO pipeline's stages, and reduces inter-core communication and coordination for processing within a machine. The result is that Hydra can support much larger MIMO configurations than prior state-of-the-art, demonstrating support for 150×32 MIMO for the first time in software. Importantly, we have demonstrated that massive MIMO processing can be efficiently distributed over multiple servers, using only 20% additional network I/O compared to the required fronthaul traffic. We believe that our design can be used to scalably support even more challenging MIMO configurations in the future.

Acknowledgments. We thank the NSDI reviewers for their helpful feedback. We are grateful to Jian Ding and Rahman Doost-Mohammady for their feedback, and help with the Agora code. We also thank Lin Zhong for early discussions on the project. Junzhi Gong and Minlan Yu are supported in part by the NSF CNS-1955422 and CNS-1955487.

References

[1] A technical look at 5G energy consumption and performance. https://www.ericsson.com/en/blog/2019/9/energy-consumption-5g-nr.

[2] AirSpan 7200: Massive Throughput in a Single, Compact Unit OpenRANGE 7200. airspan.com/5g-products/.

[3] An Overview of FlexRAN* Software Wireless Access Solutions. https://software.intel.com/content/www/us/en/develop/videos/an-overview-of-flexran-sw-wireless-access-solutions.html.

[4] Building an open vRAN Ecosystem. https://www.delltechnologies.com/asset/en-us/solutions/service-provider-solutions/technical-support/altiostar-redhat-nec-and-dell-technologies-vran-solution-reference-architecture.pdf.

[5] Dish selects Fujitsu, Altiostar for 5G radios, Open vRAN. https://www.fiercewireless.com/operators/dish-selects-fujitsu-altiostar-for-5g-radios-open-vran.

[6] FlexRAN LTE and 5G NR FEC Software Development Kit Modules. https://software.intel.com/content/www/us/en/develop/articles/flexran-lte-and-5g-nr-fec-software-development-kit-modules.html.

[7] O-RAN Fronthaul Control, User and Synchronization Plane Specification v6.0. https://www.o-ran.org/specification-access.

[8] Open RAN and the mission to crack massive MIMO. https://www.lightreading.com/open-ran/open-ran-and-mission-to-crack-massive-mimo/d/d-id/768081.

[9] Operator Defined Open and Intelligent Radio Access Networks. https://www.o-ran.org/.

[10] Rakuten Mobile and NEC to Build Open vRAN Architecture in Japan. https://global.rakuten.com/corp/news/press/2019/0605_01.html.

[11] Telefonica invests in vRAN vendor Altiostar. https://www.fiercewireless.com/tech/telefonica-invests-vran-vendor-altiostar.

[12] T-Mobile Achieves Mind-Blowing 5G Speeds with MU-MIMO. https://www.t-mobile.com/news/network/t-mobile-achieves-mind-blowing-5g-speeds-with-mu-mimo.

[13] Vodafone starts trials of OpenRAN in Europe and Africa. https://www.gsma.com/futurenetworks/digest/vodafone-starts-trials-of-openran-in-europe-and-africa/.

[14] vRAN 2.0 on HPE Infrastructure. https://h50146.www5.hpe.com/products/servers/document/pdf/edgeline/vran2.0.pdf.

[15] Open RAN Alliance. O-RAN: towards an open and smart RAN. *white paper, October*, 2018.

[16] Robin Chataut and R. Akl. Massive mimo systems for 5g and beyond networks—overview, recent trends, challenges, and future research direction. *Sensors (Basel, Switzerland)*, 20, 2020.

[17] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. Agora: Real-time massive MIMO baseband processing in software. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 232–244, 2020.

[18] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. Agora: Real-time massive MIMO baseband processing in software. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 232–244, 2020.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[20] Xenofon Foukas and Bozidar Radunovic. Concordia: teaching the 5G vRAN to share compute. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 580–596. ACM, 2021.

[21] Chin-yun Hung and Tzu-hsien Sang. A sphere decoding algorithm for mimo channels. In *2006 IEEE International Symposium on Signal Processing and Information Technology*, pages 502–506, 2006.

[22] Minsung Kim, Salvatore Mandrà, Davide Venturelli, and Kyle Jamieson. Physics-inspired heuristics for soft mimo detection in 5g new radio and beyond. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 42–55, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Minsung Kim, Davide Venturelli, and Kyle Jamieson. Leveraging quantum annealing for large mimo processing in centralized radio access networks. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 241–255, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Steffen Malkowsky, João Vieira, Liang Liu, Paul Harris, Karl Nieman, Nikhil Kundargi, Ian C. Wong, Fredrik Tufvesson, Viktor Öwall, and Ove Edfors. The world's first real-time testbed for massive mimo: Design, implementation, and validation. *IEEE Access*, 5:9073–9088, 2017.

[25] Konstantinos Nikitopoulos, Juan Zhou, Ben Congdon, and Kyle Jamieson. Geosphere: Consistently turning mimo capacity into throughput. *SIGCOMM Comput. Commun. Rev.*, 44(4):631–642, aug 2014.

[26] Clayton Shepard, Jian Ding, Ryan E Guerra, and Lin Zhong. Understanding real many-antenna MU-MIMO channels. In *2016 50th Asilomar Conference on Signals, Systems and Computers*, pages 461–467. IEEE, 2016.

[27] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M Voelker. Sora: high-performance software radio using general-purpose multi-core processors. *Communications of the ACM*, 54(1):99–107, 2011.

[28] Qing Yang, Xiaoxiao Li, Hongyi Yao, Ji Fang, Kun Tan, Wenjun Hu, Jiansong Zhang, and Yongguang Zhang. BigStation: Enabling scalable real-time signal processing in large MU-MIMO systems. volume 43, pages 399–410. ACM New York, NY, USA, 2013.

[29] Qing Yang, Xiaoxiao Li, Hongyi Yao, Ji Fang, Kun Tan, Wenjun Hu, Jiansong Zhang, and Yongguang Zhang. BigStation: Enabling scalable real-time signal processingin large MU-MIMO systems. *ACM SIGCOMM Computer Communication Review*, 43(4):399–410, 2013.

DChannel: Accelerating Mobile Applications With Parallel High-bandwidth and Low-latency Channels

William Sentosa[•], *Balakrishnan Chandrasekaran*[†], *P. Brighten Godfrey*[•⋆], *Haitham Hassanieh*[◇], *Bruce Maggs*[‡]
[•]*UIUC,* [†]*VU Amsterdam,* [⋆]*VMware,* [◇]*EPFL,* [‡]*Duke University and Emerald Innovations*

Abstract

Interactive mobile applications like web browsing and gaming are known to benefit significantly from low latency networking, as applications communicate with cloud servers and other users' devices. Emerging mobile channel standards have not met these needs: 5G's general-purpose eMBB channel has much higher bandwidth than 4G but empirically offers little improvement for common latency-sensitive applications, while its ultra-low-latency URLLC channel is targeted at only specific applications with very low bandwidth requirements.

We explore a different direction for wireless channel design to address the fundamental bandwidth-latency tradeoff: utilizing two channels – one high bandwidth, one low latency – simultaneously to improve performance of common Internet applications. We design DChannel, a fine-grained packet-steering scheme that takes advantage of these parallel channels to transparently improve application performance. With 5G channels, our trace-driven and live network experiments show that even though URLLC offers just 1% of the bandwidth of eMBB, using both channels can improve web page load time and responsiveness of common mobile apps by 16-40% compared to using exclusively eMBB. This approach may provide service providers important incentives to make low latency channels available for widespread use.

1 Introduction

Low latency is critical to interactive applications such as web browsing, virtual and augmented reality, and cloud gaming. For web applications, even an increase of 100 ms latency can result in as much as 1% revenue loss, as noted by Amazon [21]. Emerging VR, AR, and cloud gaming applications also rely on low latency to deliver a seamless user experience. For instance, VR requires 20 ms or lower latency to avoid any simulator sickness [19].

Current mobile broadband, serving general Internet applications such as web browsing and video streaming, have not yet delivered consistent low latency performance, in part due to the inherent trade-off between latency and bandwidth [22]. One approach is to provide two separate channels (or services) – one optimizing for bandwidth, the other optimizing for latency – with different types of user applications assigned to them. 5G NR follows this pattern with its enhanced mobile broadband (eMBB) and ultra-reliable and low-latency communication (URLLC) channels. eMBB, which serves general-purpose Internet use, is heavily focused on delivering gigabit bandwidth. This channel will be useful for streaming media but offers little to no improvement for latency-sensitive applications, such as web browsing [34, 35, 50]. Experimentally, web page load time in existing 5G deployments, even in close-to-ideal circumstances (a stationary device and a channel with little utilization), is similar to 4G for pages smaller than 3 MB in size and about 19% faster than 4G for pages larger than 3 MB [34]. This is due to 5G eMBB having 28 ms or larger latency, broadly similar to 4G [34]. Our measurements of 5G mmWave showed similar results, at around 22 ms in ideal conditions.

Meanwhile, 5G URLLC promises an exciting capability of very low latency, in the range of 2 to 10 ms [6], but compromises severely on bandwidth, making it unsuitable for common mobile applications. Our experiments emulating web browsing (the most widely used mobile application [44], and far from the most bandwidth-intensive application) over URLLC with 2 Mbps bandwidth show web page load times would be 5.87× worse than with eMBB. Hence, neither using URLLC alone nor using eMBB alone provides good performance. As the latency-bandwidth trade-off is fundamental, this separation between a high bandwidth channel (HBC) and a low latency channel (LLC) is likely to persist; 6G, for example, is also expected to include both [54].

We believe, however, that the availability of two channels offers an opportunity to deal with the fundamental latency-bandwidth tradeoff in a new way, beyond simple static assignment of an application to a single channel. Specifically, we argue that *by using high bandwidth and low latency channels in parallel on mobile devices, significant performance and user experience improvements are possible for latency-sensitive applications.* Here, we explore this hypothesis for the case of web browsing and web-based mobile applications.

Mapping an application's traffic to HBC and LLC is difficult since we have to use LLC's bandwidth very selectively. Indeed, the main deployed transport-layer mechanism to combine multiple channels, MPTCP [49], assumes two interfaces that are each of significant bandwidth, with the goal of aggregating that bandwidth or supporting failover. LLC's bandwidth, however, is a rounding error compared to HBC's. Other works – particularly Socket Intents [42] and TAPS [38] – exploit multi-access connectivity through application-level input, which we prefer to avoid to ease deployment and expand relevance to other applications in the future; therefore we expect new mechanisms are necessary.

To solve these problems, we design DChannel, a system that leverages parallel channels to improve the performance of mobile applications. DChannel comprises two modules

running at either end of the channels – namely, in the mobile device OS and in a gateway device operated by the service provider. Central to the approach is a packet steering scheme that operates at the network layer (i.e., IP packets) without requiring any application input. Such fine-grained, per-packet decisions (as opposed to, for example, HTTP object-level steering) are key to making effective use of the limited LLC bandwidth. To decide which packets are worth accelerating, since LLC bandwidth is extremely limited, DChannel treats the channel as an expensive resource and calculates the benefit and cost of utilizing the LLC for each packet. Finally, since the parallel channels could occasionally confuse the transport layer with out-of-order delivery, DChannel employs a reordering buffer in the mobile device and gateway.

To evaluate our design with a concrete scenario, we leverage 5G's eMBB and URLLC as our HBC and LLC. We evaluate the benefit of DChannel in our experimental testbed (§4). Our testbed includes a prototype that can capture and steer application traffic, and a high-fidelity trace-driven network emulator that emulates cellular network latency variability and delay caused by radio resource control (RRC) state transitions [41]. We gather two types of real 5G eMBB traces – mmWave and lowband – in three different scenarios: stationary, low mobility, and high mobility. Our evaluations cover popular web applications such as web browsing and Android mobile applications. Using the testbed, we evaluate our packet steering scheme and compare it with prior approaches such as MPTCP [2] and ASAP [29]. We also evaluate DChannel in live 5G eMBB networks. Our key findings are as follows:

- DChannel, which requires little per-connection state and no application knowledge, yields superior performance compared to the other evaluated schemes—object-level steering, static packet-size-based steering, as well as prior work, MPTCP and ASAP [29], which used multiple channels in other settings.
- Compared with exclusively utilizing the eMBB, allocating a modest bandwidth of 2 Mbps to URLLC allows DChannel to improve web page load time (PLT). Under conditions that are ideal for eMBB (a stationary client with a line of sight to the base station and full signal strength), DChannel reduces PLT by 20% and 33% in 5G mmWave and low-band settings, respectively. Under more challenging mobile conditions, DChannel improves PLT by 37% and 42% in 5G mmWave and low-band, respectively.
- In addition to web browsing, we evaluated three Android mobile apps in a live environment and find DChannel improves apps responsiveness by 16% on average.
- Somewhat surprisingly, DChannel improves *sustained throughput* in our mobile 5G setting by roughly 10% – a useful side benefit of accelerating the TCP control loop in dynamic environments.

Finally, we discuss deployment strategies, challenges, and future opportunities. We believe our basic techniques can apply to a variety of latency-sensitive applications, and open new opportunities for app developers and cellular providers.

2 Background and Motivation

2.1 Channels in 5G

5G wireless networks are designed to support applications with very different service level requirements. The 5G standard known as New Radio (NR) specifies three service models: (1) enhanced mobile broadband (eMBB) for standard high-data-rate Internet and mobile connectivity, (2) ultra-reliable low-latency communication (URLLC) for mission-critical and latency-sensitive applications, and (3) massive machine-type communications (mMTC) for large-scale IoT deployments. We describe eMBB and URLLC in more depth.

(1) Enhanced Mobile Broadband: This service focuses on providing high-data-rate mobile access. It is considered an upgrade to 4G mobile broadband that will satisfy the ever-increasing demand for mobile and wireless data. 5G eMBB can operate either at the low-frequency bands below 6 GHz which we refer to as low-band or the high-frequency bands around 28 GHz/39 GHz which we refer to as millimeter wave (mmWave). The mmWave bands are a key new technology in 5G as they offer 10× the bandwidth that is currently available to 4G LTE networks [4], enabling user throughput of around 1 Gbps [15].

Providers like Verizon, AT&T, and T-Mobile have already deployed both the low-band and mmWave 5G in several major US cities, including Chicago, Atlanta, New York, and Los Angeles [9–11, 34]. A recent measurement study on commercial mmWave 5G networks in the US shows TCP throughput of up to 2 Gbps for download and 60 Mbps for upload, with a mean RTT of 28 ms measured between the client and the first-hop edge server right outside the cellular network core [34]. The measurements were performed, however, in conditions favorable to mmWave such as line-of-sight, no mobility, and few clients.

eMBB latency is expected to be higher as the number of users increases and as users move. This is because radio access networks (RANs) operating in the mmWave bands use very directional beams to compensate for high signal attenuation, making them vulnerable to blockage and mobility. High data rate communication is possible only when the RAN access point aligns its beam towards the user [27]. This process, commonly referred to as beam alignment, can introduce significant delays, especially when users are moving, which requires the access point to keep realigning the beam of each user [23, 27]. Furthermore, the user or other obstacles can easily block the beam, leading to unreliable and inconsistent performance both in terms of changes in throughput and highly variable RTT [3, 32, 34]. Our own experiments in Chicago also confirm this and show that the RTT can vary sig-

nificantly even for stationary clients and is further exacerbated while walking or driving. This is because 5G eMBB mainly optimizes for high data rates, focusing less on reliability and low latency.

(2) Ultra-Reliable Low-Latency Communication: Unlike eMBB, this channel focuses on providing highly reliable, very low latency communication at the cost of limited throughput. It aims to support mission-critical and emerging applications with stringent latency and reliability requirements such as self-driving cars, factory automation, and remote surgery. While the URLLC channel is yet to be deployed in practice, the standard specifies a target 0.5 ms air latency between the client and the RAN (1 ms RTT) with 99.999% reliability for small packets (e.g. 32 to 250 bytes) [15]. It also specifies a target end-to-end latency (from a client to a destination typically right outside the cellular network core) of 2 to 10 ms with throughput ranging between 0.4 to 16 Mbps depending on the underlying application [6]. URLLC is expected to operate in the sub-6 GHz frequency bands (e.g. 700 MHz or 4 GHz) and operators are expected to use network slicing to provide dedicated resources to URLLC clients in order to guarantee consistent performance in terms of latency and reliability across both the radio access network (RAN) and the cellular core [6]. Finally, client access to the URLLC channel will be controlled by the network operators. The access control network slicing mechanisms, however, are left to the operators' own implementations [8].

2.2 Web browsing traffic

While we evaluate several applications, web browsing is the major focus of this work and serves as a running example.

A single web page may contain tens to hundreds of relatively small-sized web objects distributed across multiple servers and domains. Consequently, web browsing traffic is characterized by its often short and bursty flows. A study across Alexa Top 200 pages found that the median number of objects in a page is 30, while the median object size is 17 KB [48]. Fetching these web objects translates to many HTTP request-and-response interactions across many short flows. The browser fires a page load event when it finishes rendering a page, which is used to determine Page Load Time (PLT), a performance metric for web browsing. Although PLT has some shortcomings, the alternatives are not free from issues, and PLT is most widely used. PLT is typically dominated by DNS lookup, connection establishment, and TCP convergence time—which require little throughput but are highly dependent on RTT. Prior work also showed that increasing TCP throughput beyond $\approx$ 16 Mbps offers little improvement in PLT [45].

Of course, web page loading is affected by client CPU and server delay, in addition to network delay. Prior work found that 35% of the PLT is spent in client-side computations [47]. But the above characteristics, combined with the fact that mobile CPUs have been getting increasingly powerful [26], still suggest that network latency plays an important part in mobile web performance. Moreover, a significant portion of network latency lies in the "last mile" connection of the cellular network. Many other mobile apps also rely on HTTP-based interaction with cloud services, resulting in similar network performance requirements.

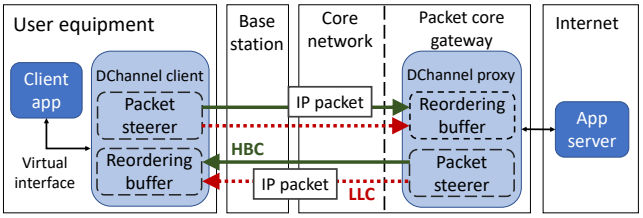


Figure 1: *The overview of DChannel. It has two main components: packet steerer that steers application traffic to LLC and HBC, and reordering buffer that reorders packets coming from LLC.*

3 DChannel Design

3.1 High-Level Architecture

To steer application traffic in both uplink and downlink channels, there will be two main components, one in the mobile client device and one in the mobile core network (Figure 1).

On the client, applications interact with the network through a network interface as usual. In our prototype, this is a special virtual TUN interface designated for traffic that should utilize both the HBC and LLC. The client-side agent captures outgoing packets on this interface and implements an algorithm to steer traffic between the two channels. The agent also captures incoming traffic on both channels and merges it into the virtual interface, after buffering it as needed to reorder packets (§3.6).

The proxy-side agent performs symmetric functions using the same algorithms – steering traffic headed towards the client, and merging and reordering traffic outbound to the Internet. This agent runs in the service provider's network, on a gateway at the point where the separate HBC and LLC channels begin. The exact location of the proxy-side agent may depend on the service provider's internal architectural choices; note that it is not necessarily located at the RAN base station, because the LLC's latency optimizations may extend into the packet core (e.g., for prioritized queuing and routing) [5].

The next subsections detail how we design the steering component, in several steps, as it is the more complex component. After that, we describe the reordering buffer.

3.2 Steering Granularity

To build the packet steering module, we begin with the question of the granularity, and corresponding layer, at which steering should occur. We considered splitting at two different layers: the application layer and the network layer.

Application-layer splitting refers to steering application requests and responses to the appropriate channels. In the context of web browsing, this approach translates to requesting and delivering web objects (in the form of HTTP requests) on either LLC or HBC. Application-layer splitting is broadly similar to Socket Intents [42].

Object-level splitting may benefit from application-level knowledge about web objects, which vary in size and priority. Since LLC is bandwidth constrained, LLC can only deliver small objects faster than HBC.[1] Web pages have complex dependency structures, and certain objects can be on the critical path for web page loading. These critical-path objects need not necessarily be small in size. Small objects might have low priorities such that accelerating them will not improve load time and thus would waste LLC bandwidth. In contrast, high-priority objects can be large such that sending those to LLC will be slower than HBC. Application-level input could help distinguish between these cases.

But object-level splitting has two drawbacks. First, we want to avoid requiring application input, which creates deployment hurdles and extra work for developers. Second, it misses opportunities for latency improvement. A web object that's not small enough to be sent over LLC will still involve small and latency-sensitive DNS lookups, TCP connection establishment, TLS handshaking, and ACKs. Accelerating this traffic could significantly reduce object delivery time. We later demonstrate (§5.3) that object-level splitting is less effective than finer-grained packet-level steering.

Steering packets at the network layer (e.g., IP datagrams) comes with its own challenges, however. First, we do not have any application-level insight into the flow: we do not necessarily know how packet-level acceleration affects application-level acceleration, so we will need a careful steering heuristic. Second, even if we identify the packets to accelerate, sending packets within a flow across two different channels might result in the packets arriving out-of-order, confusing TCP. To address this issue, we will introduce a small *reordering buffer (ROB)* at the endpoints. The following subsections discuss these components of the design.

3.3 Packet Steering Intuition

Define a "message" as a sequence of one or more packets such that the receiving endpoint can take some useful action after receiving the full message. For example, an individual SYN or ACK is a message (because the transport layer can act on it), and an HTTP request or a full response spread across multiple packets is a message (because the application may be able to process the request, display an object to the user, etc.). In contrast, an individual data packet belonging to a large HTTP request/response is not a message on its own and would not be worth accelerating individually since we need to accelerate the whole sequence of packets to finish the message.

Ideally, we would like to accelerate the delivery of messages, especially those that are most valuable to accelerate, within the bandwidth constraints of the LLC. This suggests a *cost-rewards calculation* weighing the benefit of accelerating a message against the cost of utilizing the meager bandwidth of the LLC which might be better spent on other messages.

A direct, exact cost-rewards calculation is infeasible since DChannel running at the network layer lacks full knowledge of message boundaries (in the application's data stream), as well as the relative value of messages to the receiver's transport layer or application. This leads us to begin with a permissive assumption: any packet *might* be a message boundary and we will optimistically consider accelerating it. Nevertheless, even operating transparently at the network layer, DChannel does have certain information about rewards and costs that will help it distinguish among packets.

First, the benefit of steering a packet to the LLC depends on how much its arrival time would improve, if at all, compared to using the HBC. This depends on packet size, current output queue lengths for both channels (which are locally observable), and latency of both channels (which can be estimated). In addition, the vast majority of applications utilize TCP or other transport that delivers messages in order.[2] This means that for a message inside packet P_i, *delivery of the message to the application* (as opposed to the delivery of P_i to the receiving host) will depend not only on the arrival time of P_i, but also on the arrival time of packets $P_0, \ldots, P_{i-1}$ (which can also be estimated). For example, suppose P_{i-1} was sent over the HBC, and P_i is ready to send immediately after. If P_i is also sent over HBC, the pair will arrive at about the same time. If P_i is sent over LLC, it will very likely arrive much sooner, but will end up waiting for P_{i-1} before it can be delivered to the application, meaning sending over the LLC is likely not useful in this case.

Second, the cost of utilizing LLC resources will depend on the packet length and how much the LLC will be in demand for other messages in the near future. The latter is not perfectly known, but current or recent outgoing LLC queue depths provide some signal.

The net effect of the above considerations is that packets should tend to get steered to the LLC when they are smaller, and when they are more isolated in time as individual packets or members of short packet sequences. This corresponds well with the intuition of prioritizing acceleration of control messages or small application-level messages. We now proceed to describe how we realize this cost-rewards approach.

3.4 Rewards and Cost

Problem statement. The packet steering algorithm is presented with a sequence of packets and needs to decide if each

[1] If URLLC is assigned a capacity of 2 Mbps (≈250 bytes per ms) and its RTT is ≈15 ms less than that of eMBB, any object of size larger than 3.75 KB are likely to be delivered faster on eMBB.

[2] Some don't, of course, but our goal in this work is to develop generic packet steering, leaving application-specialized schemes for the future.

packet P_n should be sent via LLC or HBC. We let $P_1, \ldots, P_n$ denote the sequence of packets in a single end-to-end flow (by which we mean a unidirectional transport layer connection, which may contain multiple messages).

Rewards. At the packet level, the objective is to minimize the *packet completion time* C_n, defined as the time by which all packets $P_0, \ldots, P_n$ would arrive at the receiver. This captures the intuition (§3.3) that any P_n might be a useful message to accelerate on its own, but it wouldn't be delivered to the application until prior packets are also delivered. The benefit of sending a packet P_n via LLC is thus the reduction of C_n if P_n is sent via LLC (denoted $C_{n,LLC}$), compared to when it is sent via HBC (denoted $C_{n,HBC}$). Thus, we calculate the rewards for sending P_n via LLC as: $R(P_n) = C_{n,LLC} - C_{n,HBC}$.

To calculate the above, we first need to estimate the delivery time D for a packet that depends on the channel/link[3] propagation delay $Dprop_{link}$ and bandwidth B_{link}, packet size, and the link's queue size Q_{link} at time t_n . The Q_{link} counts the number of bytes that have been enqueued for transmission through a *link* but have not yet been transmitted out the interface. Delivery time for P_n on a certain *link* is thus:

$$D_{link}(P_n) = Dprop_{link} + (size(P_n) + Q_{link}(t_n))/B_{link} \quad (1)$$

The packet completion time for P_n (C_n) should also account for completion times of P_0 through P_{n-1} (i.e., C_{n-1}) since P_n may arrive at the receiver before P_{n-1}, especially if P_n is sent over LLC and P_{n-1} was sent over HBC. Thus, we can calculate ($C_{n,link}$) as:

$$C_{n,link} = max(C_{n-1}, (t_n + D_{link}(P_n))) \quad (2)$$

Note that $Dprop_{link}$ are nondeterministic, comprising dynamic channel delay and any congestion along the channel's path, and will thus have to be estimated. We return to this later.

Cost. The cost of sending a packet to the LLC comes from the increased utilization of LLC. Intuitively, the cost should increase with the added queueing delay that a packet arriving very soon after P_n would experience, i.e., $size(P_n)/B_{llc}$. The cost should also be higher if the LLC is currently more highly utilized so that its limited capacity is reserved for higher-reward packets. We use a heuristic that captures this by adding these two effects; specifically, we compute the cost (or fare F) of putting P_n on LLC as:

$$F(P_n) = (size(P_n) + Q_{llc}(t_n))/B_{llc} \quad (3)$$

Note that to be more precise, we should compute the difference in costs of putting the packet on LLC vs. HBC. But as the HBC bandwidth is dramatically higher, its cost is negligible and we omit it for simplicity.

[3]We use these terms interchangeably for convenience. Note, however, the LLC channel may involve acceleration in the WAN in addition to the RAN, so it actually may span multiple physical links.

Comparing rewards and cost. At a high level, we want to steer packets to LLC when the rewards outweigh the costs, but comparing them involves a tradeoff: the benefit is immediate to packet P_n, whereas the cost affects possible subsequent packets which may not appear. We introduce a parameter α to capture this, so that we will send a packet to LLC when: $R(P_n) > \alpha F(P_n)$.

Calibrating α. If we set α too low, a flow may aggressively send packets to LLC so that it will deny resources to another flow in a multi-flow application. If we set it too high, we can be too conservative in utilizing the fast LLC. To find a good α and determine how sensitive performance is to its value, we conduct experiments with web browsing across different alpha values. We load 40 web pages from our corpus over different α values and pick α with the best Page Load Time (PLT) result on average. We use our testbed (§5.1) and apply the packet steering over HBC and LLC. For LLC, we use 5G NR URLLC as a reference where the RTT and bandwidth is 5 ms and 2 Mbps. For HBC, we vary its RTT while fixing bandwidth at 200 Mbps.

The detailed results are in §A.2. In summary, the results confirm that setting α too low or high has suboptimal performance. The best value for HBC RTT of 20 ms to 60 ms is 0.75. This RTT range covers most cases of 5G eMBB. As the RTT increases to 80 ms and higher, $\alpha = 1$ is slightly better. The difference, however, is less than 1%. We use $\alpha = 0.75$ for all subsequent experiments.

Note on design. The steering approach described here is not an optimal choice derived from a model – it is a heuristic, particularly the calculation of cost and calibration of α, in part since some of the relevant information (like the application-level importance of a particular packet) is unavailable. However: (1) we find the heuristic does perform well in realistic environments, (2) even if poor decisions do occur, they lead only to suboptimal performance, rather than a correctness problem, and (3) performance is not very sensitive to the exact value of α. In particular, even with $\alpha = 0$ – which corresponds to the greedy strategy, where each packet uses LLC whenever it expects a reward for itself – there is still a very good PLT improvement, within 5% or less of the best α. That said, this problem could be interesting to formalize in the future, perhaps as an online algorithm that could provide worst-case guarantees, or using queueing-theoretic tools.

3.5 The Packet Steering Algorithm

Putting together the above pieces, the complete steering algorithm is shown in Algorithm 1 in Appendix A.1. To make a decision, the algorithm requires (1) packet size, (2) current LLC queue size, (3) LLC bandwidth, and (4) latency of both LLC and HBC. The LLC bandwidth is controlled (assigned by the operator) so it is known, and (1) is directly observable.

LLC queue size (2) may directly be observable at the client, assuming its NIC is limited to the LLC bandwidth. But the proxy may have a higher local NIC rate. The proxy, therefore,

tracks outgoing traffic per user and computes what the queue depth would be if the NIC had been limited. Depending on the service provider's admission control policy, the rate could alternately be explicitly limited at the proxy. Client can also apply similar approach if (2) is not directly observable.

Latency (4) has to be estimated. To do this, we perform periodic handshakes (e.g., in every 500 ms in our use case). The handshakes consist of four steps, all with UDP packets: (1) the client agent sends a special packet we call a "D-SYN" to the proxy agent using both HBC and LLC. (2) The proxy agent upon seeing a D-SYN responds with "D-SYN/ACK" packets sent across both HBC and LLC. (3) The client agent receives the D-SYN/ACK packets, updates the base RTT value for both channels based on the difference between D-SYN/ACK receive time and D-SYN release time, and replies with "D-ACK" packets sent across both channels. (4) The proxy agent receives the D-ACK packets and updates the base RTT value for both channels. We use the minimum RTT value for the measurement. As we will see in the evaluation (§5), very rough latency estimates are sufficient.

The algorithm requires maintaining per-flow state, specifically to store C_{n-1}, the estimated completion time of the most recent previous packet. The proxy also stores per-user state for its queue depth calculation.

3.6 Reordering buffers at the endpoints

Splitting packets across asymmetric paths (particularly with a latency differential, as there is for LLC vs HBC) can cause out-of-order packet delivery, which can be harmful to application performance. In particular, TCP uses out-of-order packets as a signal of congestion, potentially causing retransmissions and a drop in sending rate. To solve this problem, we adopt a reordering buffer (ROB) in the receiving direction of each of our agents, to buffer packets arriving only from LLC. Note that we only buffer packets arrived from LLC as we only want to handle packet reordering caused by sending packets through the faster LLC and not to solve reordering caused by external factors such as wireless losses.

To avoid unbounded buffering delay if the previous packet was lost, the ROB also releases packets after a timeout. Ideally, the timeout should equal the latency of HBC, but because the latency of HBC can be variable and hard to track, we use a conservative 100 ms timeout. We evaluate the effectiveness of this timeout value under random packet loss in §5.

4 Prototype and Experimental Setup

Our experiments involve a client representing a mobile end-user application (e.g., a web browser) fetching content from a web or content server. Both the client and server endpoints have access to two interfaces, one representing the high-bandwidth channel (HBC) and the other the low-latency channel (LLC). In the case of 5G, HBC and LLC map to eMBB and URLLC, respectively. Depending on the experiment conditions, the interfaces may be real or emulated. We masked the two interfaces at the endpoints, however, using a smart DChannel virtual interface implemented on top of a `TUN` device; the client and server use only this virtual interface to send and receive data. Our DChannel prototype then performs endpoint-transparent (and application-agnostic) steering of traffic.

We developed a DChannel prototype and packaged it as a UNIX shell, similar to the shells in Mahimahi [36]. The shell captures all outgoing traffic from any *unmodified* application running within it and tunnels them to our DChannel implementation; it processes incoming traffic in a similar application-transparent manner, so both the steering and buffering modules of DChannel are used. Our DChannel prototype attaches additional metadata (sequence number and flow ID) prior to transmission to assist the receiver in reordering packets and strips this before delivering to the application. We used our own metadata header as a convenience, but in a real implementation, this could be avoided by looking inside the layer 4 header.

We evaluated the performance of DChannel using this prototype under two settings. The first is a *live* setting where we used the actual 5G NR eMBB channel as HBC. The second setting, in contrast, is one where we emulated the eMBB channel based on traces that we gathered from an actual 5G eMBB channel. In both settings, since URLLC is not yet commercially available, we emulated its "expected" behavior (based on the 5G specification [6]) using a low-latency, bandwidth-limited wired Ethernet connection.

4.1 Live-eMBB Setting

In this setting, DChannel steers traffic over two real interfaces (Fig. 2): One interface is tethered with a 5G phone for providing access to a *live* eMBB channel, while another is connected to a low-latency bandwidth-limited Ethernet connection for emulating the URLLC channel. Packets transmitted over the 5G eMBB channel traverse the core network of the mobile provider before exiting via the packet gateway (i.e., *mobile path*) and then one or more ASes in the public Internet (i.e., *Internet path*) to reach our server. Data sent over the Ethernet interface, in contrast, traverse a traditional ISP and then one or more ASes to reach the server. On the server side, DChannel receives all the packets from both the interfaces, reorders them (if required), and then delivers them to the server-side application via the `TUN` device.

We used Ethernet and not WiFi for emulating URLLC, since the channel is expected to provide high reliability (≥ 0.9999) [8]. We capped the bandwidth of this link using `netem` to emulate the low bandwidth of URLLC. Since the client must remain physically plugged in to a wired network for emulating URLLC, this setting allows us to study performance only in stationary conditions.

4.2 Emulated-eMBB Setting

To evaluate DChannel under a wide variety of scenarios, specifically those including client mobility, we used *trace-*

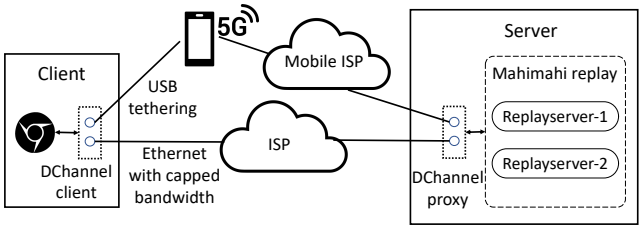


Figure 2: *In our live 5G eMBB testbed, the client has two paths to the server: One path over a tethered connection to a 5G phone for utilizing the eMBB channel, and the other through a bandwidth-capped connection over Ethernet, for emulating the URLLC channel.*

driven emulations. Below, we describe how we captured the network (latency and bandwidth) traces of the 5G eMBB channel under stationary and low-to-moderate mobility scenarios and used them in our emulations.

4.2.1 Collecting network traces

To capture the temporal variability of mobile networks, we measured both the latency and throughput of the eMBB channel over time.

Latency traces. We measured the latency of the eMBB channel by periodically sending probes (UDP packets) from the client to the server. We set the probing period to 15 ms to force the UE radio to remain always in "active" mode and generate only a small amount of probe traffic to avoid queuing. Our measurements capture the latency imposed by the base station and core network, since our server was always in close proximity to the client (i.e., less than 150 miles), minimizing the Internet-path latency. Our `traceroute`s from the client to the server, although not shown in the paper, also confirmed that the latency between the client and the server was very close to the latency between the client and the packet gateway.

Bandwidth traces. We measured the throughput across time of both uplink and downlink channels by saturating them with MTU-sized UDP packets. Since TCP cannot reliably saturate the highly variable cellular uplink and downlink concurrently, we used an overestimated fixed sending rate to always fill the queue. First, we measured the maximum supported upload and download UDP throughput using existing tools such as `iperf`. Then, we sent traffic at this maximum rate from both endpoints. Finally, we used the actual packets received over time by the endpoints to estimate the uplink and downlink capacities.

Measuring both latency and bandwidth. A key challenge in measuring both latency and bandwidth simultaneously is avoiding interference: bandwidth-intensive operations can saturate the link and fill the queue, thereby inflating the latency. Since cellular networks use per-user queues, we addressed this challenge by measuring latency and bandwidth from separate devices. When using two separate devices, we did not see any perceivable interference for measurements on 5G low-band, although we observed them on 5G mmWave. Specifically, we observed inflation in latency if a nearby device was uploading data at more than 5 Mbps using mmWave.[4] For 5G mmWave, we measured, hence, only the downlink throughput over time; we set the uplink bandwidth to a single, fixed rate of 60 Mbps.

The accuracy of temporal variations in latency matters *most* for our trace-driven emulations, since the main applications that we use in our evaluations, web browsing and mobile apps, are latency-sensitive. The performance of such applications crucially depends on TCP-related configurations (e.g., initial congestion window) and network latency (or RTT) rather than on available bandwidth, particularly when the bandwidth is more than 16 Mbps [45]. Our approach to estimating bandwidths, therefore, is adequate for our evaluations.

4.2.2 Emulating the traces

In the emulated-eMBB setting, we run both the client and the server on the same machine. DChannel then steers traffic between them over two virtual interfaces, emulated using an extended version of Mahimahi [36]. Specifically, we extended Mahimahi's delay shell to vary the eMBB channel latency over time, based on a trace generated from a real 5G deployment. The modified delay shell accepts a trace comprising a "timeline" of RTT values and halves each value to derive the individual uplink and downlink latency timelines. The shell then assigns per-packet latency by choosing an uplink or downlink latency by matching the time a packet arrives at the interface against the timelines. Since the trace-file granularity is one RTT sample per 15 ms, we use linear interpolation for assigning RTTs arriving between two samples. Similarly, we emulated URLLC with a propagation delay of 5 ms and bandwidth of 2 Mbps, unless noted otherwise.

Mobile applications' traffic (especially web browsing) is typically bursty in nature and contains periods of inactivity. To preserve energy during *idle* periods, UEs switch to a low-power (or "sleep") state, which supports discontinuous reception (DRX). The transition to the low-power state depends on an *inactivity timer* that we observed (through probing [35]) to be around 30 ms for 5G mmWave; once the device enters this state, it will "wake up" periodically (every 40 ms). When emulating the latency traces, we therefore also estimate the radio power states of the device (based on its activity) and take into account any additional latency the state transitions may impose. A packet that arrives 20 ms after the UE enters the sleep state, for instance, will experience an additional 20 ms delay before it is processed. This delay, however, is not incurred on the uplink. For 5G low-band, we set the inactivity timer to 100 ms and wake-up interval to 20 ms.

For the bandwidth emulation, we extended Mahimahi's link shell to emulate a time-varying bandwidth that changes every second. To emulate a link of capacity 60 Mbps at time

[4] Low-band uses OFDMA so multiple devices can communicate at the same time and the latency is not inflated, while mmWave uses single carrier modulation, where multiple devices must take turns transmitting and the antenna must switch its beam pattern.

Table 1: *Characteristics of network traces gathered from actual 5G deployments at different locations and under different conditions. 'p50' and 'p98' refer to the 50th and 98th percentiles, and 'CV' refers to the coefficient of variation.*

Trace name	***Span (mins.)***	***RTT (ms)***					***Mean bw. ↑/↓ (Mbps)***	***Description***
		min.	***p50***	***p98***	***mean***	***CV***		
mmWave-Stationary (MM-S)	60	18	22	106	29.88	0.77	60/140	UE was in a building in the downtown Chicago, placed near a window with a base station in line of sight.
mmWave-Walking (MM-W)	56	16	22	120	30.32	0.98	60/110	UE was held by user walking in downtown area of Chicago.
mmWave-Driving (MM-D)	18	18	40	236	56.15	0.96	60/100	Phone was with a user driving through the downtown area of Chicago at low to moderate driving speeds.
LowBand-Stationary (LB-S)	60	34	40	132	45.20	0.50	26/93	Phone was located in a building in a university campus. It was placed near a window with full signal strength.
LowBand-Walking (LB-W)	53	32	52	156	58.94	0.50	21/63	Phone held by user walking in a university campus.
LowBand-Driving (LB-D)	23	34	54	202	68.84	0.62	15/57	Phone was with a user driving near a university campus.

n seconds, for instance, this extended link shell will release 7.5 KB per millisecond. In our emulation tests, we also used a FIFO (drop-tail) queue, and we set the buffer to 800 MTU-sized packets.

5 Evaluation

We evaluated DChannel using 5G eMBB and URLLC as HBC and LLC, respectively. We ran the client (e.g., a web browser) on a laptop, unless otherwise mentioned. The laptop had 16 GB RAM, 512 GB SSD, and an Intel Core i7 processor running Ubuntu 20.04 (Focal Fossa).

5.1 Testbed Configuration

In the live-eMBB setting, we tethered the laptop with a Google Pixel 5 phone using USB (refer Fig. 2). We ran the live experiments from two locations: UIUC campus with access to 5G low-band and the Chicago downtown area for 5G mmWave access. We emulated the URLLC link between the client and server using a wired (Ethernet) link and configured it based on URLLC end-to-end specification and use-cases [6]. The emulated link provides 5 ms RTT between the client and the network gateway and has 2 Mbps capacity. At the 5G mmWave test site, however, the wired link only provided a minimum latency of 8 ms for the URLLC emulation.

We also collected latency and bandwidth traces (summarized in Tab. 1) at the two test locations under three mobility conditions: stationary, walking, and driving. All traces were collected using Google Pixel 5 phones with Verizon 5G. Though mmWave offers lower latency (for eMBB) than low-band, it also experiences higher variance than low-band, even when the UE was stationary. This inconsistency in performance becomes even worse under mobility. Low-band offers a stable, albeit relatively higher, RTT than mmWave. We ran all the components, i.e., the client, DChannel's modules, and the server, on the laptop in the emulated-eMBB setting, and used the traces (in Table 1) for emulating the eMBB channel.

5.2 Application use cases

We evaluated DChannel on 5G under a wide variety of network conditions to highlight its benefits for web browsing and web-based mobile (Android) applications. We supplemented these experiments with a bulk-download application for demonstrating DChannel's merits for long (i.e., bandwidth-intensive) flows.

Web browsing. To measure the improvements brought about by DChannel for web browsing, we first fetched a set of 200 web pages of "popular" websites, selected uniformly at random from the Hispar corpus [16]. The sample comprised 40% of landing and 60% of internal pages from 200 websites. The median web page size and the number of objects are 3.7 MB and 60, while the 95th percentile are 11.8 MB and 168. When fetching these pages, we recorded all the HTTP requests and responses using mitmproxy [1]. Then we used a version of Mahimahi with HTTP/2 support [53] to serve the responses from our server. While recording the pages, we also estimated the server response time for each request by subtracting the *time-to-first-byte (TTFB)* from the client-to-server RTT. We used the server-response times to emulate server-side processing delays during the replays.

We used an unmodified Chromium browser spawned within a DChannel shell to fetch the pages from our server. We cleared the browser and DNS caches prior to each fetch and used the default Linux congestion control, TCP CUBIC, unless noted otherwise, for all web-browsing experiments. We measured the *page-load time (PLT)*, based on the `onLoad` event [37] in each experiment, on each fetch. In the live-eMBB setting, we first used the DCHANNEL scheme to fetch a page and repeated that page fetch in quick succession using a different scheme. We calculated the difference in PLT between the different schemes and repeat the fetch five times to compute the mean difference in PLTs. In the emulated-eMBB setting, we picked a random sub-sequence from a trace for each page fetch. Given a page, we used the same sub-sequence for measuring the PLT across different schemes

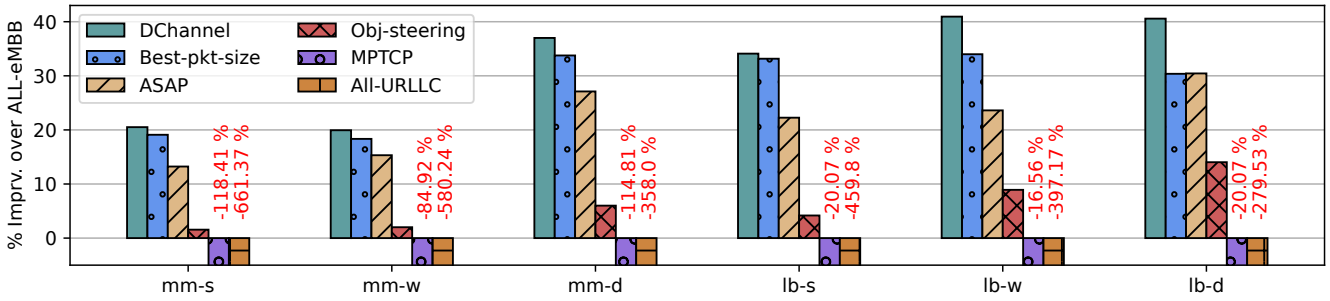


Figure 3: *DChannel offers at least 20% lower PLT compared to that of All-eMBB, and it performs better than all other schemes. MPTCP's PLTs are 17% to 118% worse than when using a single eMBB channel across all traces.*

and computed the difference in PLTs. We used the mean of the PLT differences across five trials, with each using a random sub-sequence of the trace, for comparing and contrasting the different schemes.

Mobile application. We downloaded three popular Android applications from the Google play store, one each from three categories: social (Reddit), shopping (eBay), and news (CNN). We ran the applications on the phone (Google Pixel 5), but tunneled all network traffic to a DChannel shell running on our laptop. We then used the DChannel to steer the traffic over the appropriate channels. We evaluated the mobile applications only under the emulated-eMBB setting. Note that this setup may underestimate the performance improvements because of the additional overheads in tunneling the traffic from the phone to the laptop. We measured the RTT from our laptop to the application server to be 12 to 30 ms, which is significant given that our emulated URLLC and eMBB RTT can go as low as 5 ms and 16 ms.

To evaluate application performance, we calculated the *interaction response time (IRT)* [40]. IRT measures the time elapsed between when a user performs an interaction to when the end screen for that interaction is completely rendered. We automated user interactions with AndroidViewClient [13] and cleared application caches within each trial. We recorded the phone screen using `FFmpeg` [14] during each interaction and identified when the screen stopped changing visually using `scenecut-extractor` [12]. Since we do not control the servers used by the application, we repeated the interaction experiments with DChannel and other schemes, one after the other, in quick succession. We used the median IRT across ten trials to compare the performance of different schemes.

Bulk download. The setup for this use case is similar to that for web browsing. We simply used `curl` as the client to download a file hosted in our (Mahimahi) server and repeated the download five times.

5.3 Comparing steering schemes

We compared the PLT of DChannel's DCHANNEL scheme with that of five other schemes across multiple scenarios (using the traces in Tab. 1) in the emulated-eMBB setting.

Table 2: *Comparing the performance of DChannel with All-eMBB and All-URLLC when fetching the (*182 KB*) landing page of* `amazon.com`.

Perf. metric	*All-eMBB*	*All-URLLC*	*DChannel*
DNS lookup (ms)	44	8	8
TCP connect (ms)	42	6	6
TLS connect (ms)	53	30	30
Object transfer (ms)	209	809	144
Total load time (ms)	349	853	189

The other schemes are as follows. **All-URLLC** steers all traffic over URLLC. **Obj-steering** requests web objects (requests and responses) on URLLC whenever its fetching time is smaller than eMBB. **Best-pkt-size** steers packets whose size is lower than the *best* predefined threshold to URLLC. **MPTCP** uses the two channels concurrently, by running the Linux kernel implementation of MPTCP [2] with the default configuration. **ASAP** [29] was designed for satellite networks. It diverts traffic from satellite links (or eMBB in our case) to 3G or 4G (or URLLC) since the latter has lower latency with a higher cost per byte than the former. ASAP code is not publicly available, so we used our in-house version.

Fig. 3 compares the relative difference in PLT of each scheme compared to that obtained when only using eMBB (i.e., All-eMBB). DChannel offers the best performance under all network conditions in our emulations. In stationary scenarios (MM-S and LB-S), it improves PLT by about 20% when using 5G mmWave and 35% with 5G low-band. These improvements in 5G mmWave and low-band correspond to absolute reductions in PLTs of about 290 ms and 642 ms, respectively, compared to the All-eMBB scheme. Per Fig. 3, DChannel's performance increases in scenarios that involve UE mobility (MM-W, MM-D, LB-W, and LB-D). The PLT improvements are higher for low-band than mmWave since the former exhibits higher latency than mmWave (refer Tab. 1). We examined the relation between DChannel's performance and eMBB latency in detail in §A.3.1.

Where do DChannel's gains come from? To illustrate the an-

swer to this question, we used `curl` to fetch the landing page (i.e., the root document, "/") of `amazon.com` in the emulated-eMBB setting (40 ms RTT and 200 Mbps) without any server-side response delay. DChannel performs better than not only all-eMBB, but also all-URLLC (Tab. 2). While URLLC has lower latency than eMBB, it also has a significantly lower bandwidth than eMBB. We identified three key sources of performance gains by analyzing DChannel's per-packet decisions. DChannel steers three types of packets over URLLC: (1) DNS packets; (2) *control* packets (e.g., `SYN` and client-to-server `ACK` packets); and (3) small *data* packets. Sending DNS and SYN packets over URLLC reduces DNS lookup and TCP connection-setup times, and accelerating ACK packets reduces the object transfer time. The last category includes small data transfers such as the TLS client-key exchange and HTTP requests.

Packet vs. web objects steering. Obj-steering performs HTTP request and response on URLLC when the web object size is small such that it will finish faster than eMBB. The scheme only offers slight improvement to PLT (2-14%). This is because not all small objects are critical. In fact, we found out that only 14% of small web objects have VeryHigh priority [25].

DChannel vs. static packet-size-based steering. Best-pkt-size steers individual IP packets whose size is smaller than the *best* static threshold to URLLC, and the reordering buffer will reorder any out-of-order packets. To find the best threshold, we performed web page load experiments for each trace with five different (size) thresholds: 250, 500, 750, 1000, 1250, and 1400 bytes. We found 750 bytes and 1000 bytes give the best-averaged result (across the stationary, walking, and driving scenarios) for mmWave and Low-Band traces, respectively.

DChannel shows an overall better improvement than best-pkt-size across different network conditions, albeit best-pkt-size offers similar improvements in stationary traces. In stationary traces, network latency is more predictable, and static decisions might suffice. When the network conditions are more variable, such as in the driving scenario, however, the static decisions do not suffice. DChannel observes network conditions, as they evolve, and estimates URLLC channel usage to make steering decisions dynamically. DChannel will not steer small packets to URLLC, for instance, if the channel is already congested. Note that these results are overly generous to best-pkt-size, for comparison purposes: the best size is selected in retrospect after running on the test scenarios. In reality, determining a single packet size threshold would be complicated: it depends on application traffic patterns as well as network conditions.

Is MPTCP not designed to exploit multiple channels? MPTCP works at the transport layer, and in general, it load-balances application traffic among the available paths and aggregates their throughput. In our evaluations, MPTCP performs worse (by inflating PLTs between 16% and 66% across different conditions[5]) than simply using only the eMBB path. This poor performance stems from MPTCP's default scheduler (*minRTT*), which prefers the path with the smallest estimated RTT. This scheduler thus infers that URLLC is better than eMBB and diverts traffic to URLLC until experiencing congestion. DChannel, unlike MPTCP, works at the network layer such that it allows steering data packets on eMBB and ACK packets on URLLC. MPTCP cannot perform such packet-level steering, since it results in each path having a separate data-ACK loop, which MPTCP cannot support.

Table 3: *The p50 and (p95) of the avg. and max. buffer sizes (in bytes) when loading 200 web pages under MM-S and LB-D traces.*

	MM-S		***LB-D***	
	Proxy	***UE***	***Proxy***	***UE***
Avg buffer size (b)	2 (15,7)	12 (63.5)	14 (96)	130 (2638)
Max buffer size (b)	392 (1122)	944 (2597)	757 (2375)	2848 (15521)

DChannel vs ASAP. ASAP identifies the different phases of a web transaction (e.g., TLS handshake and HTTP request) and accelerates packets of latency-sensitive phases. It accelerates, for instance, TLS/SSL handshake as well as HTTP request traffic, but leaves HTTP responses to eMBB. ASAP performs better than all other schemes except DChannel. It falls behind DChannel, however, because of its static heuristics (e.g., accelerate all HTTP requests). HTTP requests are typically, but not always, small. A user uploading a photo, for instance, is one example where the assumption fails to hold. ASAP also encounters problems when the user browses complex internal pages that push some data to the server.

In the above experiments, we emulated URLLC based on the 5G standard. We found, however, that DChannel continues to offer significant PLT improvements even if URLLC latency is doubled or tripled or when URLLC latency changes over time (§A.3.2). DChannel offers good performance even in situations we cannot accurately estimate the eMBB RTT (§A.3.4), which is crucial for calculating rewards and cost (§3.4). We also examined DChannel's performance under different URLLC bandwidths in §A.3.3. Finally, we evaluated DChannel's rewards calculation accuracy in §A.4

5.4 Live 5G Experiments

We repeated the web-page fetches (similar to those in §5.3) over both the live-eMBB and emulated-eMBB settings. We then compared the relative improvements in PLTs brought about by DChannel across these settings, for both 5G mmWave and Low-Band (Fig. 4). In conclusion, the PLT improvements are quite similar between the live and

[5]We clipped the bottom of the Y-axis in Fig. 3 to focus on performance gains.

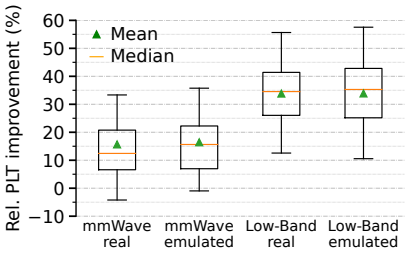


Figure 4: *Comparison of relative PLT improvements in live-eMBB and emulated-eMBB settings*[6].

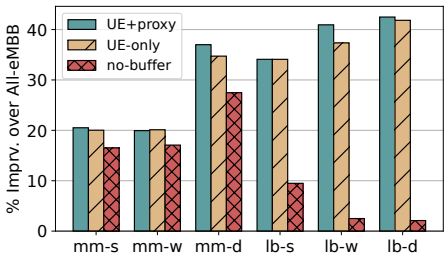


Figure 5: *Adding reorder buffer to DChannel significantly improves web page load time.*

emulated-eMBB settings, albeit the mean PLTs in the former are higher than those in the latter. The mean PLT for the All-eMBB scheme in the live-eMBB setting for 5G Low-Band is 1718 ms, while that in the emulated-eMBB setting is 1522 ms. The high mean PLTs in the emulated-eMBB settings could be due in part to the limitations of the setup. The emulations do not capture all eMBB network characteristics such as out-of-order delivery. Moreover, the web server is hosted on dedicated hardware in the live-eMBB setting, whereas in the emulated-eMBB setting it runs on the same laptop as the client. These minor setup differences, however, do not affect our claims concerning the relative performance improvements (which are similar across the two settings).

5.5 Evaluating the reordering buffer

We evaluated the effectiveness of the reordering buffer (ROB) by comparing the web browsing performance of DChannel with and without the buffer. Along with the other experiments, this evaluation uses the default TCP CUBIC, which is sensitive to in-order packet delivery. Fig. 5 shows that without the buffer, the mean PLT improvement is decreased by up to 40% (in LB-D). Overall, DChannel without the buffer will be worse when the gap between eMBB and URLLC latency increases (implied in the figure as DChannel without the buffer is performing much worse in 5G Low-Band than mmWave) because DChannel will be more aggressive in offloading packets to URLLC, causing more out of order packets.

If ROB is implemented only on the UE (downlink), PLT improvement is only reduced by 2%. The decrease in PLT improvement is because DChannel tends to offload most web browsing uplink traffic to URLLC as the client requests are generally small such that minimal out-of-order persists. Our buffer analysis in Tab. 3 confirms that DChannel proxy does not use the buffer as frequently as the DChannel client (UE). Tab. 3 also shows that ROB requires only little memory as we do not have to buffer much URLLC traffic.

DChannel under random packet drop. Since ROB holds packets from URLLC for a fixed (100 ms) timeout when they are not in order, DChannel may suffer when packet loss happens as it may delay the packet loss signal, which can happen under certain conditions (§3.6). To evaluate this effect, we

[6]The mean PLT improvements of mmWave is lower than what we reported on MM-S in Fig.3 because we used 8 ms of URLLC RTT (rather than 5 ms).

Table 4: *PLT under different random packet drop rates.*

Loss	*All-eMBB (ms)*		DCHANNEL *(ms)*	
	MM-S	*MM-D*	*MM-S*	*MM-D*
0.0%	1108	1899	883 (20%)	1096 (42%)
0.1%	1203	1963	1011 (16%)	1311 (34%)
1.0%	2643	3421	2502 (5%)	3072 (10%)

investigated DChannel performance under stationary and driving traces in the case of random packet drops. We applied a stochastic packet drop in the uplink and downlink channels with packets being dropped in both eMBB and URLLC. This reflects the case where packet drops happen on the Internet path. Note that 5G eMBB generally exhibits low packet loss rates with the 99th percentile being 1.2% [34]. Per Tab. 4, DChannel is quite resilient to random packet loss, offering better performance than All-eMBB even under high loss rates. We also investigated the interplay between DChannel and latency-based congestion control algorithms in Appendix A.3.5.

5.6 Bulk download performance

Although our primary focus is latency-sensitive applications, we also evaluated how DChannel performed for a bandwidth-intensive use–bulk HTTP transfer of a file. Fig. 6 shows the download time for various file sizes using the mmWave driving (MM-D) trace. As expected, DChannel gets the largest improvement for small objects. But interestingly, DChannel also usefully improves large object download. This is because all control packets including TCP ACKs are accelerated in URLLC, which reduces the control loop delay, helping the transport layer adapt to bandwidth changes more quickly. Specifically, we found DChannel resulted in better utilization of the available eMBB throughput by $\approx 10\%$ when there are large throughput variations (e.g., in the driving scenario).

5.7 Mobile application performance

Fig. 7 shows the application response time improvement of DChannel across three common user interactions that require communication to the server. On average, DChannel improves the response times of application launch (15%), query searching (12%), and information (e.g., product or news) loading (21%). It is unsurprising that query searching gives lower improvement since it incurs higher server-side delay. The overall

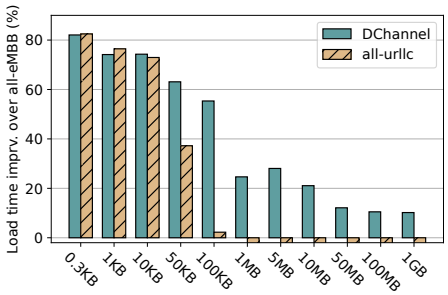


Figure 6: *Download time improvement of variable-sized data under HTTP. The experiment used the MM-D trace with the buffer set to 800 packets (≈ 2× trace BDP).*

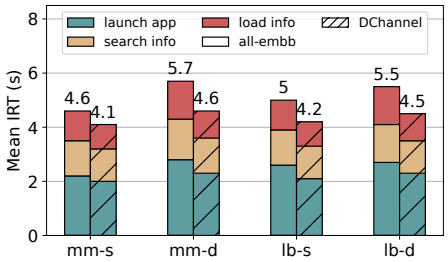


Figure 7: *Android mobile application interaction response time (IRT) of All-eMBB and DChannel when performing three different tasks. We averaged the result from three applications.*

mobile apps improvement is lower than the web browsing in part to our experimental setup (§5.2).

6 Discussions and Future Work

Deployment for 5G networks: DChannel requires cellular operator support, to allow URLLC for non-critical traffic and to perform stateful packet steering. However, operators may omit some of the DChannel implementations to make deployment easier, such as eliminating the reordering buffer (ROB) on the core gateway since DChannel shows only minor performance degradation without ROB in the proxy-side (§5.5). DChannel stateful packet steering may not be simple to implement especially when there are multiple gateways. We leave this to future work.

URLLC scalability: The number of users that can send general traffic to URLLC is an important matter which deserves to be evaluated quantitatively in the future. At the time of writing, URLLC is not yet deployed in public. However, based on the white paper [7], URLLC is targeted to support a relatively high connection density with modest per-user bandwidth. For instance, one of the URLLC use cases (discrete automation) requires a user-experienced data rate of 10 Mbps, traffic density of 1 Tbps/km^2, connection density of 100,000/km^2, and max end-to-end latency of 10ms. Thus, the 2 Mbps maximum bandwidth per user for general application traffic used in our experiments is still reasonable based on others' proposed use cases for URLLC, even in a dense urban area.

Disrupting URLLC native traffic: URLLC is primarily built to serve latency-sensitive critical applications. To ensure we do not compromise the performance of these applications, the network operator can limit the per-user bandwidth and even choose to deprioritize non-critical packets as our approach does not require 99.999% reliability and is resilient to small increases in URLLC latency (§A.3.2).

Resource contention among applications: Multiple applications inside a user device may compete to use URLLC. We can regulate them using prioritization. One simple approach is to prioritize applications running in the foreground since mobile phone users are typically single-tasking.

Incentives for operators: While URLLC targets critical applications, it is up to the network providers to open URLLC for general mobile applications like web browsing. This is possible as 5G chipsets are typically designed to support multiple bands including the sub-6GHz bands for URLLC [6]. Expanding URLLC applications can encourage providers to foster a faster and broader deployment of URLLC as it brings a smoother experience to their major customers – mobile phone users; especially as the current market for URLLC applications like self-driving cars and remote surgery is still in its infancy.

Emulation uncertainty: The real URLLC performance might not match our emulated URLLC that follows the 5G NR white paper. However, we have performed several experiments to show the robustness of DChannel under variable URLLC conditions. Emulating the real behavior of a cellular network (eMBB) is also a known hard problem [51], and our approach of using two phones to capture both eMBB latency and bandwidth might not be perfect. We have compared DChannel performance with the emulated eMBB and live eMBB in stationary conditions and conclude that DChannel offers the same performance benefit (§5.4). However, we have not yet evaluated DChannel under non-stationary live eMBB due to the environment limitation (§4.1).

Other applications: LLC and HBC combination can also properly support applications from different domains that require high bandwidth and low latency, something that cannot be satisfied by utilizing a single channel. For instance, cloud gaming, which allows users to play games from remote servers, requires high bandwidth to stream content and low latency to remain responsive to user input. Since these applications can be vastly different than web browsing, a superior steering scheme may exist. We plan to analyze them further to determine an effective way of leveraging LLC and HBC.

Beyond mobile networks: Our insights may apply to other LLC and HBC combinations with analogous bandwidth and latency trade-offs. Examples include quality of service (QoS) differentiation providing separate latency- and bandwidth-optimized services [17, 39]; and routing traffic among multiple ISPs where one is more expensive but provides better latency, as may happen with very low Earth orbit satellite-based [24] or specialty [18] ISPs. To achieve the optimum cost-to-performance ratio, we can route only the latency-

sensitive traffic to the low-latency ISP.

Future wireless design: The 5G URLLC is only equipped with limited user bandwidth, and hence it is not suitable to serve general application traffic. The bandwidth is severely compromised because it needs to provide *both* low latency and very high reliability (99.999%). However, general applications do not need the almost-perfect reliability that URLLC guarantees. Future wireless networks (such as 6G) may reconsider this trade-off and provide a low-latency channel with somewhat greater bandwidth and somewhat lower reliability.

7 Related work

There have been multiple works that try to exploit the multi-access connectivity on the client.

Application layer multipath: Socket Intents [42] and Intentional networking [28] both expose custom APIs to applications and offer OS-level support for managing multiple interfaces. Both of them regulate application traffic based on application-specific information. Our work, in contrast, does not require application inputs or modifications, although in the future we might consider giving input to the steerer to support more specific applications.

Transport layer multipath: There are already numerous efforts to design multipath transport protocols such as R-MTP [33], pTCP [30], mTCP [52], SCTP multihoming [31], and MPTCP [49]. These protocols deliver application traffic through multiple paths to achieve better throughput and reliability. Due to the bandwidth aggregation focus, multipath transport protocols give notable benefits to long-flow dominated applications but not to short-flow dominated applications such as web browsing [20]. Our approach works transparently with single-path transport protocols (e.g., TCP and UDP).

Network layer multipath: Tsao and Sivakumar [46] proposed a super aggregation concept where TCP can achieve better WiFi throughput by selectively steering packets to 3G. ASAP [29] steers network packets over satellite ISP and lower-latency terrestrial networks to improve HTTPS. We compared DChannel against ASAP in our evaluation and found that DChannel is better for eMBB and URLLC pairs as it benefits from finer-grained decisions.

An early version of DChannel was presented in [43]. This work comes with a new and better-performing packet steering algorithm, a more robust evaluation with real-world traces and live 5G eMBB, and new use cases including mobile apps and bulk transfer.

Acknowledgements

We thanked the anonymous reviewers and our shepherd Fadel Adib for their valuable inputs. This work was supported by a gift from T-Mobile and NSF CNS Awards 1763742 and 1763841.

References

[1] mitmproxy. https://mitmproxy.org/. [Last accessed on April 18, 2022].

[2] Multipath TCP in the Linux Kernel v0.94. http://www.multipath-tcp.org, March 2018. [Last accessed on June 16, 2020].

[3] MWC: Are Your 5 Fingers Blocking Your 5G? https://www.eetimes.com/mwc-are-your-5-fingers-blocking-your-5g/, February 2018. [Last accessed on June 24, 2020.

[4] 3GPP Release 15. https://www.3gpp.org/release-15, April 2019. [Last accessed on May 24, 2020].

[5] 3GPP TR 23.725 version 16.2.0 Release 16. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3453, June 2019. [Last accessed on January 20, 2022].

[6] 3GPP TR 38.824 Release 16. https://www.3gpp.org/release-16, March 2019. [Last accessed on June 16, 2020].

[7] 3GPP TS 22.261 version 15.7.0 Release 15. https://www.etsi.org/deliver/etsi_TS/122200_122299/122261/15.07.00_60/ts_122261v150700p.pdf, March 2019. [Last accessed on January 20, 2021].

[8] 3GPP Release 16 Description; Summary of Rel-16 Work Items. https://www.3gpp.org/release-16, March 2020. [Last accessed on June 16, 2020].

[9] AT&T: 5G Coverage Map. https://www.att.com/5g/coverage-map/, 2020. [Last accessed on June 13, 2020].

[10] T-Mobile: The Only Nationwide 5G Network Coverage Map. https://www.t-mobile.com/coverage/5g-coverage-map, 2020. [Last accessed on June 13, 2020].

[11] Verizon: 5G Coverage Map. https://www.verizon.com/5g/coverage-map/, 2020. [Last accessed on June 13, 2020].

[12] Scenecut extractor. https://github.com/slhck/scenecut-extractor, December 2021. [Last accessed on April 15, 2022].

[13] AndroidViewClient. https://github.com/dtmilano/AndroidViewClient, March 2022. [Last accessed on April 15, 2022].

[14] FFmpeg. https://ffmpeg.org/, January 2022. [Last accessed on April 15, 2022].

[15] 3rd Generation Partnership Project. Study on scenarios and requirements for next generation access technologies. Technical report, 2017.

[16] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M Maggs. On landing and internal web pages: The strange case of jekyll and hyde in web performance measurement. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[17] Jozef Babiarz, Kwok Chan, and Fred Baker. Configuration guidelines for diffserv service classes. *Network Working Group*, 2006.

[18] Debopam Bhattacherjee, Waqar Aqeel, Sangeetha Abdu Jyothi, Ilker Nadi Bozkurt, William Sentosa, Muhammad Tirmazi, Anthony Aguirre, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. cISP: A Speed-of-Light Internet Service Provider. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[19] Eduardo Cuervo. Beyond reality: Head-mounted displays for mobile systems researchers. *GetMobile: Mobile Computing and Communications*, 21(2), 2017.

[20] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. WiFi, LTE, or both? Measuring multi-homed wireless internet performance. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2014.

[21] Yoav Einav. Amazon found every 100ms of latency cost them 1% in sales, January 2019.

[22] A El Gamal, James Mammen, Balaji Prabhakar, and Devavrat Shah. Throughput-delay trade-off in wireless networks. In *IEEE INFOCOM 2004*, volume 1, 2004.

[23] M. Giordani, M. Polese, A. Roy, D. Castor, and M. Zorzi. A Tutorial on Beam Management for 3GPP NR at mmWave Frequencies. *IEEE Communications Surveys & Tutorials*, 21(1), 2019.

[24] Giacomo Giuliari, Tobias Klenze, Markus Legner, David Basin, Adrian Perrig, and Ankit Singla. Internet backbones in space. *ACM SIGCOMM Computer Communication Review*, 50(1), 2020.

[25] Sergio Gomes. Resource prioritization – getting the browser to help you. https://developers.google.com/web/fundamentals/performance/resource-prioritization, June 2020. [Last accessed on June 12, 2020].

[26] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76. IEEE, 2016.

[27] Haitham Hassanieh, Omid Abari, Michael Rodriguez, Mohammed Abdelghany, Dina Katabi, and Piotr Indyk. Fast Millimeter Wave Beam Alignment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[28] Brett D Higgins, Azarias Reda, Timur Alperovich, Jason Flinn, Thomas J Giuli, Brian Noble, and David Watson. Intentional networking: opportunistic exploitation of mobile network diversity. In *Proceedings of the 16th annual international conference on Mobile computing and networking (MobiCom)*, 2010.

[29] Se Gi Hong and Chi-Jiun Su. ASAP: fast, controllable, and deployable multiple networking system for satellite networks. In *IEEE Global Communications Conference (GLOBECOM)*, 2015.

[30] Hung-Yun Hsieh and Raghupathy Sivakumar. pTCP: An end-to-end transport layer protocol for striped connections. In *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP)*, 2002.

[31] Janardhan R Iyengar, Paul D Amer, and Randall Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Transactions on networking (ToN)*, 14(5), 2006.

[32] Adrian Loch, Irene Tejado, and Joerg Widmer. Potholes Ahead: Impact of Transient Link Blockage on Beam Steering in Practical mm-Wave Systems. In *The 22nd European Wireless Conference*, May 2016.

[33] Luiz Magalhaes and Robin Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *Proceedings of the 9th IEEE International Conference on Network Protocols (ICNP)*, 2001.

[34] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A First Look at Commercial 5G Performance on Smartphones. In *Proceedings of The Web Conference*, 2020.

[35] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. A variegated look at 5G in the wild: performance, power, and QoE implications. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2021.

[36] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.

[37] Jan Odvarko. Har 1.2 spec, 2007.

[38] Tommy Pauly, Brian Trammell, Anna Brunstrom, Gorry Fairhurst, Colin Perkins, Philipp S Tiesel, and Christopher A Wood. An architecture for transport services. *Internet-Draft draft-ietf-taps-arch-00, IETF*, 2018.

[39] Maxim Podlesny and Sergey Gorinsky. RD network services: differentiation through performance incentives. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2008.

[40] Murali Ramanujam, Harsha V Madhyastha, and Ravi Netravali. Marauder: synergized caching and prefetching for low-risk mobile app acceleration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2021.

[41] Sanae Rosen, Haokun Luo, Qi Alfred Chen, Z Morley Mao, Jie Hui, Aaron Drake, and Kevin Lau. Discovering fine-grained RRC state dynamics and performance impacts in cellular networks. In *Proceedings of the 20th annual international conference on Mobile computing and networking (MobiCom)*, 2014.

[42] Philipp S Schmidt, Theresa Enghardt, Ramin Khalili, and Anja Feldmann. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2013.

[43] William Sentosa, Balakrishnan Chandrasekaran, P Brighten Godfrey, Haitham Hassanieh, Bruce Maggs, and Ankit Singla. Accelerating mobile applications with parallel high-bandwidth and low-latency channels. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 1–7, 2021.

[44] M Zubair Shafiq, Lusheng Ji, Alex X Liu, Jeffrey Pang, and Jia Wang. Characterizing geospatial dynamics of application usage in a 3G cellular data network. In *Proceedings IEEE INFOCOM*, 2012.

[45] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, and Nazanin Magharei. Measuring and mitigating web performance bottlenecks in broadband access networks. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2013.

[46] Cheng-Lin Tsao and Raghupathy Sivakumar. On effectively exploiting multiple wireless interfaces in mobile hosts. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[47] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with wprof. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[48] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How Speedy is SPDY? In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[49] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[50] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.

[51] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.

[52] Ming Zhang, Junwen Lai, Arvind Krishnamurthy, Larry L Peterson, and Randolph Y Wang. A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths. In *USENIX Annual Technical Conference (ATC)*, 2004.

[53] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, and Klaus Wehrle. Is the web ready for http/2 server push? In *Proceedings of the 14th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2018.

[54] Baiqing Zong, Chen Fan, Xiyu Wang, Xiangyang Duan, Baojie Wang, and Jianwei Wang. 6g technologies: Key drivers, core requirements, system architectures, and enabling technologies. *IEEE Vehicular Technology Magazine*, 14(3), 2019.

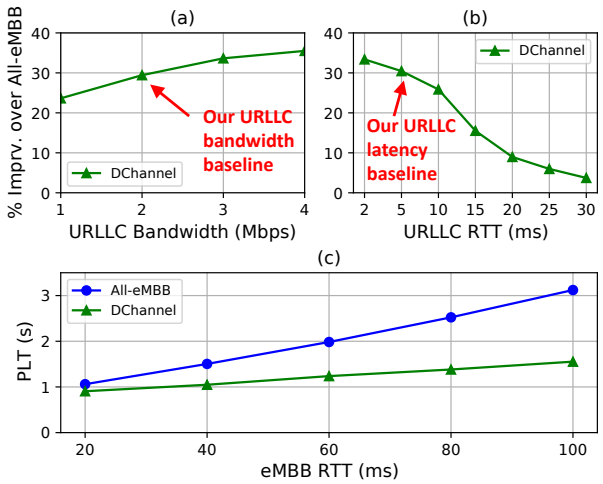


Figure 8: *DChannel under varying eMBB and URLLC network conditions, conducted using the baseline (RTT (ms) / bandwidth (Mbps) for eMBB and URLLC are 40 / 200 and 5 / 2) with a single varying parameter.*

A Appendix

A.1 Algorithm Listing

The packet steering algorithm is listed as Algorithm 1. Note that since HBC bandwidth (B_{hbc}) is typically large and relatively hard to measure, for simplicity, we omitted the queueing delay caused by $(size(P_n) + Q_{hbc}(t_n))/B_{hbc}$.

Algorithm 1: DChannel steering algorithm

```
Result: Send a packet to either HBC or LLC
c_llc = t_now + llc_prop + (size(packet) + queue_llc) /
  band_llc;
c_hbc = t_now + hbc_prop;
rewards = c_hbc - max(prev_c, c_llc);
cost = (size(packet) + queue_llc) / band_llc;
if rewards > alpha * cost then
    send(pkt, LLC);
    prev_c = max(prev_c, c_llc);
else
    send(pkt, HBC);
    prev_c = max(prev_c, c_hbc);
end
```

A.2 Parameter Calibration

The results of our parameter sweep are shown in Table 5.

A.3 Understanding DChannel Performance

Below, we investigated how DChannel performs under tightly controlled network variables. We used a fixed network latency and bandwidth for the experiments below.

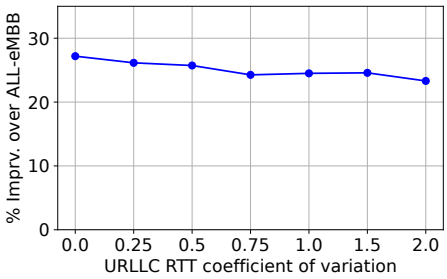


Figure 9: *DChannel PLT improvement under time-varying URLLC RTT randomly generated according to a Gaussian distribution.*

A.3.1 Performance under high eMBB RTT

We evaluated DChannel under eMBB RTT inflation and found it to be resilient towards the RTT increase (Fig. 8c). Specifically, DChannel is 2× faster than the baseline when eMBB RTT is held at 100 ms, which is possible in device mobility as Tab. 1 shows. As eMBB RTT increases, DChannel web PLT degrades at a slower rate compared to All-eMBB because it uses eMBB primarily for bandwidth-sensitive (low rewards) traffic that is not affected as severely by the increased latency. Inline with our trace-based evaluation under mobility, the parallel channel setup can be extremely effective when the eMBB channel quality degrades.

A.3.2 Varying URLLC latency

Although URLLC is expected to deliver consistent low latency, we evaluated latency inflation on URLLC to reflect scenarios where web traffic is de-prioritized in favor of critical traffic. DChannel is still superior even when URLLC latency increases up to 30 ms and eMBB held at 40 ms (Fig. 8b). As expected, as URLLC latency increases and becomes closer to eMBB, the PLT improvement rapidly diminished because LLC no longer offers a competitive resource despite its higher cost.

To evaluate DChannel sensitivity to URLLC latency instability, we changed URLLC propagation latency over time to random values that follow Gaussian distribution. We kept the URLLC mean to 10 ms and ran experiments with an increasing amount of variation, controlled with the Gaussian distribution's coefficient of variation (CoV). We set the URLLC bandwidth to 2 Mbps while the eMBB RTT and bandwidth are 40 ms and 200 Mbps. We found that DChannel performance is relatively robust to the URLLC latency change (Figure 9); the PLT improvement only drops from 27.18% to 23.31% when the URLLC latency CoV changes from 0 to 2.

A.3.3 Varying URLLC bandwidth

The URLLC bandwidth is generally limited to ensure its reliable and low latency service. We tested DChannel under varying URLLC and summarize that its improvement flattens as URLLC bandwidth increases past 2 Mbps (Fig. 8a). As

Table 5: *Percentage of improvement or 'speedup' in PLT (%ps.) along with the percentage of bytes (%sz.) sent via URLLC for various values of the eMBB channel RTT. The table also shows how the different values of* α *affect the performance benefits.*

RTT	α = 0		α = 0.25		α = 0.5		α = 0.75		α = 1		α = 2		α = 3	
(ms)	***%ps.***	***%sz.***	***%ps.***	***%sz.***	***%ps.***	***%sz.***	***%ps.***	***%sz.***	***%ps.***	***%sz.***	***%ps.***	***%sz.***	***%ps.***	***%sz.***
20	16.7	13.3	16.6	10.7	18.4	5.9	18.9	5.7	18.5	5.5	15.6	4.7	14.2	4.1
40	31.1	18.8	33.2	16.2	34.0	13.9	34.7	11.9	34.0	11.1	32.8	8.5	31.8	5.8
60	37.5	22.6	40.9	19.5	41.2	17.4	42.1	14.9	40.8	14.2	40.6	11.1	37.4	9.8
80	43.2	26.3	46.3	22.9	46.3	19.9	46.3	17.7	47.1	16.5	45.6	13.3	45.1	11.6
100	47.1	29.4	48.6	26.1	49.7	22.5	50.2	19.9	50.7	18.3	49.8	14.9	48.5	13.1

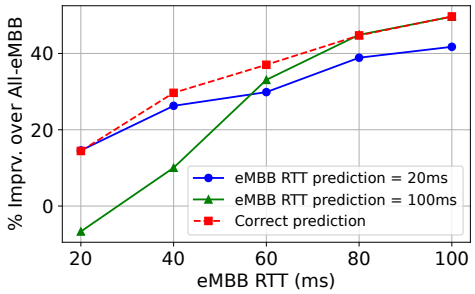


Figure 10: *Effect of underestimating and overestimating eMBB latency on the mean PLT improvement.*

URLLC bandwidth increases, DChannel aggressively offloads packets belonging to a larger transfer (e.g., HTTP response) to URLLC. It may not affect the completion time, however, as it still needs to wait for the remaining part to be transferred over the eMBB.

A.3.4 Working with Incorrect Latency Estimates

DChannel requires estimates of eMBB and URLLC latency to calculate rewards and cost (§3.4). While URLLC latency is predictable, it can be hard to get an accurate measurement of eMBB latency, especially under mobility. To better understand the sensitivity of DChannel to the latency estimates, we evaluated DChannel under underestimated and overestimated latency. Fig. 10 shows that underestimating eMBB latency is safer: When DChannel underestimates eMBB latency as 20 ms (from 100 ms, which is 5× higher), the PLT improvement only decreases by 8%. Underestimating eMBB latency will underestimate the rewards, causing a more conservative use of URLLC and ensuring that the offloaded packets are high rewards packets. Overestimating the latency will, in contrast, overestimate the rewards, resulting in unnecessary packets being offloaded to URLLC, and slowing down the channel. Per Fig. 10, overestimating eMBB latency by 5×, DChannel gets worse performance than the baseline.

A.3.5 DChannel under TCP BBR

Since DChannel steers packets over two channels, the sender may notice an abrupt change in the flow RTT. We evaluated DChannel in TCP BBR, which uses RTT measurement to determine whether a path is congested. Tab. 6 shows the result when low (RTT=20 ms) and high (RTT=100 ms) eMBB latency are used. When the eMBB RTT is 20 ms, BBR works perfectly with DChannel because eMBB and URLLC latencies are not that different. As the latency gap widens, however, BBR starts to treat abrupt latency inflation as a congestion signal, reduces its windows rate, and increases PLT. We found that for 20% of the web page loads, DChannel performs worse than All-eMBB. These sites rely on a single TCP connection to deliver most web objects, and that flow suffers from a low sending rate. This can happen as DChannel accelerates the early packets (such as TCP SYN) to URLLC and suddenly switches back to eMBB once it sees large traffic. The abrupt RTT change gives a wrong congestion signal to the sender. One possible solution is to modify BBR to be aware of eMBB and URLLC use so that it can tolerate a change in RTT and maintain its sending rate. We leave this as future work. Alternatively, we can use different heuristics (pkt-uplink) that will send all uplink packets to URLLC and downlink packets to eMBB. This heuristic is based on the observation that client traffic is generally small and accelerating those packets will accelerate the flow RTT in a more consistent way. Table 6 shows we can get 39% improvement in PLT under BBR from this scheme.

Table 6: *Mean PLT with TCP BBR under different eMBB RTTs. The URLLC RTT is set to* 5 ms

	RTT=20ms	***RTT=100ms***
All-eMBB	915 ms	2661 ms
DChannel	716 ms (21%)	2713 ms (-2%)
pkt-uplink	860 ms (6%)	1628 ms (39%)

A.4 DChannel rewards calculation accuracy

We evaluated DChannel packet rewards (R) calculation accuracy by comparing the calculated rewards and the *real* rewards. As we used a trace-based emulated network, we knew both network bandwidth, latency, and the link's queue depth at any given time. Leveraging this information, we can calculate the

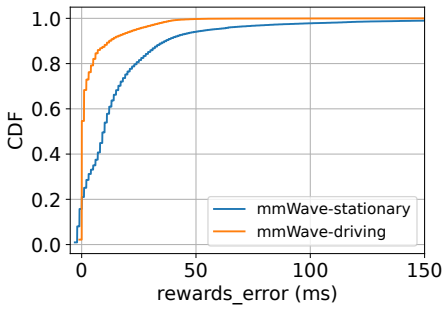


Figure 11: *DChannel calculates rewards error distribution ($R_{real} - R_{est}$) across individual packets. A positive value denotes R underestimation. We limited the x-axis due to the high rewards error in the tail (*155 ms *and* 604 ms *at the 99th and 100th percentile).*

real rewards before we commit a packet to a link. Figure 11 shows the rewards calculation error for the mmWave stationary (MM-S) and driving (MM-D) network traces. DChannel is able to accurately estimate R with just 12 ms of error in the 90th percentile in the stationary network traces due to the network latency being relatively stable and the bandwidth being large. In the driving traces, the error is noticeably higher with a long tail. However, 90% of the time the error is less than 37 ms. The R error mainly arises from the less accurate network eMBB latency estimation and the impact of ignoring the eMBB queueing delay (§A.1), since eMBB bandwidth may be low, and the delay becomes more significant).

The above is also why, as can be seen in the figure, R is rarely overestimated. Underestimation is the preferred direction of error, as we have shown that DChannel can tolerate some incorrect latency estimates and rewards underestimation (§A.3.4). However, better network measurement may improve DChannel performance; we leave this as future work.

SkyPilot: An Intercloud Broker for Sky Computing

Zongheng Yang*, Zhanghao Wu*, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj,
Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker[§], Ion Stoica
University of California, Berkeley [§]*UC Berkeley and ICSI*

Abstract

To comply with the increasing number of government regulations about data placement and processing, and to protect themselves against major cloud outages, many users want the ability to easily migrate their workloads between clouds. In this paper we propose doing so not by imposing uniform and comprehensive standards, but by creating a fine-grained two-sided market via an *intercloud broker*. These brokers will allow users to view the cloud ecosystem not just as a collection of individual and largely incompatible clouds but as a more integrated Sky of Computing. We describe the design and implementation of an intercloud broker, named SkyPilot, evaluate its benefits, and report on its real-world usage.

1 Introduction

The modern information infrastructure is built around three components. The Internet provides end-to-end network connectivity, cellular telephony provides nearly ubiquitous user access via increasingly powerful handsets, and cloud computing makes scalable computation available to all. These ecosystems obviously have many superficial differences, but perhaps their most fundamental difference lies in the degree of compatibility between providers in each of these ecosystems.

The Internet and the cellular infrastructure were designed with the goal of universal reachability. This required both uniform and comprehensive industry standards and broadly-adopted interconnection agreements (for Internet peering and cellular roaming) that led to a globally connected federation of competing providers. The cloud ecosystem has very different origins, emerging as a replacement for dedicated on-premise computing clusters rather than serving as an interconnected communication infrastructure. As a result, cloud providers began by emphasizing their differences rather than their similarities; though the clouds are all based on the same basic conceptual units (e.g., VMs, containers, and now FaaS), they initially differed greatly in their orchestration interfaces. These orchestration interfaces have become more similar over time, but some clouds continue to differentiate themselves through numerous proprietary service interfaces, such as for storage or key-value stores. In addition, clouds typically impose much higher charges on data leaving than on data entering, resulting in "data gravity" (i.e., the difficulty of moving jobs to another cloud due to the expense of transferring the data). The combination of proprietary service interfaces and data gravity have led to significant customer lock-in: it is hard for companies who have established their computational workloads on one cloud to move them to another.

However, as cloud computing has become a critical part of our computational infrastructure, enterprises are increasingly worried about how difficult it is to migrate workloads between clouds. There are two compelling reasons for wanting more freedom in workload placement. First, no business wants any critical part of their infrastructure tied to a single provider because such lock-in reduces their negotiating leverage and also makes the business vulnerable to large-scale outages at the provider. Second, there are now strict regulations about data and operational sovereignty that dictate where data can be stored and computational jobs run. Not all cloud providers have datacenters in all countries, so the inability to migrate jobs between cloud providers could be a painful roadblock to satisfying these new regulations. These two reasons are not theoretical problems whose solutions would be "nice-to-have"; the recent occurrence of large-scale cloud outages and the increasing number of government regulations are quickly making such a solution a "must-have" for large-scale users of the cloud. This paper is about how we can ease the migration of workloads through the rise of Sky Computing, a concept first introduced in [81] but significantly extended and more deeply explored here. Sky Computing is when users, rather than directly interacting with the cloud, submit their jobs to what we call *intercloud brokers* who handle the placement and oversee the execution of their jobs.

To explain our approach in more depth, we first review related concepts and recent developments (§2). We then (§3) describe our vision of Sky Computing and its transformative possibilities. We present the requirements, architecture, and

*Equal contribution.

implementation of an intercloud broker, named SkyPilot, that focuses on computational batch jobs (§4). We then demonstrate its benefits on several applications (§5). Finally, we share our experiences with early deployments (§6), survey related work (§7), and conclude (§8). While the body of this paper is devoted the technical characteristics of our system, in the appendix (§A.1) we speculate on how the cloud ecosystem might evolve once Sky Computing is more widely adopted.

SkyPilot is open source and available at `https://github.com/skypilot-org/skypilot`.

2 Related Concepts and Recent Developments

In this section we first review two concepts related to the ability to migrate workloads – standards and multicloud – and then discuss the recent progress towards compatibility.

2.1 Why Not Just Adopt Standards?

The first question one might ask is if seamless migration is the goal, why not adopt a set of uniform and comprehensive cloud standards, as was done for the Internet and cellular? In fact, a decade ago IEEE proposed a set of Intercloud standards for portability, interoperability, and federation among cloud providers [88] involving an Intercloud Service Catalog and an Intercloud federation layer. There are two fundamental problems with this and other proposals for such uniform and comprehensive cloud standards. First, there is no incentive for the dominant clouds (i.e., those with large market shares) to adopt such standards; it would decrease their competitive advantage and make it easier for customers to move their business to other clouds. Second, users interact with clouds at many levels, using high-level service interfaces such as PyTorch [76] or TensorFlow [53] in addition to low-level orchestration interfaces such as Kubernetes [36]. If the goal is to make workload migration seamless, then all of these interfaces would need to be standardized. Requiring every cloud to standardize every interface is both unrealistic (as noted in the first objection) and unwise (because these higher-level interfaces have changed significantly over time, and standardizing them would greatly hinder innovation).

2.2 Why Isn't This Just Multicloud?

Multicloud is now an industry buzzword, and there are reports [33, 52] that most enterprises have, or will soon have, multicloud deployments; this would seemingly imply that our goal of seamless workload migration has already been realized. However, the common use of the term multicloud only requires that an enterprise have workloads on two or more clouds (e.g., the finance team runs their backend functions on Amazon while the analytics team runs their ML jobs on Google), *not* that they can easily move those workloads between clouds. It is clear, from everyone we have talked to in the industry, that moving many workloads between clouds remains difficult. The exceptions to this are the recent third-party offerings (e.g., by Trifacta, Confluent, Snowflake, Databricks, and others) that run on multiple clouds; users can indeed migrate their workloads that only use these services between clouds relatively easily (BigQuery, offered by Google, offers similar cross-cloud support). However, these are for specific workloads, and do not provide general support for workload migration.

In addition, there are several programming or management frameworks that support multiple clouds. JClouds [8] and Libcloud [10] offer portable abstractions over the compute, storage, and other services of many providers. However, the user still does the placement manually, whereas automatic placement is a key feature of Sky Computing. On the management front, Terraform [51] provisions and manages resources on different clouds, but requires the usage of provider-specific APIs, and also does not handle job placement. Kubernetes [36] orchestrates containerized workloads and can be run across multiple clouds (e.g., Anthos [5]). These frameworks, while quite valuable, focus on providing more compatibility in the lower-level infrastructure interfaces offered by the clouds (see §2.3), and as such are nicely complementary with Sky Computing but do not obviate the need for Sky Computing.

2.3 Growth In Interface Compatibility

Turning away from related concepts, we now discuss a recent development that Sky Computing will leverage. As noted before, users of cloud computing invoke a wide variety of computational and management interfaces. Many of these are open source systems that have become the *de facto* standards at different layers of the software stack, including operating systems (Linux), cluster resource managers (Kubernetes [36], Apache Mesos [63]), application packaging (Docker [27]), databases (MySQL [41], Postgres [43]), big data execution engines (Apache Spark [93], Apache Hadoop [89]), streaming engines (Apache Flink [57], Apache Spark [93], Apache Kafka [9]), distributed query engines and databases (Cassandra [7], MongoDB [39], Presto [44], SparkSQL [48], Redis [45]), machine learning libraries (PyTorch [76], TensorFlow [53], MXNet [58], MLflow [38], Horovod [79], Ray RLlib [66]), and general distributed frameworks (Ray [71], Erlang [55], Akka [1]). In addition, some of AWS's interfaces are increasingly being supported on other clouds: Azure and Google provide S3-like APIs for their blob stores to make it easier for customers to move from AWS to their own clouds. Similarly, APIs for managing machine images and private networks are converging.

These trends increase what we call *limited interface compatibility*, where both of these qualifiers are crucial. This compatibility applies only to individual interfaces and these interfaces are typically not supported by all clouds but by more than one. Our contention, based on what we see in the ecosystem, is that the number and the usage of these interfaces that have this limited compatibility – i.e., are supported on more than one cloud – is increasing, largely but not exclusively due to open-source efforts.

We are basing our approach on the belief that this trend will

continue, and that leveraging this trend is far preferable to pursuing uniform and comprehensive standards. To paraphrase a quote attributed to Lincoln, we know that all interfaces are supported by some clouds, and some interfaces may be supported by all clouds, but we cannot and should not require that all interfaces be supported by all clouds.[1]

3 The Vision of Sky Computing

We first describe what Sky Computing is, and articulate why we see it as not just tactical but transformative.

3.1 What Is Sky Computing?

Given this increasing level of limited interface compatibility, how do we leverage it to ease workload migration? There are two key components. First, in order to reduce data gravity, clouds can enter into reciprocal free data peering; i.e., two clouds can agree to let users move data from one cloud to another without charge. With high-speed connections prevalent (many clouds have 100 Gbps connections to various interconnection points where they can peer with other clouds), we think such free peering can easily be supported, with its costs more than offset by the increase in computational revenue that it enables. One might worry about the delay that such transfers incur, but if the resulting computation times are superlinear in the data size (or linear with a reasonably high constant) then no matter how large datasets become, the networking delays will not be a major bottleneck.

The second component, and the one we focus on for the rest of this paper, is what we call *intercloud brokers*. In this paper we describe our intercloud broker, which is designed specifically for *computational batch jobs* (§4). While batch jobs (e.g., ML, scientific jobs, data analytics) represent only a fraction of today's diverse cloud use cases, their computation demands are growing quickly [74] and are responsible for the recent surge of specialized hardware [15, 22, 23]. Thus, we have started with a broker designed for batch jobs as a tractable but common and rapidly growing workload. We expect future versions of the broker will address a wider range of workloads, and provide a broader set of features, but that is not our focus here. In addition, we expect that eventually there will be an open market in intercloud brokers that charge a small fee for their brokerage service; some of those brokers will be general purpose and others more tailored to specific workloads, as ours is.

An intercloud broker takes as input a computational request that is is specified as a directed acyclic graph (DAG) in which the nodes are *coarse-grained* computations (e.g., data processing, training).[2] For lack of a better term we call these computations "tasks". The request also includes the user's preferences about price and performance.

[1] The following adage is widely but incorrectly attributed to Lincoln: "You can fool part of the people some of the time, you can fool some of the people all of the time, but you cannot fool all the people all of the time."

[2] This is informed by workflow systems [6] that are now the de facto standard for orchestrating complex batch applications.

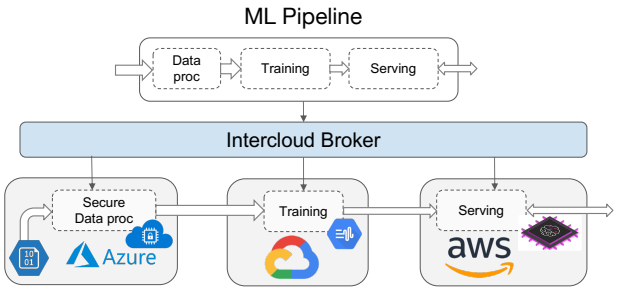


Figure 1: **An ML pipeline running on top of Sky.** The goal is to minimize cost while processing the input data securely.

The intercloud broker is then responsible for placing these tasks across clouds. Unlike existing multicloud applications which run an application instance per cloud, an intercloud broker can run a single application instance across several clouds. For example, Figure 1 shows a machine learning (ML) pipeline with three tasks: data processing, training, and serving. The user may wish to minimize the total cost while processing data securely. The intercloud broker might decide to run data processing on Azure Confidential Computing [16] to anonymize data and thus protect data confidentiality, training on GCP to take advantage of TPUs [23], and serving on AWS to take advantage of the Inferentia accelerator [15].

The ability to partition applications enables the emergence of specialized clouds. For example, a cloud provider can build a successful business by just focusing on a single task, such as ML training, and offering the best price-performance for that task; see §A.1 for a more detailed discussion of this.

In addition, the intercloud broker provides benefits even when the application (i) entirely runs on a single cloud, by automatically choosing the cloud that best matches the user's preferences and choosing the best region and zone within that cloud, or (ii) uses services[3] provided only by a single cloud, by placing a task on that cloud but still having the freedom to use other clouds for the other tasks.

3.2 Why Is This Transformational?

There are three reasons, each from a different perspective, why we see this as a transformational change in cloud computing, not as merely a tactical mechanism for workload migration.

User's Perspective: When using an intercloud broker, users are no longer interacting with individual clouds, but with a more integrated "Sky" of computing. They merely specify their computation and their criteria, and the broker then places the job. This makes it significantly easier to use the cloud, and may lead to increased cloud adoption. Note that such an interface hides the heterogeneity between and within clouds. Users no longer need to research which clouds have the best prices, or offer a particular service. This also applies *within* individual clouds, because different regions within a cloud

[3] By "service" we mean the compute services or a hosted service provided by one or more clouds, such as hosted Apache Spark (e.g., EMR [4], HDInsight [17]) and hosted Kubernetes (e.g., EKS [3], GKE [32], or AKS [18]).

can offer different hardware options and different prices.

Competitive Perspective: Note that by serving as an intermediary between users and clouds, the intercloud broker is creating a fine-grained two-sided market for computation: users specify their tasks and requirements, and clouds offer their interfaces with their pricing and performance. Job placement is no longer driven mostly by measures to promote lock-in (e.g., proprietary interfaces and data gravity), but increasingly by the ability of each cloud to meet the user's requirements through faster and/or more cost-efficient implementations. This means that the clouds, in order to increase their market, will likely start supporting interfaces that are commonly used in jobs, driving the market towards increased compatibility.

Ecosystem Perspective: Once there is a two-sided market established, the cloud ecosystem can transition from one in which all clouds offer a broad set of services and try their best to lock customers in, to one in which many clouds focus on becoming part of a computational Sky, where they can specialize in certain tasks because the intercloud broker will automatically direct computations to them if they best meet user needs for those particular tasks; the economic analysis in the appendix (§A.1.2) makes this case more precisely.

This vision should be tempered with several doses of reality. First, while we envision some clouds will embrace the vision of Sky Computing by focusing on compatible interfaces and adopting reciprocal free data peering, we expect others, particularly those with dominant market positions, to continue with lock-in as a market strategy. Nonetheless, the presence of a viable alternative cloud ecosystem will set the bar for innovation and meeting user requirements, so all users will benefit. Second, we assume that the creation of Sky Computing will be a lengthy process that will start slowly and gradually gather momentum. Our goal in this paper is to investigate how to start this transformation, not to define its ultimate form. As such, we start with with an intercloud broker for batch jobs—a small but important set of workloads. Third, given our focus on the early stages of the Sky, we do not provide solutions to several problems that must eventually be addressed, such as how to troubleshoot failures that occur with applications running across multiple clouds.

4 Intercloud Broker

We now present an intercloud broker that targets *batch applications*. We first review the requirements of such a broker, and then propose an architecture. Finally, we describe our implementation of the resulting design, called *SkyPilot*.

4.1 Requirements

Cataloging cloud services and instances. There is a huge and growing number of services, instances, and locations[4] across clouds. As shown in Table 1, the top three public clouds alone provide hundreds of compute VM types in dozens of regions across the globe. Even for a simple request of a 4-vCPU VM in the "compute-optimized" family—advertised by all three clouds—there are at least 90 choices within the US in terms of region and VM type. Furthermore, each cloud has hundreds of software services (e.g., hosted Kubernetes/Spark, blob storage, SQL databases) to choose from. This is clearly beyond what can be navigated manually by ordinary users.

[4]We use "locations" to refer to regions and zones, collectively.

Cloud	Regions	Zones	VM types
AWS	20 (US: 4*)	64 (US: 15*)	$\geq$ 558
Azure	51 (US: 8*)	124 (US: 23*)	$\geq$ 714
GCP	35 (US: 9)	106 (US: 28)	$\geq$ 155

Table 1: **Top public clouds with their myriad choices of locations and compute instance types**. Data is gathered from each cloud at the time of writing. *Not counting government cloud regions.

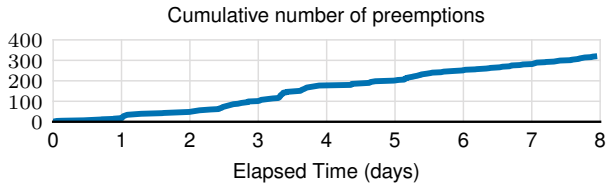


Figure 2: **Dynamic resource unavailability:** preemptions over time from a real-world bioinformatics workload trace. The workload ran for 8 days, using 24 large-CPU spot VMs on GCP, us-west1.

To provide the automatic placement of jobs, the broker must catalog the variety of instances and services, the APIs to invoke these services, and the subset of clouds and regions where these offerings are available.

Even after they have been cataloged, these many options are hard to navigate. Thus, the broker should expose filters on common attributes to applications so that they can easily narrow down the many options across clouds. For compute instances, filters may include the number of vCPUs, RAM, and accelerator types. For managed services (e.g., hosted analytics), filters may include the service or the package version (e.g., AWS EMR 6.5, or Apache Spark 3.1.2). Moreover, the broker should allow an application to choose specific services or instances supported only by one cloud.

Tracking pricing and dynamic availability. The price and availability of resources can vary dramatically across clouds and even regions or zones in the same cloud, often, but not always, following a diurnal pattern [73]. The variations are especially acute for *scarce resources* (§5.4), such as GPUs or preemptible spot instances that many applications use due to their lower costs, and change over time.

To illustrate the potential changes in resource availability, consider a real user's application: a bioinformatics task running for 8 days on 24 spot VMs on GCP (see §5.2 for more detail). When a VM is preempted, it waits for another spot VM to become available. Figure 2 shows the cumulative number of preemptions over time. Note that preemptions happened every day and at *unpredictably* different rates (e.g.,

compare day 3–4 vs. day 4–5). The application experienced 319 preemptions, a preemption every 36 minutes on average.

Thus, the broker should track the availability and pricing to provide applications with the best choices at run time. One challenge is that clouds do not publish availability information explicitly. The broker may have to learn about availability implicitly by observing preemptions or allocation failures of both on-demand and spot resources in different locations.

Dynamic optimization. Recall that the goal of the broker is to meet the application's cost and performance requirements under various constraints, such as data residency. This means the broker should choose the types of instances or services, clouds, and locations to run the tasks in the application DAG. This is a challenging optimization problem because of (1) the sheer number of choices (Table 1), (2) DAG topologies becoming complex (Figure 10), and (3) the unpredictable resource availability and price changes during the application's provisioning or run time (Figure 2).

As a result, the broker should implement a dynamic optimizer that can reflect the current resource availability and prices, and quickly find an optimal execution plan out of the large search space. To use up-to-date prices, the broker needs to compute the execution plan whenever an application starts. In addition, when a task in an application DAG cannot run as the broker originally planned due to availability changes, the broker needs to generate a new execution plan by *re-optimization* during the application's run time.

Managing resources and applications. Once the optimizer decides the placement of an application, the broker must provision the resources and free them when the application terminates. This involves starting and *reliably* shutting down instances on various clouds, or creating and terminating services (e.g., sending requests to a hosted service like AWS EMR). While these lifecycle operations may seem straightforward, bugs or failures can easily lead to inconsistencies between the broker state and the cloud provider state (e.g., leaking instances or intermediate data), which can be costly.

In addition, the broker must manage the execution of the application, i.e., start an application's task when its inputs are available, possibly restart it in case of failures or preemptions, and move the task's inputs across clouds/regions, if remote.

4.2 Architecture

Given these requirements, we propose an intercloud broker architecture consisting of the following components (Figure 3).

Catalog. The catalog records the instances and services available in each cloud, detailed locations that offer them, and the APIs to allocate, shut down, and access them. It also stores the long-term prices for on-demand VMs, data storage, egress, and services (typically these prices do not change for months). The catalog can provide filtering and searching functionalities. The catalog can be based on information published by the clouds, listed by a third party, or collected by the broker.

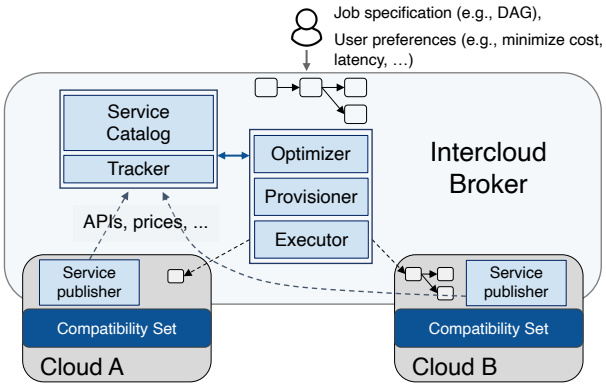


Figure 3: **Architecture of the intercloud broker.**

Tracker. This component tracks spot prices (which can change more frequently, e.g., hourly or daily) as well as resource availability across clouds and their locations.

Optimizer. The optimizer takes as inputs (1) the application's DAG and its requirements, and (2) the instance and service availability as well as their prices provided by the catalog and tracker, and then computes an optimal placement of the tasks. Upon resource availability and price changes, the optimizer may perform re-optimization.

Provisioner. This component manages resources (§4.1) by allocating the resources required to run the execution plan provided by the optimizer, and freeing them when each task exits. To handle unpredictable capacity and user quota errors, the provisioner implements automatic failover, where it asks the optimizer for a new placement plan if the provision fails. Failures are also reported to the tracker.

Executor. The executor manages the application (§4.1) by packaging each application's tasks and running them on the resources allocated by the provisioner.

In the future, we imagine intercloud brokers will offer more sophisticated services such as troubleshooting across clouds, providing more detailed performance measurements for specific applications on each cloud, the equivalent of spot-pricing but across clouds, reselling services at lower than listed prices (similar to the travel industry), and advanced configuration features for security and/or networking.

Furthermore, we expect a commercial broker to provide billing support to enable a user to have a single account with the provider of the intercloud broker, which then pays for the services rendered by each cloud on behalf of the user, and charges the user back. In our current deployment, our users have direct accounts with the three major clouds, so this functionality is not needed.

4.3 SkyPilot: An Implementation

We have implemented *SkyPilot*, which follows the architecture described in §4.2 with one difference: instead of implementing the tracker as a centralized component, SkyPilot distributes it between the catalog that refreshes prices daily, and the provisioner that tracks and caches provisioning failures.

SkyPilot is written in ≈21,000 lines of Python code, and has involved several person-years so far. It currently supports AWS, Azure, and GCP. It is being used by users from 3 universities and 4 other organizations; we report our deployment experience in §6. Next, we first describe SkyPilot in detail, then discuss the services in the compatibility set it uses.

Application API. As mentioned earlier, an application is specified as a DAG of coarse-grained tasks. Example tasks include a Spark job to process data, a Horovod [79] job to train a model, or an MPI job for HPC computations. A task starts when all of the tasks that provide its inputs have finished. Each task is self-contained and includes its executable and all library dependencies (e.g., packaged as a Docker image).

A task specifies its *input and output locations* in the form of cloud object store URIs. Optionally, a task can provide the *size estimates* of its inputs and outputs to help the optimizer estimate the cost of data transfers across clouds.

Each task specifies the *resources* it requires. For flexibility, resources are encoded as labels, such as "cpu: 4" or "accelerator: nvidia-v100", an idea we borrow from cluster managers such as Borg [85], Mesos [63], and Condor [82]. The optimizer uses these resource labels to search the service catalog for a set of feasible candidates for each task. If desired, the user can short-circuit the optimizer's selection by explicitly specifying a cloud and an instance type.

The user optionally specifies the number of instances for each task by a "num_nodes: *n*" label, which defaults to 1. Since we target coarse-grained batch jobs, our users have not found this a burden. In the future, we plan to support autoscaling or intelligently picking the number of instances [54, 84].

Finally, the user supplies an optional *time estimator* for each task, which estimates how long it will run on each specified resource. These estimates are used by the optimizer for planning the DAG. The user could determine these estimates by benchmarking the task on different configurations. If a time estimator is unspecified for a task, currently the optimizer defaults to the heuristic of choosing the resource with the lowest hourly price.[5]

Example. Listing 1 shows an application consisting of two tasks. The `train` task trains a model. It reads the input data from S3 and writes the output (the trained model) to the object store of the cloud it is assigned to run on, which is determined by the optimizer. By using `Resources`, a dictionary of resource labels, the user specifies that this training task requires either an nvidia-v100 accelerator or a google-tpu-v3-8 accelerator with 4 host vCPUs. The user also provides a `train_time_estimator_fn` lambda that estimates the task's run time on these two accelerators. For example, one can compute a rough estimate by dividing the total number of floating operations required for training the model by the accelerator's performance in FLOPS (floating point operations per second),

```
# A simple application: train -> infer.
with Dag() as dag:
  train = Task('train', run='train.py',
               arg='--data=$INPUT[0] --model=$OUTPUT[0]')
    .set_input('s3://my-data', size=150 * GB)
    # '?': saves to the cloud this op ends up running on.
    .set_output('?://my-model', size=0.1 * GB)
    # Required resources. A set ({}) means pick any Resources.
    .set_resources({
      Resources(accelerator='nvidia-v100'),
      Resources(accelerator='google-tpu-v3-8', cpu=4)})
    # A partial function: Resources -> time.
    .set_time_estimator(train_time_estimator_fn)

  infer = Task('infer', run='infer.py',
               arg='--model=$INPUT[0]')
    .set_input(train.output(0))
    .set_resources({
      Resources(accelerator='nvidia-t4'),
      Resources(accelerator='aws-inferentia', ram=16 * GB)})
    .set_time_estimator(infer_time_estimator_fn)
  # Connect the tasks.
  train >> infer
```

Listing 1: **API to express a simple application.**

or use a more accurate benchmarking-based predictor.

The `infer` task performs model serving. It takes the trained model as input (`set_input(train.output(0))`). The Airflow-like statement, `train >> infer`, enforces this dependency. These two tasks are encapsulated in a `Dag` object. The DAG is passed to the optimizer to output an execution plan, which is then passed to the provisioner and the executor.

Figure 4a visualizes the DAG. (I/O data are task attributes and not nodes in the DAG; we show them for clarity.) While simple, this basic API already exposes many degrees of freedom. For example, while train's input is on S3, the optimizer may choose to assign the task to a different cloud. In doing so, the optimizer must take into account the possible transfer costs, while satisfying the task's requirements.

For convenience, SkyPilot also offers a YAML interface to specify an application in addition to the programmatic API.

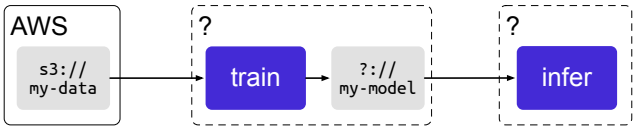
Catalog. SkyPilot implements a simple catalog to support three services (IaaS, object stores, managed analytics) on AWS, Azure, and GCP. These offerings are sufficient for our target workloads. We use the clouds' public APIs to obtain details about these offerings. Pricing is refreshed periodically.

Optimizer. The optimizer assigns each task to a cloud, location, and hardware configuration to best satisfy the user's requirements, e.g., minimize the total cost or time. It achieves this by filtering the offerings in the service catalog and solving an integer linear program (ILP) to pick an optimal assignment.

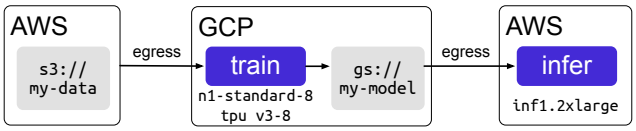
Before the actual optimization takes place, the optimizer first translates the high-level resource requirements into a set of feasible configurations, i.e., tuples of ⟨cloud, zone, instance type⟩, that can be used to run each task.[6] We call such a configuration a *cluster*. For example,

[5] Prior work [83] have considered performance prediction for analytics [84] and machine learning [78] workloads, which can also be leveraged.

[6] This also applies to most hosted analytics offerings (e.g., EMR, Dataproc) as they allow users to specify the cluster size and instance types.



(a) The declared application DAG.



(b) DAG, after optimization.

Figure 4: **An application, before and after optimization.**

`Resources(accelerator='nvidia-v100')` can be mapped to a cluster of AWS instances ⟨AWS, us-west-2a, p3.2x⟩ or Azure instances ⟨Azure, westus2-1, NC6s_v3⟩. To perform this translation, the optimizer filters the offerings in the service catalog to check if they satisfy the `Resources` required by each task. Each task is then annotated with the list of feasible clusters.

The optimizer computes execution plans at a zone level rather than a region level. This is because even in the same region, different zones can have different instance types and prices, and the data transfer between zones is not free.

ILP-based optimization. Consider a DAG with N tasks, each with C feasible clusters. Because C is typically in the 10s and can be up to 100s,[7] naively enumerating all C^N possible assignments is infeasible even for modest values of N. To solve this, we formulate the assignment problem as a 0-1 ILP.

SkyPilot supports two types of optimization objectives: either total running cost or end-to-end run time. Our ILP formulation is inspired by Alpa [94], but we additionally consider the parallelism between tasks that do not have dependency on each other. This is critical for minimizing the DAG run time.

Given a DAG (V, E) where V is the set of the tasks and E is the set of the edges representing the data dependencies between the tasks, our goal is to find an optimal mapping from each task in V to one of its annotated feasible clusters. For each task $v \in V$, we denote the set of the feasible clusters by C_v. Then we use a task time estimator to obtain a time vector $t_v \in \mathbb{R}^{|C_v|}$, where each element is the time estimate for running task v on a cluster in C_v. The time estimator can be either provided by the user or set to a default value of 1 hour. In addition, we get a cost vector $c_v \in \mathbb{R}^{|C_v|}$ by multiplying t_v by the hourly price of each cluster. To account for the data transfer overhead between two tasks $(u, v) \in E$, we define a matrix $P_{uv} \in \mathbb{R}^{|C_u| \times |C_v|}$ whose (i, j) element is the data transfer time when the parent task u is mapped to the i-th cluster of C_u and the child task v is mapped to the j-th cluster of C_v. Similarly, we define $Q_{uv} \in \mathbb{R}^{|C_u| \times |C_v|}$ for the data transfer cost between u and v.

When minimizing the total cost, we have:

$$\min_{s} \underbrace{\sum_{v \in V} s_v^T c_v}_{\text{computation cost}} + \underbrace{\sum_{(u,v) \in E} s_u^T Q_{uv} s_v}_{\text{data transfer cost}} \quad (1)$$

where $s_v \in \{0,1\}^{|C_v|}$ is a one-hot vector that selects a cluster from C_v. The objective explicitly considers the two types of cost: the first term represents the total cost spent in executing all tasks on the selected clusters, while the second term represents the total data transfer cost. After we linearize [61] the second term, we get a 0-1 ILP, which SkyPilot solves using an off-the-shelf solver, CBC [60].

Similarly, when minimizing the end-to-end time, we have:

$$\min_{s} f_{\text{sink}} \quad (2)$$

$$\text{s.t.} \quad f_v \geq \underbrace{f_u}_{\substack{\text{parent} \\ \text{finish time}}} + \underbrace{s_u^T P_{uv} s_v}_{\substack{\text{data transfer} \\ \text{time}}} + \underbrace{s_v^T t_v}_{\substack{\text{computation} \\ \text{time}}} \quad \forall (u,v) \in E \quad (3)$$

where $s_v \in \{0,1\}^{|C_v|}$ is the one-hot decision vector and $f_v \in \mathbb{R}$ is the *finish time* of the task v. The optimization constraint ensures that a task finishes no earlier than its parents, the input data arrive, and the task produces its outputs. Under these constraints, the running time of the DAG becomes the finish time of its sink.[8] Again, as we can linearize the second term, this problem can be efficiently solved by 0-1 ILP solvers.

While we cover the two representative objectives above, our ILP formulation allows any combination of cost and time to be used for the optimization. For example, we can minimize the cost under a time budget (or vice versa), by augmenting Equation 1 with the constraint in Equation 3 and bounding f_{sink} by the time budget. Future work can incorporate carbon footprint of cloud regions [21] into placement decisions.

Provisioner. SkyPilot implements a *provisioner* that reads the optimized plan and allocates a cluster for the next task ready to execute. As discussed, allocations may fail due to either *insufficient capacity* in a cloud's location or *insufficient quota* of the user's account. On such failures, the provisioner kicks off *failover* as follows. First, the failed location is temporarily blocked for the current allocation request with a time-to-live. Then, the optimizer is asked to *re-optimize* the DAG with this new constraint added. The provisioner then retries in the newly optimized location (another location of the same cloud or a different cloud). If all available locations fail to provide the resource, either an error is returned to the user or the provisioner can be configured to wait and retry in a loop.

We found failover to be especially valuable for scarce resources (e.g., large CPU or GPU VMs). For example, depending on request timing, it took 3–5 and 2–7 location attempts to allocate 8 V100 and 8 T4 GPUs on AWS, respectively.

[7] For instance, the previous example that requires one V100 GPU maps to 79 feasible clusters globally across AWS, Azure, and GCP.

[8] If the DAG has multiple sinks, we create a dummy sink that has a fake dependency on the real sinks.

Executor. After a cluster is provisioned, the *executor* orchestrates a task's execution, e.g., setting up the task's dependencies on the cluster, performing cross-cloud data transfers for the task's inputs, and running the task (which can be a distributed program utilizing a multi-node cluster). We built an executor on top of Ray [71], a distributed framework that we use for intra-cluster task execution with fault tolerance support. Using Ray, rather than building a new execution engine, allowed us to focus on building the higher-level components new to the broker. For example, our executor implements a storage module that abstracts the object stores of AWS, Azure, and GCP and performs transfers. The executor also implements status tracking of task executions for resource management. On execution failures, the executor optionally exposes cluster handles to allow login and debugging.

The executor interface is modular. We envision other executors will be added in the future, e.g., for Kubernetes [36]. In addition, while our system formulation is generic enough to support arbitrary DAGs, our implementation of the executor has focused on supporting pipelines (sequential DAGs).

Compatibility set. One of the distinguishing features of Sky is leveraging the already existing services and APIs across clouds (i.e., compatibility set; §2.3), rather than building uniform services and APIs across all clouds. However, a broker still needs to develop some glue-code to handle similar but not identical services supported by different clouds. The natural question is what is the effort to implement such glue-code? The answer for our applications so far is "minimal".

To manage clusters, SkyPilot uses Ray's cluster launcher, which already supports AWS, GCP, and Azure. (Other frameworks could also be used, e.g., Terraform [51].) The main functionality we added is the control for automatic failover.

One of the most important components of any Sky application is storage. While the APIs provided by the object stores of the three major clouds are similar, they are not identical. Fortunately, all have libraries [20, 30, 46] exposing the POSIX interface, which allows us to mount different object stores as directories. Providing this functionality required only 400–500 lines of code (LoC) per object store.

Finally, for analytics applications we use high-level APIs, e.g., hosted analytics services provided by AWS (EMR) and GCP (Dataproc). Abstracting these services required us to implement just two methods: provisioning and termination. This involved only 200 LoC for EMR and Dataproc together.

5 Experiments

We conduct a series of experiments to evaluate the benefits of our intercloud broker. Overall, we found that:

- SkyPilot enables batch applications to take advantage of unique hardware, unique managed services, and improved availability across locations and clouds.
- On three applications (ML pipelines, scientific jobs, and data analytics), SkyPilot saves up to 2.7× in time, 80% in cost, and 2× in makespan, compared to using a single cloud or location.
- Even for single-cloud applications, the broker improves availability by migrating jobs across regions, a policy not supported by cloud providers' own solutions (§5.2).

Workload	Uses	Benefits from
ML	IaaS	unique hardware
Bioinformatics	IaaS (spot VMs)	improved availability
Analytics	managed analytics	unique software service & unique hardware

Table 2: **Evaluated workloads, cloud services used, and benefits.**

Table 2 shows all workload types and their respective benefits.

5.1 Machine Learning Pipelines

We start with running two ML pipelines on SkyPilot to leverage the strengths of different clouds. In both pipelines, the goal is to minimize the total cost. We consider two scenarios:

- Single-cloud: all tasks are constrained to a single cloud;
- Broker: each task runs according to the plan generated by SkyPilot's optimizer, possibly on different clouds.

Overall, both pipelines benefit from SkyPilot's flexibility to run compute-intensive tasks on clouds with unique hardware accelerators (e.g., Inferentia, TPUs) that can provide speedups which offset the cost and latency of moving the data.

Due to space limit, we show in appendix (§A.2) an additional experiment on SkyPilot leveraging spot instances across clouds to run ML training with improved availability and cost.

5.1.1 Vision Pipeline

The vision pipeline consists of two tasks: train and infer (see Listing 1). The train task trains a ResNet-50 model on the ImageNet dataset (150 GB, stored on AWS S3). The infer task runs offline inference on 10^8 images (e.g., nightly photo categorization for services like Instagram or Google Photos).

Since training deep learning models often requires iterative and heavy computations, we demonstrate a large reduction in cost and run time by moving the training data from AWS to GCP to leverage its TPU accelerators for training [23].

Setup. We specify resource candidates for each task as:

- train: `'nvidia-v100'`, `'google-tpu-v3-8'`
- infer: `'google-tpu-v3-8'`, `'nvidia-t4'`, `'aws-inferentia'`

For train, we use a V100 (common high-end GPU for training) or a TPU. For infer, we use a TPU, a T4 GPU (marketed as the most cost-effective GPU for model inference), or an Inferentia accelerator designed by AWS for cost-effective inference [15].

The best single-cloud plans are shown in Figure 5, termed {AWS, GCP, Azure}-only. The Broker plan is SkyPilot's optimizer output that minimizes the total cost. In this experiment, we used a simple time estimator that divides the total FLOPs

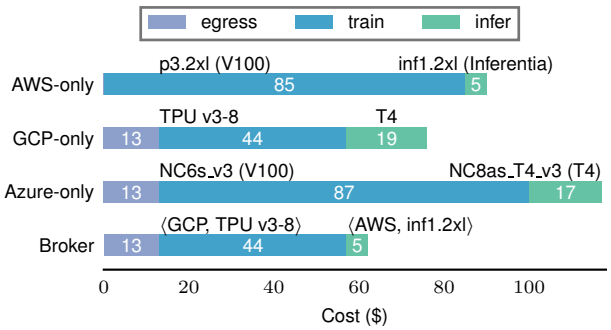


Figure 5: **Vision pipeline:** hardware and costs of each deployment. For simplicity, the zones chosen for the plans are omitted. For training we use mixed-precision and the XLA compiler [50] with TensorFlow Keras 2.5.0. For inference we use half-precision. On GCP, accelerators are attached to an n1-standard-8 VM.

required to train the model by the hardware FLOPS:[9]

```
def train_time_estimator_fn(resource):
  train_tflops = ...  # Obtained from model analysis.
  if resource.accelerator == 'nvidia-v100':
    hardware_tflops = 120
  if resource.accelerator == 'google-tpu-v3-8':
    hardware_tflops = 420
  return train_tflops / hardware_tflops
```

We used a similar FLOPs-based time estimator for infer.

Results. We show the plan generated by SkyPilot's optimizer in Figure 4b and the results in Figure 5.

While this pipeline is simple, *its search space is already large*, with a total of 2,170 possible assignments (details in §5.4), as we have multiple choices in hardware, cloud, and location. The optimizer successfully finds an optimal solution. Compared with the three single-cloud plans, the Broker plan lowers the total cost by 18%–47%, by taking advantage of the unique hardware capabilities across two clouds.

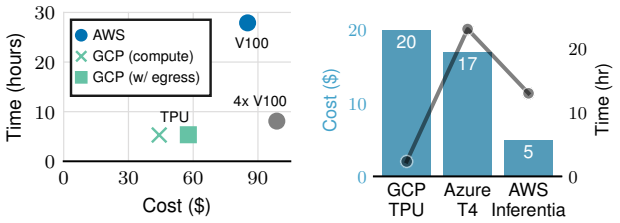
For train, the optimizer decides that, despite the input being stored on AWS, it is better to incur an egress cost and ship it to GCP to use the TPU. This choice leads to a cost of $57 ($44 compute, $13 egress) which is less than training on AWS, at $85.[10] SkyPilot's storage module uses GCP's storage transfer service [31] to copy the data in about 3 minutes.

For infer, the optimizer estimates that AWS's Inferentia is more cost-effective than the T4 GPU, after factoring in a small data egress cost (shipping the first task's output, a 0.1 GB model, from GCP to AWS with a cost of $0.01).

To understand the cost savings, we compare the detailed time and cost per task. For training (Figure 6a), SkyPilot's choice of GCP TPU takes 5.4 hours and costs $57 with egress included, which is 5.2× faster and 33% cheaper than the AWS V100 plan. (Azure V100 is similar but has $13 for egress; hence omitted.) To make the hardware more comparable, we

[9] While crude, this estimate is a reasonable approximation for throughput-bound models with intensive matrix operations, such as ResNet.

[10] If we set the input 4× as large, at 600 GB, the optimizer decides against transferring the data as the egress cost will dominate.



(a) Training (b) Serving

Figure 6: **Vision pipeline:** detailed breakdown per task.

submitted the task again requesting 4 V100s on AWS to match the FLOPS performance of a TPU v3-8: still, TPU is 1.5× faster and 42% cheaper than 4 V100s. For serving (Figure 6b), AWS's custom Inferentia chip saves both cost (71%) and time (1.8× faster) compared to the widely available T4 GPU.

Thus, clouds offer *unique hardware incentives* to different tasks, even if the data is stored on a different cloud.

Optimizing for time vs. cost. To test SkyPilot's ability to minimize the total time rather than cost (§4.3), we resubmit this pipeline to SkyPilot with the time-minimizing objective. The resource selection for train remains the same. For infer, SkyPilot now chooses GCP TPU (estimated to take 2.5 hours and cost $21, per 10^8 images) over AWS Inferentia (which was *cost-optimal*; estimated to take 8.2 hours and cost $3). The estimates reflect the actual ranking in Figure 6b. Even though the TPU costs 4× more in total than Inferentia, it reduces inference time by 5.7×. This example shows that optimal placements can change based on user preferences.

5.1.2 NLP Pipeline

We next run a natural language processing (NLP) pipeline that emulates an increasingly prevalent workload: fine-tuning "foundation models" [56]. It consists of three tasks (Figure 1):

- **Confidential data processing**: remove sensitive information from raw data using Intel SGX hardware enclaves. We use the Amazon Customer Reviews Dataset [2] and treat it as if it contained personally identifiable information (PII) and thus must be processed securely. To remove sensitive data, we run Opaque [95] on an SGX-enabled instance to filter on a column (i.e., the filtered-out information is assumed sensitive), and output only the review texts and star ratings. The size of the output dataset is 1 GB.
- **Train**: fine-tune BERT-base [59], a popular natural language understanding model, on the preprocessed and now non-sensitive data. This model predicts a rating given a review text. We fine-tune the model for 10 epochs.
- **Infer**: use the model to classify 1M new reviews.

Setup. The first task requires `Resources(intel_sgx=True)`, which is currently only offered by Azure [16]. For training, we consider either 4 V100s, or a TPU v3-8. For serving, we consider either a T4 GPU, or AWS's Inferentia.

Due to the confidential computing requirement, the only possible single-cloud plan is to run all three tasks on Azure:

		proc	train	infer	egress	**Total**
Time	**Azure**	0.6	13.3	1.5	–	15.4
(hr.)	**Broker**	0.6	3.8 -71%	1.4 -7%	0.03	5.8 -62%
Cost	**Azure**	0.8	163	1.2	–	165
($)	**Broker**	0.8	32 -80%	0.5 -58%	0.1	33.4 -80%

Table 3: **NLP pipeline:** run time and cost of each deployment plan.

a DC8 VM for SGX, an NC24s VM with 4 V100 GPUs for training, and an NC8as instance with a T4 GPU for serving.

Results. Table 3 shows the time and cost comparison between the single-cloud and Broker plans. Different from before, the Broker plan for this pipeline uses all three clouds. The search space is larger, *with over 16K possibilities* (§5.4).

As expected, the single-cloud plan restricts its choices of hardware to Azure and thus results in suboptimal cost and performance. While Azure's Intel SGX offering is unique for secure processing, SkyPilot allows this pipeline to leverage different clouds for other tasks of the same application. SkyPilot's optimizer picks the TPU (GCP) over 4 V100s for training, and the Inferentia (AWS) over the T4 GPU for serving. This considerably reduces both the total run time (by 62%) and cost (by 80%) compared with the Azure-only plan.

5.2 Bioinformatics

The intercloud broker should *dynamically* respond to the changing availability of resources (§4.1). We evaluate SkyPilot's handling of availability changes by modeling a *real user's workload*: A bioinformatic task of mapping DNA cells of sequencing data [67, 92]. The jobs are independent, have variable-sized inputs and variable run times, with each using all CPUs within one machine. Jobs are not checkpointable and failures require recomputation from scratch. Finally, these jobs are *recurring*: there are 10s to 100s of jobs to run every week based on incoming data. Due to long run times, this user exclusively uses spot VMs on GCP to save costs, and has been continuously using SkyPilot to do so for several months.

We submit 40 jobs to SkyPilot, each running on an n1-highmem-96 spot VM on GCP for 8–12 hours. We implement and compare two policies in SkyPilot: (1) *SingleRegion*, which retries each preempted job in other zones of the same region—this models providers' managed instances solutions [35]; (2) *Broker*, which retries each preempted job in the next cheapest region chosen by the optimizer. We start two sets of 40 jobs together (to minimize variance due to time) in the region with the cheapest price for this VM (us-west1). We ensure the jobs are within quotas so all job migrations are due to preemptions.

Overall, the Broker policy finishes significantly faster than the SingleRegion baseline, due to experiencing fewer preemptions. Figure 7 (top) shows that Broker completed 75% of the jobs 1.6× or 7 hours faster than SingleRegion. At around $T = 16$ hours, all Broker jobs finished, while 30% (12) of SingleRegion jobs were still running. The last SingleRegion job finished at $T = 32$ hours, yielding a 2× longer makespan.

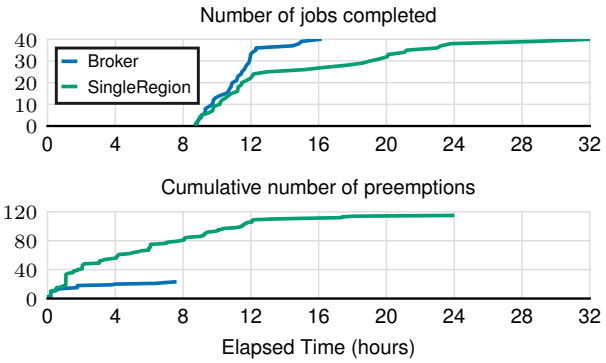


Figure 7: **Dynamically adjusting to availability** on a bioinformatics workload of 40 jobs on spot CPU VMs. *Broker* moves preempted jobs to a new region, while *SingleRegion* moves preempted jobs to other zones in the same region. Note the shared x-axis. Cloud: GCP.

Figure 7 (bottom) shows the speedup comes from Broker incurring 5× fewer preemptions. Since both policies started in the same region, the preemption curves initially overlapped. Broker swiftly moved the 22 preempted jobs to another region, which remained non-preemptive for the entire duration (e.g., last preemption occurred before $T = 8$ hours). The original region continued to experience a high preemption rate in all zones, causing SingleRegion to have far more stragglers.

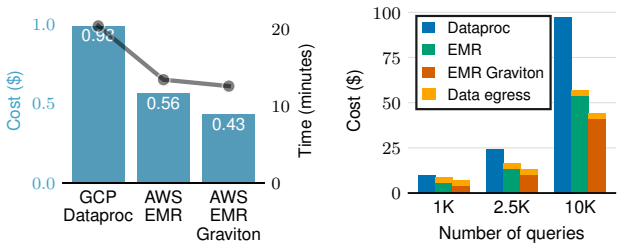
While this example represents a good case (moving from a region with a high preemption rate to a region with a low preemption rate), it shows that SkyPilot can dynamically use multiple regions to improve availability *when needed*. Managed solutions from cloud providers, e.g., spot fleets [49] or managed instances [35], are *confined within a region* and thus cannot support such a cross-region (or cross-cloud) policy.

Finally, note that this policy is not always better than SingleRegion. For example, if the jobs started in a region with a low preemption rate, some unlucky jobs could be preempted and moved to a region with a higher preemption rate, which could be worse than SingleRegion. Importantly, SkyPilot allows new policies (cross-cloud/region) to be implemented easily, and we expect this to be an area of future research.

5.3 Managed Data Analytics

So far, we demonstrated SkyPilot's ability to use IaaS (VMs) on different clouds. We now use the broker to run an analytics workload on the *managed analytics services* of two clouds: AWS EMR [4] and GCP Dataproc [29]. While VMs with the same hardware on different clouds should have mostly the same performance, we expect hosted services to exhibit more performance variations due to differences in software. We run TPC-DS [72] on the following (scale factor 100, or 33 GB of data in Parquet, generated locally on each cloud):

- GCP Dataproc: which runs vanilla Spark 3.1.2, on a 3-node n2-standard-16 cluster. Version 2.0.29-debian10.
- AWS EMR: which runs an *optimized runtime* [42] for



(a) Data is local (99 queries) (b) Data shipped from GCP

Figure 8: **Using managed analytics services with SkyPilot.** TPC-DS. **(a)** Cost (left y) and time (right y) of two hosted services in three configurations, where data is generated locally. Benefits of software and hardware offerings can combine. Mean of 3 runs. **(b)** Assuming data is stored in GCP, running more queries offsets the egress cost.

Spark 3.1.2, on a 3-node m5.4xlarge cluster. Version 6.5.0.

- AWS EMR Graviton: like above, but on a 3-node m6g.4xlarge cluster, which uses the Graviton2 ARM-based processors custom-designed by AWS [14]. Due to its cost-performance benefits, several large companies such as Netflix and Snap have moved some of their workloads to Graviton2 from traditional x86 instances [13].

Figure 8a shows AWS EMR finishes 34% faster and 43% cheaper than GCP Dataproc. We ensured that GCP's n2 cluster has the same or better hardware than AWS's m5.4x cluster. Thus, the speedup is due to EMR's optimized software runtime [42] for Spark, representing a *unique software incentive* for users with similar analytics workloads.

In addition, AWS EMR Graviton *improves* both the cost and run time over AWS EMR by 23% and 6%, respectively. Thus, this is a case of combining the unique *software* and *hardware* advantages to attract such workloads even more.

To understand the tradeoff between better services vs. data gravity, Figure 8b shows the cost of running more queries from the benchmark, assuming the data is not generated locally but initially resides in GCP and has to be copied. (Here, we simply execute the TPC-DS benchmark's 99 queries multiple times to increase the number of queries we ran.) With 1K queries, EMR's speed advantage already offsets the data transfer cost ($2.8). Running 2.5K queries yields a cost saving of 32% for EMR and 46% for EMR Graviton, while running 10K queries yields 42% and 55% savings, respectively.

To request a managed service for a task, we specify

```
task.set_managed_service(
  AnalyticsService(
    dependencies={'Spark': '3.1.2', 'Hadoop': '3.2.1', ...},
    resources=Resources(cpu=16, ram=64 * GB, num_nodes=3)))
```

where `AnalyticsService` is backed by concrete implementations such as EMR or Dataproc. The `dependencies` field specifies the desired package versions for the hosted service; such version lists are published by the cloud providers [11,26] and recorded in SkyPilot's service catalog.

Type	Hardware	Zones	On-demand $ Max/Min	On-demand $ CV	Spot $ Max/Min	Spot $ CV
CPU	AMD (8 cores)	146	2.5×	16%	7.3×	59%
	Arm (8 cores)	88	2.1×	12%	2.5×	17%
	Intel (8 cores)	248	1.6×	12%	9.4×	39%
GPU	K80 (1 chip)	56	9.5×	48%	5.9×	60%
	T4 (1 chip)	146	1.7×	12%	10.8×	29%
	V100 (1 chip)	79	1.6×	14%	1.9×	19%
	A100 (8 chips)	46	1.9×	23%	6.4×	84%
TPU	v2 (8 cores)	5	1.2×	6%	1.2×	6%
	v3 (8 cores)	4	1.1×	4%	1.1×	4%

Table 4: **Capturing the large heterogeneity of locations and pricing in the catalog.** We show for a subset of offerings, the number of zones that provide them (out of 294 zones globally across the top 3 clouds), the pricing ratios of the most costly to the cheapest zone, and the coefficients of variation (CV) of prices across zones. CPUs are the latest generation in the "general-purpose" family.

5.4 Analyzing the Broker

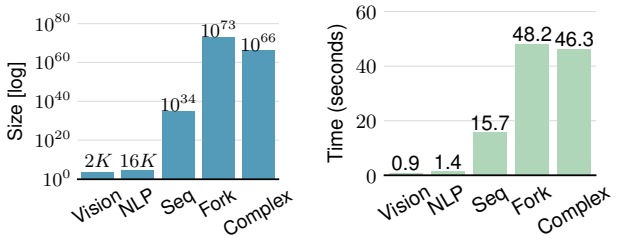
Location and pricing heterogeneity in the catalog. We analyze SkyPilot's service catalog (over 76K entries) to see how well it captures the heterogeneity in locations and prices for all three clouds. Table 4 shows the results. We see that not all offerings (VMs, accelerators) are present in all zones, and there can be large price differences across zones.

Among the 294 zones across the three clouds, the latest Intel CPUs are widely offered, but AMD is only offered in 50% of the zones, while ARM is in only 30%. CPU workloads, e.g., bioinformatics (§5.2) and analytics (§5.3), can suffer from up to 2.5× price premiums if run in the most expensive zone, which increase to 9.4× if spot instances are used. These differences are even larger for NVIDIA GPUs, which are present in just 16–50% of all zones, and their prices vary by up to 9.5× for on-demand and 10.8× for spot. Finally, despite TPUs being offered only in 4–5 (or 5%) GCP zones, there is still a 10%–20% price difference across those zones.

This significant heterogeneity in *locations* and *pricing* makes it hard for users to manually find the best placement. By capturing this heterogeneity, SkyPilot's catalog enables the optimizer to automatically exploit these differences.

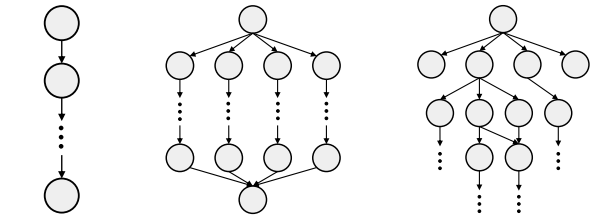
Optimizer overhead. We evaluate SkyPilot's optimizer overhead on a variety of DAGs. Figure 9 shows the search space sizes and the optimization time for the two ML pipelines in §5.1 and 3 other DAGs (see below). Despite the pipelines' simple structures (Vision, NLP), *their search spaces already have 2K–16K possible assignments*, making them non-trivial or infeasible to optimize by hand. Using the ILP, however, our optimizer can find an optimal solution in under 1.4 seconds.

Additionally, we test on three larger and more complex DAGs, found in Airflow's repository [6]: the first two (Figure 10a, Figure 10b) are commonly used in the real world [68], while the third (Figure 10c) has a more complex structure.



(a) Search space sizes (b) Optimization time

Figure 9: **Search spaces and optimization times.** Timing is measured on an M1 MacBook Pro; mean of 3 runs. Objective is cost. Locations of feasible clusters are limited to all US zones on 3 clouds.



(a) Sequential (b) Fork-Join (c) Complex

Figure 10: **Larger DAGs found in Airflow's repository. (a)** Sequential: $|V| = 20, |E| = 19$. **(b)** Fork-Join: $|V| = 42, |E| = 44$. **(c)** Complex: $|V| = 38, |E| = 53$.

We assume each task requires an 8-vCPU Intel VM in US zones, which leads to 55 feasible clusters for each task. We assign random time estimates (sampled from $U(0,1)$ hours) to each task and a random data transfer size (sampled from $U(0,100)$ GB) to each edge. While the search spaces for the DAGs are combinatorially large (10^{34}–10^{73} possible assignments), optimization takes at most 48.2 seconds. Since each task in a DAG is coarse-grained (e.g., can take hours), this optimization time is a negligible portion of the DAG run time.

If resource availability changes during run time, the DAG may need to be re-optimized to generate a revised execution plan. As the process of re-optimization involves updating the list of feasible clusters and restarting the ILP optimization, its overhead is comparable to that of the initial optimization.

6 Deployment Experience

We have deployed SkyPilot to dozens of users from 3 universities and 4 other organizations, who have been using the broker to run both adhoc and recurring batch jobs in the clouds for many months. These users have switched to the intercloud broker from their prior solutions of manually interacting with specific clouds, either via web consoles or low-level APIs. Below, we discuss our experiences with the system so far based on user feedback.

Benefits of an intercloud broker. By surveying our users, we found that users value the broker not only for cost reduction, but also for improved availability (see §5.2) and in general for *improving their productivity*. For example, users like the broker's ability to automatically provision scarce resources across clouds or regions, the easy access to best-of-breed hardware (e.g., TPUs), and the simple packaging of existing programs. Moreover, by interacting with the broker rather than the clouds, they value the ability to run the same jobs on different clouds with no change to their code or workflow.

Cluster reuse for faster development and debugging. Users have reported that the typical provisioning time of several minutes for a new cluster is too long, especially during the iterative code development phase. To alleviate this, we added the ability to reuse existing clusters for running a new application. This also helps the debugging of Sky applications as the users can log into a cluster to inspect and troubleshoot.

Moving data is acceptable for many workloads. Data gravity can prevent workloads from being moved across clouds. However, we found that for many batch workloads, cross-cloud data transfers are not as slow or costly as we expected. In fact, moving data can be profitable even after factoring in the egress (Figure 5; Figure 8).

There are several reasons for this. First, the computation complexity of many batch jobs, such as ML training, is typically *super-linear* in the input size. Second, many datasets are not excessively large. For example, a study from Microsoft reports that most production ML datasets are between 1 GB to 1 TB [75]. Our results (§5.1.1) suggest that a 1 TB dataset can likely be moved in ∼20 minutes with a cost of ∼$90. Depending on the job, this delay and cost can be easily offset by the destination offering better hardware, software, or pricing.

On-premise clusters as part of the Sky. Users have requested the support for running jobs on on-premise clusters through the broker. There are several benefits. First, this would enable users to take advantage of idle local clusters and burst to the cloud when they are overloaded. Second, the broker would offer the same interface that hides the heterogeneity (to the extent possible), so the same Sky applications could run both in the cloud and locally. Challenges include designing spillover policies and handling compatibility and storage.

7 Related Work

Sky Computing. We are not the first to use the name "Sky Computing" as several papers, dating back to 2009, also used this term [62, 69, 70]. However, these papers focus on particular technical solutions, such as running middleware (e.g., Nimbus) on a cross-cloud Infrastructure-as-a-Service platform, and target specific workloads such as high-performance computing (HPC). This paper takes a broader view of Sky Computing, seeing it as a change in the overall ecosystem and considering how technical trends and the market forces can play a critical role in the emergence of Sky Computing.

The work most closely related to this paper is [81], but here we significantly extend that work by refining the vision, designing and building a broker, demonstrating its benefits in several applications, and reporting on early adoption.

Cross-cloud compute, storage, and egress. Supercloud [65] is a virtual cloud that can span multiple zones and clouds, using nested virtualization and live VM migration to move stateful workloads across locations. Our proposal shares the goal of easing workload migration, but supports migrating higher-level jobs (not VMs), considers a broader set of cloud services in addition to IaaS, and focuses on batch jobs by optimizing for price, performance, and availability.

There have been several proposals for cross-cloud storage solutions. CosTLO [91] and SPANStore [90] use request redundancy and replication to minimize storage access latencies. Perhaps the most comprehensive is Gaia-X, a European effort to create a federated open data infrastructure that enables data sharing with strong governance properties and respecting data and cloud sovereignty [28]. These efforts are largely orthogonal to our focus on computational tasks.

Several industry efforts have been started to reduce cross-cloud data egress fees. The Bandwidth Alliance [19] is one such effort, consisting of several cloud providers who agree to reduce or even eliminate egress fees from their clouds to Cloudfare or other members. Closely related is Cloudfare R2 [24], an object store that promises to charge zero egress fees. Naturally, Sky Computing benefits from these efforts to combat data gravity, and the intercloud broker can be extended to support zero-egress storage systems.

Middleware. Middleware solutions (e.g., CORBA [25], Microsoft BizTalk [37], IBM WebSphere [34], etc.) bear some resemblance to our work. While these solutions allow systems from different vendors to communicate and interoperate, our proposal allows an application to utilize cloud services offered by different cloud providers.

There are several differences between these efforts and the intercloud broker. First, we consider satisfying requirements such as minimizing costs which have not been a concern of these systems. Second, the intercloud broker focuses on placing the components of the same application rather than on how systems from different vendors interoperate. Finally, we are operating in a cloud setting rather than a traditional distributed system setting.

Differences aside, middleware solutions that allow cloud services to interoperate (e.g., connect an AWS S3 bucket with GCP Dataproc) could be considered as being part of the compatibility set, which the intercloud broker can leverage.

Integration Platform-as-a-Service (iPaaS). Like the middleware systems discussed above, iPaaS solutions [40, 47] also integrate distinct systems but are often run as managed services on the cloud. iPaaS solutions provide adaptors to connect APIs from different services and systems (e.g., APIs for Snowflake, Jira, or Stripe). Developers can build workflows on top (e.g., on receiving a new case in Salesforce, call Jira's API to open a ticket) and deploy them through the iPaaS.

While iPaaS can run integration workflows on the cloud, our proposal places and runs compute-intensive jobs on the most suitable cloud based on price, performance, and availability. Similar to middleware, iPaaS is complementary as we can leverage these adaptors to expound the compatibility set.

Optimization for geo-distributed analytics. A line of work has optimized the performance of geo-distributed analytics [64, 77, 86]. This setting is similar in spirit to ours: it considers running a MapReduce-style job (an analytics query) across many sites, while we consider running a DAG of coarse-grained computations potentially across several clouds.

There are three main differences. First, these techniques are system-specific optimizations, and we in general do not assume as much knowledge about the application. Second, these techniques mostly assume different sites to differ only in their WAN bandwidths and otherwise have identical hardware, while we exploit the inherent differences in hardware, software, pricing, and resource availability of several clouds or regions/zones within a cloud. Third, these solutions optimize for faster completion times, while we also consider minimizing costs and improving resource availability.

That said, we note that the intercloud broker could potentially leverage system-specific optimizations if it is told that the application is of a certain type (e.g., MapReduce).

8 Conclusion

This paper describes the design, implementation, applications, and early deployment of an intercloud broker, SkyPilot. SkyPilot enables users to seamlessly run their batch jobs across clouds to minimize cost and/or delay. We see this as the first step towards a paradigm we call Sky Computing, which we hope will transform the cloud computing ecosystem to better meet user needs.

Acknowledgements. We thank the NSDI reviewers and our shepherd, Paolo Costa, for their valuable feedback. This work is in part supported by NSF CISE Expeditions Award CCF-1730628 and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware.

References

[1] Akka. https://akka.io/.

[2] Amazon customer reviews dataset. https://s3.amazonaws.com/amazon-reviews-pds/readme.html.

[3] Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/.

[4] Amazon EMR. https://aws.amazon.com/emr/.

[5] Anthos. https://cloud.google.com/anthos.

[6] Apache Airflow. https://airflow.apache.org/.

[7] Apache Cassandra. https://cassandra.apache.org/.

[8] Apache jclouds. https://jclouds.apache.org/.

[9] Apache Kafka. https://kafka.apache.org/.

[10] Apache Libcloud. https://libcloud.apache.org/.

[11] Application versions in Amazon EMR 6.x releases. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-release-app-versions-6.x.html.

[12] Artificial Intelligence: From the Public Cloud to the Device Edge. https://www.equinix.com/resources/whitepapers/nvidia-distributed-ai-cloud-infrastructure-edge.

[13] AWS and Arm. https://www.arm.com/why-arm/partner-ecosystem/aws.

[14] AWS Graviton Processor. https://aws.amazon.com/ec2/graviton/.

[15] AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia/.

[16] Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[17] Azure HDInsight. https://azure.microsoft.com/en-us/services/hdinsight/.

[18] Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service/.

[19] Bandwidth Alliance. https://www.cloudflare.com/bandwidth-alliance/.

[20] BlobFuse - A Microsoft supported Azure Storage FUSE driver. https://github.com/Azure/azure-storage-fuse.

[21] Carbon free energy for Google Cloud regions. https://cloud.google.com/sustainability/region-carbon.

[22] Cerebras. https://cerebras.net/.

[23] Cloud TPU. https://cloud.google.com/tpu.

[24] Cloudflare R2. https://www.cloudflare.com/products/r2/.

[25] Common Object Request Broker Architecture (CORBA). https://www.omg.org/spec/CORBA.

[26] Dataproc 2.0.x release versions. https://cloud.google.com/dataproc/docs/concepts/versioning/dataproc-release-2.0.

[27] Docker. https://github.com/docker.

[28] Gaia-X: A Federated Secure Data Infrastructure. https://www.gaia-x.eu/.

[29] Google Cloud Dataproc. https://cloud.google.com/dataproc/.

[30] Google Cloud Storage FUSE. https://cloud.google.com/storage/docs/gcs-fuse.

[31] Google Cloud, Storage Transfer Service. https://cloud.google.com/storage-transfer-service.

[32] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine.

[33] HashiCorp State of Cloud Strategy Survey. https://www.hashicorp.com/state-of-the-cloud.

[34] IBM WebSphere Application Server. https://www.ibm.com/products/websphere-application-server.

[35] Instance groups, Google Compute Engine. https://cloud.google.com/compute/docs/instance-groups.

[36] Kubernetes. https://github.com/kubernetes/kubernetes.

[37] Microsoft BizTalk Server documentation. https://learn.microsoft.com/en-us/biztalk/.

[38] MLFlow. https://mlflow.org/.

[39] MongoDB. https://github.com/mongodb/mongo.

[40] MuleSoft CloudHub. https://www.mulesoft.com/platform/saas/cloudhub-ipaas-cloud-based-integration.

[41] MySQL. https://www.mysql.com/.

[42] Optimize Spark performance, Amazon EMR. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-performance.html.

[43] PostgreSQL. https://www.postgresql.org/.

[44] Presto. https://github.com/prestodb/presto.

[45] Redis. https://github.com/redis/redis.

[46] s3fs. https://github.com/s3fs-fuse/s3fs-fuse.

[47] SAP Integration Suite. https://www.sap.com/products/technology-platform/integration-suite.html.

[48] SparkSQL. https://spark.apache.org/sql/.

[49] Spot Fleet, AWS EC2. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html.

[50] TensorFlow XLA. https://www.tensorflow.org/xla.

[51] Terraform. https://www.terraform.io/.

[52] The Cloud Imperative For Software and Platforms, Accenture. https://www.accenture.com/_acnmedia/PDF-139/Accenture-The-Cloud-Imperative-Software-Platforms-Industry.pdf.

[53] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[54] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[55] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[56] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[57] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[58] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NIPS Workshop on Machine Learning Systems (LearningSys'16)*, 2016.

[59] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[60] John Forrest and Robin Lougee-Heimer. Cbc user guide. In *Emerging theory, methods, and applications*, pages 257–277. INFORMS, 2005.

[61] Richard J Forrester and Noah Hunt-Isaak. Computational comparison of exact solution methods for 0-1 quadratic programs: Recommendations for practitioners. *Journal of Applied Mathematics*, 2020, 2020.

[62] José A.B. Fortes. Sky computing: When multiple clouds become one. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 4–4, 2010.

[63] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[64] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[65] Qin Jia, Zhiming Shen, Weijia Song, Robbert Van Renesse, and Hakim Weatherspoon. Supercloud: Opportunities and challenges. *ACM SIGOPS Operating Systems Review*, 49(1):137–141, 2015.

[66] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[67] Hanqing Liu, Jingtian Zhou, Wei Tian, Chongyuan Luo, Anna Bartlett, Andrew Aldridge, Jacinta Lucero, Julia K Osteen, Joseph R Nery, Huaming Chen, Angeline Rivkin, Rosa G Castanon, Ben Clock, Yang Eric Li, Xiaomeng Hou, Olivier B Poirion, Sebastian Preissl, Antonio Pinto-Duarte, Carolyn O'Connor, Lara Boggeman, Conor Fitzpatrick, Michael Nunn, Eran A Mukamel, Zhuzhu Zhang, Edward M Callaway, Bing Ren, Jesse R Dixon, M Margarita Behrens, and Joseph R Ecker. DNA methylation atlas of the mouse brain at single-cell resolution. *Nature*, 598(7879):120–128, October 2021.

[68] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[69] A. Matsunaga, J. Fortes, K. Keahey, and M. Tsugawa. Sky computing. *IEEE Internet Computing*, 13(05):43–51, sep 2009.

[70] André Monteiro, Joaquim S. Pinto, Cláudio J. V. Teixeira, and Tiago Batista. Sky computing: Exploring the aggregated cloud resources - part i. In *Conference: Information Systems and Technologies (CISTI)*, 2021.

[71] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[72] Raghunath Othayoth Nambiar and Meikel Poess. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1049–1058. VLDB Endowment, 2006.

[73] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, , and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *VLDB DISPA Workshop 2020*.

[74] OpenAI. AI and Compute. https://openai.com/blog/ai-and-compute/, 2018.

[75] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. End-to-end optimization of machine learning prediction queries. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 587–601, New York, NY, USA, 2022. Association for Computing Machinery.

[76] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[77] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[78] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. *International Conference on Learning Representations (ICLR)*, 2016.

[79] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[80] Statista. Infographic: Amazon leads $150-billion cloud market. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

[81] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[83] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A Kozuch, and Gregory R Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. *Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep.*, 2016.

[84] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[85] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[86] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: Wan-aware optimization for analytics queries. In *OSDI*, volume 16, pages 435–450, 2016.

[87] Sarah Wang and Martin Casado. The Cost of Cloud, a Trillion Dollar Paradox. https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization/.

[88] Joe Weinman. Intercloudonomics: Quantifying the value of the intercloud. *IEEE Cloud Computing*, 2(5):4047, September 2015.

[89] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[90] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 292–308, New York, NY, USA, 2013. Association for Computing Machinery.

[91] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557, Oakland, CA, May 2015. USENIX Association.

[92] Zizhen Yao, Hanqing Liu, Fangming Xie, Stephan Fischer, Ricky S Adkins, Andrew I Aldridge, Seth A Ament, Anna Bartlett, M Margarita Behrens, Koen Van den Berge, Darren Bertagnolli, Hector Roux de Bézieux, Tommaso Biancalani, A Sina Booeshaghi, Héctor Corrada Bravo, Tamara Casper, Carlo Colantuoni, Jonathan Crabtree, Heather Creasy, Kirsten Crichton, Megan Crow, Nick Dee, Elizabeth L Dougherty, Wayne I Doyle, Sandrine Dudoit, Rongxin Fang, Victor Felix, Olivia Fong, Michelle Giglio, Jeff Goldy, Mike Hawrylycz, Brian R Herb, Ronna Hertzano, Xiaomeng Hou, Qiwen Hu, Jayaram Kancherla, Matthew Kroll, Kanan Lathia, Yang Eric Li, Jacinta D Lucero, Chongyuan Luo, Anup Mahurkar, Delissa McMillen, Naeem M Nadaf, Joseph R Nery, Thuc Nghi Nguyen, Sheng-Yong Niu, Vasilis Ntranos, Joshua Orvis, Julia K Osteen, Thanh Pham, Antonio Pinto-Duarte, Olivier Poirion, Sebastian Preissl, Elizabeth Purdom, Christine Rimorin, Davide Risso, Angeline C Rivkin, Kimberly Smith, Kelly Street, Josef Sulc, Valentine Svensson, Michael Tieu, Amy Torkelson, Herman Tung, Eeshit Dhaval Vaishnav, Charles R Vanderburg, Cindy van Velthoven, Xinxin Wang, Owen R White, Z Josh Huang, Peter V Kharchenko, Lior Pachter, John Ngai, Aviv Regev, Bosiljka Tasic, Joshua D Welch, Jesse Gillis, Evan Z Macosko, Bing Ren, Joseph R Ecker, Hongkui Zeng, and Eran A Mukamel. A transcriptomic and epigenomic cell atlas of the mouse primary motor cortex. *Nature*, 598(7879):103–110, October 2021.

[93] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[94] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[95] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 283–298, USA, 2017. USENIX Association.

A Appendix

A.1 Implications and Economics of the Sky

While the body of this paper was firmly rooted in what an intercloud broker could offer now, we turn our attention to the future and ask: what are the implications of the Sky for the future cloud ecosystem? This section is inherently more speculative, so we have included it as an appendix to provide some context for where we think this approach could take us.

A.1.1 Embracing Diversity

While there is an increase in limited interface compatibility, in the overall ecosystem there is an increasing diversity in terms of location and hardware. The aforementioned regulatory concerns require greater flexibility in location; Sky Computing provides an easy way to specify the necessary location constraints. However, there are two other important location considerations. First, some tasks should be run on nearby edge clouds to lower latencies between client and cloud. Second, some tasks should be on on-premise clusters, rather than public clouds, to lower costs (see [87] for an argument as to why this is crucial). These concerns can be met by bringing edge and on-premise clouds into the Sky. The intercloud broker could then automatically send jobs to the closest edge cloud (if lowering latency is important) or to the on-premise cloud (if lowering costs is important and there is enough capacity).

In addition, by allowing users to specify specific hardware requirements in their request, one can automatically seek out clouds that have the appropriate hardware support. Or one can merely ask for high performance, and the intercloud broker will find the highest-performing cloud for that task, regardless of how they achieve it. Thus, Sky Computing turns the diversity of the current clouds from an impediment to an advantage: as long as one cloud meets a user's needs in terms of location or hardware or other constraints, the intercloud broker will find it.

A.1.2 Economic Analysis

For analytical convenience here we assume that in the future clouds will fall into two categories. Some clouds will remain *proprietary*, offering their own APIs for some tasks and charging for data egress in an attempt to keep customers tied to their cloud. However, others will join the Sky and become a *commodity* cloud in that they fully embrace the open source interfaces and do reciprocal data peering with other clouds that have joined the Sky. The economic choice facing clouds is which of these alternatives they choose. Note that even proprietary clouds can be used by the intercloud broker, but doing so may entail data egress charges.

The choice facing consumers is which of these two types of clouds they choose to use: do they send their workloads to a single proprietary cloud, or do they let the intercloud broker find which clouds to run on? In what follows, we assume that users attempt to optimize some measure of price and

performance of each task; we will denote this metric by P2, and define it so that smaller values are better. The relative importance of price and performance will differ between users, but we do not address that here as it overcomplicates the analysis without adding much insight; instead, we assume all users attempt to minimize the same measure P2. We now analyze, in a vastly oversimplified model, how the ecosystem of clouds might evolve given this consumer behavior.

Denote by R the set of proprietary clouds and denote by S the set of commodity clouds (i.e., the Sky). Assume that the workload from user α consists of a set of tasks j, with a weight or frequency w_j^α that represents the fraction of their workload that consist of task j. Note that this analysis can either apply to individual applications (which involve a DAG of tasks), or an overall workload.

The P2 of task j on cloud c is denoted by P_j^c. If a cloud does not support that task, P_j^c is set to be infinite. Let $\tilde{P}_j^c$ be the P2 taking into account the delays (and perhaps egress charges, if a proprietary cloud is used) in sending data between different clouds. We then define P_j and $\tilde{P}_j$ as the minimal P2's achievable (the latter taking into account the extra inter-cloud delays and cost, and the former not): $P_j = \min_{c \in S \cup R}[P_j^c]$ and $\tilde{P}_j = \min_{c \in S \cup R}[\tilde{P}_j^c]$.

Assume for simplicity that these workloads are either sent to the Sky (i.e., placement determined by the intercloud broker), or to a single proprietary cloud. Given these assumptions, if the workload is sent to a proprietary cloud, the user α will choose the cloud $c \in R$ that minimizes $\sum_j w_j^\alpha P_j^c$; call this cloud $c(\alpha)$. If sent to the Sky, then the overall P2 is $\sum_j w_j^\alpha \tilde{P}_j$. Given our assumptions, a user will pick between $c(\alpha)$ and the Sky, depending on whether the sum $\sum_j w_j^\alpha [P_j^{c(\alpha)} - \tilde{P}_j]$ is positive (Sky) or negative (proprietary cloud $c(\alpha)$). Note that since by definition $P_j \leq P_j^{c(\alpha)}$ this can only be negative if the inter-cloud delays or costs are significant.

The question a cloud faces is whether to join the Sky or not. If it remains a proprietary cloud, the only customers it gains are those for whom its overall average P2 is best: i.e., for those users for whom it is $c(\alpha)$. If it joins the Sky, it gains revenue for each task j where its performance is best among the clouds (taking into account the inter-cloud delays).

Assuming most users have a broad workload including many tasks, this analysis suggests that a cloud should only remain proprietary if it can compete across a broad collection of tasks. Joining the Sky becomes the rational choice for clouds who realize they cannot compete broadly, but can find narrower market niches (i.e., sets of tasks) where they excel.

Note that two proprietary clouds compete in a zero-sum manner: for users sending their workloads to proprietary clouds, either one gets the business or the other. Sky clouds compete in a much different way. Of course, they all compete to provide the best P2 implementations for each task. However, a cloud providing a superior solution for one type of task *helps* a cloud focusing on other types of tasks, because users will only use the Sky if the overall service they get is better than that on proprietary clouds. Thus, the ecosystem of Sky clouds combines *competition* on each task type with *collaboration* to provide high-quality support across a broad spectrum of tasks. This is the interdependence in the Sky.

This analysis is obviously oversimplified in many dimensions. For instance, users make different tradeoffs between cost and delay, and workloads are more complicated than just a linear combination of tasks. However, none of these considerations undercut the general observation above that proprietary clouds must be prepared to compete across a wider range of tasks (since their egress charges and proprietary interfaces *purposely* reduce the likelihood of users offloading to other clouds).

For a fledging cloud provider, it seems clear that joining the Sky is the preferable choice. These new clouds can concentrate on narrow sets of tasks where they can compete favorably with existing commodity and proprietary clouds, and they need not worry about marketing as the intercloud brokers will seek out the best P2 available.

None of these results are surprising, as the intercloud broker effectively sets up a two-sided market. Two-sided markets are common, and they are typically opposed by market actors who have high margins and want to preserve them, but are welcomed by those struggling to get a foothold in the market and who cannot otherwise overcome the inherent advantages of the dominant market players (such as much better name recognition, much larger sales forces, etc.). In the current cloud market only Amazon and perhaps Azure can be seen as having dominant market positions; all other cloud providers have less than 10% of the market [80]. For all of these other cloud providers, which comprise roughly half of the current cloud market, the Sky may be the preferable choice.

A.1.3 Speculation

In many ways, the intercloud broker is merely a mechanism that turns cloud computing into a more competitive market. However, efforts to create the Sky will be for naught if the currently dominant clouds remain dominant and proprietary even after the intercloud broker is put in place. Here we speculate briefly on the factors that will play a critical role in how the competition plays out. We start with four basic assumptions:

Sky-based clouds may innovate faster: Sky clouds need not market their technologies; they merely need to post faster speeds and/or lower prices for various workloads. Thus, the intercloud broker itself speeds innovation because workloads will automatically follow the better P2s, no matter how they arose. In addition, Sky clouds can focus their innovative energies on narrow classes of tasks where they might have special expertise (e.g., Oracle for databases) or special hardware (e.g., Samsung for storage, Google for TPUs, NVIDIA for GPUs). In fact, this is already happening; see the recent announcements by Nvidia, Equinix, and Cirrascale [12].

Large clouds have economies of scale: There are undeni-

able advantages to operating a cloud at scale, such as greater leverage with suppliers and the ability to amortize various infrastructure costs over larger deployments. These advantages may be the single biggest barrier to the success of Sky.

Infrastructure providers might provide smaller clouds with better economies of scale: Infrastructure providers, such as Equinix, who have experience in building out clouds and who can amortize infrastructure costs, can help smaller clouds with deployment. This will not match the economies of scale of the largest clouds, but will allow small clouds to be deployed with reasonable efficiency.

Small clouds are not necessarily small companies: One worry is that the proprietary clouds would engage in predatory pricing to prevent the Sky from emerging. However, many companies that will deploy Sky-based clouds will be using them as showcases for their technology (Samsung for storage, Oracle for database workloads, etc.), and they have very deep pockets. So predatory pricing will actually hurt the large clouds more than the smaller ones (because they have smaller market share, their losses are smaller).

Based on these assumptions, the crucial question is whether the rate of innovation of the smaller clouds (which can be more narrowly targeted) is sufficient to compensate for their disadvantage in economies of scale (which is mitigated by infrastructure providers). We have no wisdom to offer on this central but speculative question. However, with innovative companies like Google, IBM, and Alibaba counted as "small clouds" likely to join the Sky rather than remain proprietary, we believe that there is a significant chance that the Sky could emerge as an economically viable alternative to the current cloud ecosystem.

A.2 ML Training on Spot Instances Across Clouds

In §5.1 we evaluated SkyPilot's benefits for ML pipelines; here, we show an additional experiment to demonstrate that SkyPilot can run a single ML training job on spot instances across clouds, improving resource availability and reducing costs. In the event of spot instance preemptions, SkyPilot supports migrating a job to another zone, region, or cloud where spot instances are available. We consider training a BERT model with a V100 GPU on a subset of Wikipedia, WikiText-103 (0.5 GB), for 30 epochs. For failure recovery, we save the current model checkpoint (1.5 GB) periodically to a persistent storage. Each epoch runs for around 40 minutes and each checkpointing incurs an overhead of 0.5 minutes.

We evaluate three different strategies to run the job:

- *On-Demand*: runs on an on-demand instance on AWS.
- *SingleRegion*: runs on a spot instance in a single AWS region, `us-east-1`.[11]
- *Broker*: runs on a spot instance, with SkyPilot having the freedom to choose among all US regions of AWS or GCP.

[11] We chose it as it had the lowest preemption rate at the time of experiment among all US regions. Spot hourly price was $0.91, vs. on-demand's $3.06.

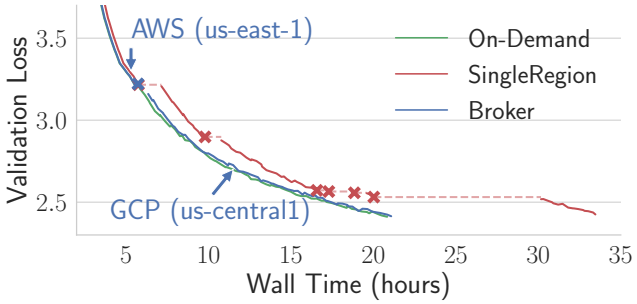


Figure 11: Loss curves of training BERT on V100 for 30 epochs. Each **x** marker is a preemption event; gaps between segments are the time periods when spot instances are not available. After the first preemption event, Broker migrates the job from AWS `us-east-1` to GCP `us-central1`, while SingleRegion waits in the same region.

	Cost	Makespan
On-Demand	$61.2	**20 hrs**
SingleRegion	$21.8	34 hrs
Broker	**$18.4**	21 hrs

Table 5: Costs and makespan for the three strategies to finish BERT training. Data transfer and checkpointing overheads are included.

For a fair comparison, we launch all strategies at the same time and in the same starting region. With SingleRegion, if no spot instances are available in the region when a preemption happens, it waits until they become available again and then resumes the job from the latest checkpoint. With Broker, if no spot instances are available it immediately triggers re-optimization and searches for availability in other regions and clouds; if found, SkyPilot transfers the data/model checkpoint to the new location and resumes the job there. The cost of each data and checkpoint egress across clouds is $0.2.

Figure 11 plots the validation loss curve for each strategy. Around hour 6, the spot instances used by both the SingleRegion and Broker strategies get preempted. SingleRegion sticks with the same region (`us-east-1`), but needs to wait for 3 hours (dashed line) to get a new spot instance. In contrast, Broker searches for spot instances in other AWS regions, which fail to provide capacity, before finding availability in GCP's `us-central1` region. After hour 6, the SingleRegion job experiences several more preemptions which cause further delays. Overall, the delays from using a single region adds more than 10 hours to the completion time.

Table 5 shows the total cost and makespan for the three strategies. Broker finishes ∼40% faster than SingleRegion because it can leverage spot instance availability across regions and clouds. Moreover, Broker is 10% cheaper than SingleRegion: despite the cross-cloud data egress costs incurred by Broker, the faster recovery time and fewer preemptions (thus, less lost progress) reduce the overall cost compared to SingleRegion. Compared to On-Demand, Broker saves 70% cost due to lower spot prices, while incurring a minimal overhead in makespan (∼5%) due to job recovery and checkpointing.

Unlocking unallocated cloud capacity for long, uninterruptible workloads

Anup Agarwal[†], Shadi Noghabi[‡], Íñigo Goiri[§], Srinivasan Seshan[†], Anirudh Badam[‡]
[†]*Carnegie Mellon University,* [§]*Azure Systems Research,* [‡]*Microsoft Research*

Abstract

Cloud providers auction off unallocated resources at a low cost to avoid keeping hardware idle. One such mechanism is Harvest VMs (HVMs). These VMs grow and shrink as the unallocated resources in a server change. While HVMs are larger in size and less prone to eviction compared to other low-cost VMs, their resource variations severely slow down long-running, uninterruptible (hard to checkpoint/migrate) workloads. We characterize HVMs from a major cloud provider and discover large spatial variations in their stability and resources. We leverage this diversity by predicting which HVMs will be stable enough to run tasks without preemptions. We use the predictions to inform scheduling and resource acquisition decisions. Our evaluation with real workloads shows that we can reduce mean and tail (90th percentile) job completion times by 27% and 44% respectively, at 75% lower cost than regular VMs.

1 Introduction

Motivation. Failure to monetize idle hardware in cloud deployments is a huge opportunity cost for cloud providers. Providers typically provision hardware for peak demand, with little over-subscription, to deliver an illusion of elastic resources with strong isolation and performance guarantees. Due to variations in demand, 25–30% hardware sits idle [3]. Many proposals try to address this problem, including better resource packing using abstractions like FaaS (Function-as-a Service), and auctioning unallocated capacity (unreliably) using Spot or Burstable VMs [2, 18, 49, 74]. A latest advancement towards isolating and exposing unallocated resources is Harvest Virtual Machines [3].

Harvest VMs (HVMs) are variable-sized VMs co-located with other regular (on-demand, high-priority) VMs. When a new regular VM is allocated to a server, the HVM shrinks in capacity, and when a regular VM finishes, it grows. This agility allows HVMs to gather 2.5–7.5× more resources compared to other low-priority low-cost fixed-sized VMs (e.g., Spot VMs) at lower eviction rates (§2.1, [3]). This creates a new opportunity and a new challenge.

Large harvested capacity overcomes a major capacity bottleneck [21, 60] allowing many large-scale applications [15, 16, 34, 47, 67, 68, 81, 89] in the financial, scientific, genomics, energy and meteorology sectors to run economically in the cloud. However, HVM's resource variations can significantly slow down these long, uninterruptible (hard to checkpoint/migrate) applications due to preemptions (§2.3).

Prior efforts try to mask such overheads using scheduling, resource acquisition, and load-balancing techniques (§5). Unfortunately, these efforts do not fit well for the combination of HVMs and long, uninterruptible workloads. They either address only VM evictions (not resource variations exhibited by HVMs), or rely on the unique properties of Spot markets. On the workload side, they often use a combination of checkpointing, migration, replication, or application level changes. These are prohibitive or impractical as uninterruptible workloads have large working sets, run at large scale, and rely on many domain-specific libraries and frameworks with complex state stored in memory (§2.2).

Our work. We seek to answer: *"How can we best use HVMs to run long, uninterruptible workloads?"* We begin by characterizing HVMs and a collection of long, uninterruptible production workloads from a major cloud provider. We find large spatial diversity in the stability and resources of HVMs, i.e., some HVMs are more stable (change less often) or get more resources than others. Simultaneously, we observe large diversity in the runtimes of tasks in our workloads.

We leverage our observations in two ways to build SLACKSCHED. First, we build a scheduling component that avoids preemptions by better matching tasks to HVMs, i.e., runs longer tasks on more stable HVMs and vice versa. Second, we build a resource acquisition component that improves overall stability of the HVM pool by retaining relatively stable HVMs and continuously de-allocating unstable HVMs.

In building SLACKSCHED, a key technical challenge is identifying which HVMs are going to remain stable in the future. Resource variations in HVMs can depend on a number of factors which are hard to predict or control (e.g., the arrivals, lifetimes, and placement of regular VMs). We work around this using our insight that the distribution of time between HVM resource changes is relatively stationary over time. We use this to estimate when new resource changes are likely to occur and match tasks to HVMs that are likely to not change during task lifetimes.

We implement our scheduler as a pluggable component of YARN [5], a popular cluster orchestrator, and the acquisition component as a module that manages resource negotiation between the cloud provider and YARN. We evaluate SLACKSCHED under a variety of production workloads, operating conditions, and HVM environments considering

HVM traces collected from multiple regions and time periods. We find that SLACKSCHED reduces mean and tail (90th percentile) job completion times by 27% and 44%, respectively.

We note that our system does not make any assumptions about, nor is reliant on, the cloud provider's allocation policy. The diversity of unallocated resources is fundamentally tied to the diversity in regular VM workloads. As evidence, we consider future sources of resource variability, e.g., if unallocated resources change in capacity based on variations in power supply to the data center due to renewable energy sources [13, 23], in addition to variations due to regular VM arrival/departure. We find SLACKSCHED has similar performance in this new environment.

Summary. We make the following contributions:

- We characterize the behavior of real-world production HVMs and long, uninterruptible cloud workloads (§2).
- We build a practical method to estimate future variations in HVMs (§3.1.1, §3.1.2).
- We design and implement SLACKSCHED, a system that enables the use of HVMs for large-scale, long-running, uninterruptible workloads (§3, §3.3).
- We show that our scheduling and resource acquisition decisions are effective in mitigating the overheads of HVMs for varied workloads and environments (§4).

2 Characterization & Motivation

We first characterize HVMs (§2.1) and long-running uninterruptible workloads (§2.2). Then, we focus on the overheads of running these workloads on HVMs (§2.3). Our characterization reveals two opportunities that motivate our design (§2.4). We detail in §5 and Appendix B.1 why past efforts at managing resource variability and building reliable infrastructure out of unreliable services are ineffective for the combination of uninterruptible workloads and Harvest VMs.

2.1 Harvest VMs

Background. HVMs dynamically expand and contract to leverage the unallocated resources left by regular VMs. As more (or fewer) on-demand VMs are placed on a server, an HVM will shrink (or grow) its core count. We focus on HVMs that harvest CPU cores but our work can be leveraged when harvesting other resources (e.g., memory [32] and storage). For Spot VMs to expose the same capacity as HVMs, one needs to provision more and/or larger size Spot VMs. This significantly increases the number of evictions to handle and the management overheads (e.g., more copies of the OS).

HVMs are configured with a minimum size (e.g., {2,4,8} CPU cores and {16GB, 32GB, 64GB} of memory). If an HVM needs to shrink below its minimum size (e.g., because of on-demand VMs), it will be evicted. HVMs are overall cheaper in price than both Spot and on-demand VMs. Today, HVM's minimum size is charged at the Spot VM discount (e.g., 48% to 88% cheaper than regular VMs [87]), and each harvested core has a further discounted price.

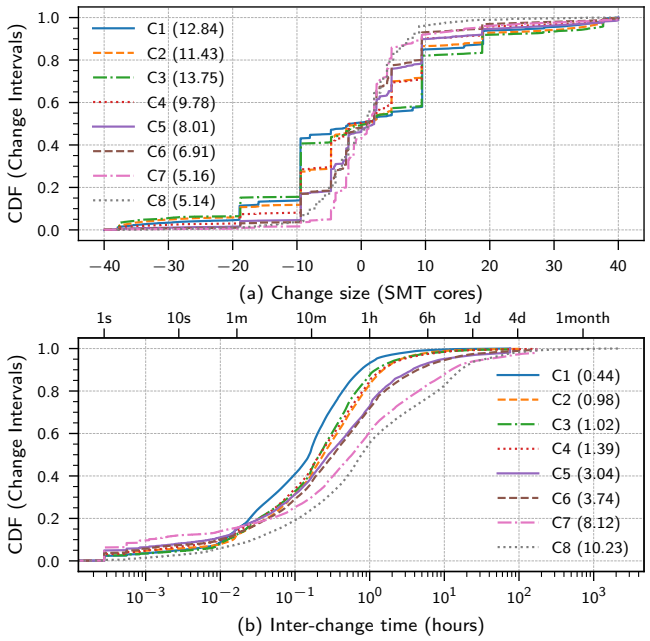


Figure 1: Resource variation in HVMs across clusters. The parenthesis in the legend lists (a) the mean magnitude of change size and (b) mean inter-change time (hrs). The clusters are sorted in increasing order of the mean inter-change time (C1–C8).

Overview. Prior work has studied properties of HVMs at an aggregate level [3, 87]. However, to understand how broader workloads might be impacted, we analyse HVMs at an individual level and answer: (1) *stability of HVMs:* how often and by how much do HVMs change?[1] (2) *spatial and temporal diversity of HVMs:* how do different HVMs compare and how do they change over time? (3) *impact of workloads:* how do the runtimes of long, uninterruptible workloads compare to the resource variations of HVMs? With this goal, we study HVM traces (from March 2019 and August 2021) for 8 clusters across 5 regions of a major cloud provider.

Stability. We measure the resource changes in terms of *size:* number of added/removed cores, and *frequency:* time between two consecutive changes (inter-change time or change interval). We count HVM evictions as a resource change to size 0.[2] We observe that different clusters witness different amounts of activity from regular VMs, so we order the clusters, with lower activity clusters on the bottom (C7–C8 in Figure 1).

Size of changes. Figure 1(a) shows the size of changes in HVM resources across clusters. Positive changes signify resource growths and negative changes signify shrink events. The mean magnitude of change is between ≈ 5 and 13 SMT cores (simultaneous multi-threaded cores or hardware threads) for different clusters. These are large variations, given the typ-

[1] While this aspect has been considered in [87], it was in the context of short-running FaaS workloads and considered only a single cluster. Hence, we revisit it in the context of our target workloads for more clusters.

[2] We do not separately study HVM evictions as these occur rarely relative to task durations in our workloads ([3], §2.2).

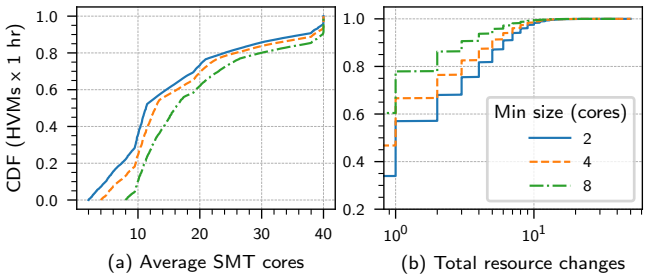


Figure 2: Spatial Variation — HVMs on different servers differ in amount of resources harvested and stability.

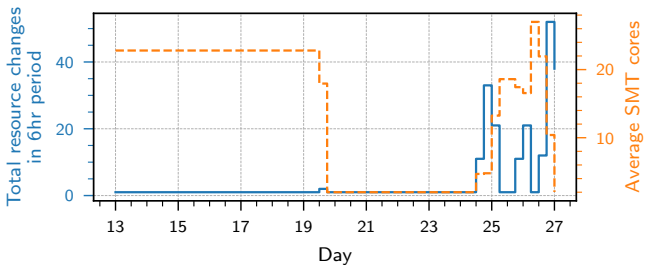


Figure 3: Temporal Variation in a single HVM.

ical minimum size of 2–8 SMT cores for an HVM and given that 1–2 cores is a popular regular VM size [25, 41].[3] Such large variations can have significant performance implications on applications (both positive and negative).

Frequency of changes. Figure 1(b) shows that the inter-change times have high variance and long tails. For higher activity clusters (C1–C6), we see 93–72% of changes within an hour, a mean inter-change time of 26–224 minutes, and a 95th percentile of 1.2–11.2 hours. For the lower activity clusters (C7–C8), we see 61–55% changes within one hour, a mean of 8–10 hours, and 95th percentile of a few days.

There are long time intervals without any resource changes (e.g., multiple days) as well as short intervals (e.g., 84% within one hour, 40% within 10 minutes, 16% within 1 minute for cluster C2). Ideally, we want to get the most out of long intervals without resource changes while coping with short change intervals. To this end, we analyse how the change intervals are distributed across space (HVMs) and time. This helps understand if there are periods of high activity or if changes are spread spatially. We show this analysis for a high activity cluster (C2). Other clusters exhibit similar trends but differ in the frequency and magnitude of variations.

Spatial diversity. The behavior of Spot and on-demand VMs is determined by the VM configuration and the region. However, HVMs with the same configuration (e.g., minimum size) can behave differently depending on the server they land on (even within the same region). This diversity is directly tied to competing VMs on the server as an HVM shrinks when new competing VMs are allocated and grows when competing VMs are deallocated.

For each HVM, for each 1-hour time window, we measure the harvested cores (time-averaged over the 1 hour) and stability (number of changes), shown in Figure 2. HVMs with a minimum size of 2, 4, and 8 get an average of 15, 17, and 20 cores respectively, which is 2.5–7.5× more resources than the minimum size. At the same time, the top 10% HVMs get a minimum of 36, 38, and 39 total cores while the bottom 10% get at most 3, 3, and 2 additional cores beyond the minimum size. Given that the additional harvested cores have an additional cost, HVMs will also exhibit a wide cost diversity.

All HVM minimum sizes show similar trends in terms of stability. Figure 2(b) shows that larger minimum size HVMs tend to be slightly more stable than smaller ones on average. However, a specific larger minimum size instance is not guaranteed to be stable. The worst 10% HVMs can witness tens of changes in an hour while the relatively stable 50% HVMs witness up to one change in an hour.

Temporal variation. The stability and resources of a particular HVM can also change over time (e.g., a stable HVM can start changing resources frequently). Figure 3 shows an example HVM over a 15 day period (aggregated over 6h windows to gauge longer term stability). For the first ≈ 4 days the HVM is very stable with over 40 harvested cores. However, towards the end, the HVM witnesses large number of resource variations.

2.2 Target Workloads

We focus on long-running and uninterruptible (hard to checkpoint or migrate) workloads. Many applications fall into this category, such as workloads in genomics, oil and gas, weather & financial simulations, geo-spatial workloads, and many scientific computing tasks [15, 16, 34, 47, 67, 68, 81, 89]. The market for these workloads is worth tens of billions of dollars with all major cloud providers pushing towards bringing them to cloud environments [35, 45–47, 59, 67, 68].

These workloads are often run at large scale. Thus, currently they are predominantly run in on-premise clusters that are perceived to be cheaper (compared to regular VMs) and more reliable (compared to Spot VMs, due to evictions). HVMs with their high resource availability and lower eviction rates pave the way for *economically* and *reliably* running these workloads in cloud environments. However, tasks in these workloads tend to be long relative to the typical change intervals of HVMs, hence a single task may see multiple resource variations leading to thrashing/preemptions. These workloads often use domain-specific libraries in containerized environments with large working sets [19, 34], making checkpointing entire containers prohibitive and making tailored checkpoints impractical due to the domain-specific nature of the code coupled with a rich ecosystem where new libraries are continuously added. Further, users of these applications are typically reluctant to modify applications [30, 61].

For concreteness, we study two large-scale production ap-

[3] A VM advertised with 2 cores may be mapped to a fractional amount of SMT cores, e.g., 1.5 or 2.5 SMT cores, depending on the VM's oversubscription or headroom.

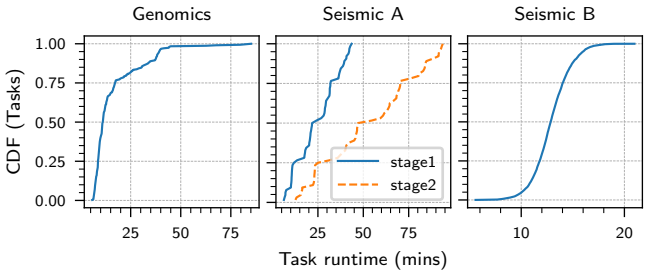


Figure 4: Analysis of task runtimes for two workloads (Genomics analysis and Seismic simulations).

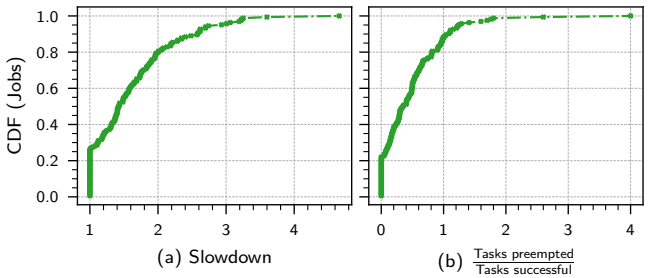


Figure 5: Impact of HVM resource variations on workloads. (a) HVM resource variations slow down jobs. Geometric mean slowdown is 1.5× with some jobs even slowing down by more than 4×. (b) HVM resource shrinks cause the preemption of a third of all launched tasks, i.e., for each successful task, there are 0.5 failed tasks on average. The same task might be preempted multiple times causing preemptions per successful task to grow beyond 1.

plications from a major cloud provider: (1) an application in the oil and gas domain that performs seismic imaging simulations [15, 81, 89], and (2) a genomics application that analyses genetic material to identify various properties (e.g., disease susceptibility for humans and physical traits of plants) [34].

To understand the impact of HVMs on these workloads, we contrast the HVM resource change intervals (representing supply variability) with the task execution times of these workloads (representing demand requirements). Figure 4 shows that tasks in both example applications are long, with a median of tens of minutes, and tails of over 75 minutes. Task distributions for different applications have various shapes (e.g., Uniform and Gaussian for Seismic A & B, and Bounded Pareto for Genomics). Task runtimes range from a few minutes to an hour either within a single job or across different jobs (for Seismic B, not shown). In other words, there is a sizable overlap in the range of the task runtimes and the range of HVM inter-change times. This means that tasks in these workloads are likely to witness a few resource change events during their execution on typical Harvest VMs.

While Gaussian-distributed runtimes are common in typical cloud workloads, Pareto and Uniform distributions show up because of concurrent tasks being heterogeneous. In the genomics applications, some tasks do the actual analysis while others do verification or data transformation. In the seismic simulations, different tasks perform imaging at different resolution or area/volume depending on the analysis requirements.

2.3 Running Workloads on HVMs

The unpredictable and arbitrary resource changes, especially shrink events, can cause tasks to slow down, thrash (for memory harvesting VMs [32]), or get preempted altogether. This leads to execution time overheads and resource wastage since preempted tasks need to be restarted from scratch (under uninterruptible workloads) wasting any previous progress. We measure these overheads comparing HVMs to on-demand VMs running a mix of our target workloads. We use trace-driven simulation for this analysis (methodology in §4).

Figure 5(a) shows the *slowdown* of jobs running on HVMs. We define job slowdown as:

$$\text{Slowdown(Job)} = \frac{\text{ExecutionTime(Job) on HVMs}}{\text{ExecutionTime(Job) on regular VMs}}$$

When running on out-of-the-box HVMs, the geometric mean job slowdown is 1.5× while some jobs are delayed by more than 4×. Such high slowdowns make HVMs impractical for many scenarios. This slowdown is caused by tasks being preempted when the HVM shrinks. For instance a task that needs 4 cores is preempted when the HVM shrinks to 2 cores (or when the HVM evicts, i.e., shrink to size 0). Figure 5(b) shows the distribution of tasks preempted across jobs. Around a third of all tasks fail which leads to an additional resource consumption of ≈ 20%. This directly translates into 20% more cost. SLACKSCHED's goal is to *make the execution of long uninterruptible workloads on HVMs similar to their execution on regular VMs.*

2.4 Opportunities for Improvement

To improve the efficiency of our target workload on HVMs, we build on the following observations:

- There is a large diversity in both task runtimes (within a job or across jobs) and HVM inter-change times and they have significant overlap in their ranges. This provides an opportunity to match long tasks to more stable HVMs and short tasks to unstable HVMs, thus allowing efficient use of HVMs by minimizing preemptions and reducing costs.
- Some HVMs are more stable than others and the stability of a HVM can change over time. There is an opportunity to improve workload execution times by acquiring and retaining a higher fraction of instantaneously stable HVMs.

These two implications motivate our design for two HVM-tailored components: (1) *Scheduler* that matches tasks to HVMs and (2) *Acquirer* that continuously maintains a relatively stable mix of HVMs. Leveraging these insights is not straightforward as the behavior of HVMs depends on multiple unknown factors (e.g., regular VM arrivals and departures).

3 SLACKSCHED Design

SLACKSCHED manages application execution on HVM clusters rented by a cloud user. Many applications running in the cloud run on top of cluster orchestrators like YARN, SLURM,

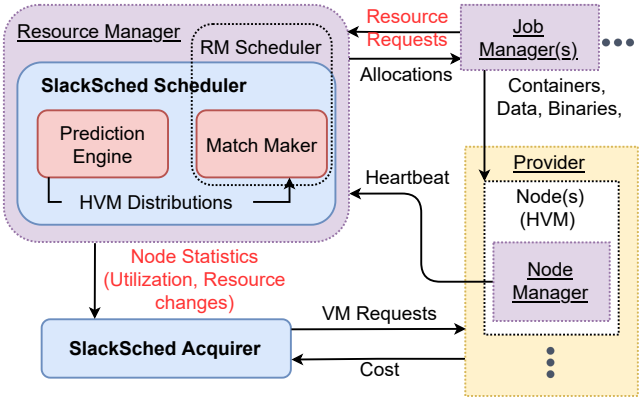


Figure 6: SLACKSCHED architecture with new components in blue and modifications shown in red and red.

Mesos, and others [5, 48, 53, 73], or their managed variants (e.g., Azure Batch [14], AWS Batch [9], GCP Batch [36], AKS [11], AWS EKS [55], and GKE [54]). SLACKSCHED works within such orchestrators, replacing key components to support more efficient use of HVMs. Cloud users can install modified version of orchestrator and/or providers can incorporate SLACKSCHED within managed variants of the orchestrators. Since SLACKSCHED mostly improves execution of individual jobs (§4) (in addition to improving aggregate job execution), providers can use SLACKSCHED to even manage workloads from different cloud users.

Figure 6 shows the architecture of SLACKSCHED and the coordination of three main entities (shown in purple). (1) A per-node *Node Manager*, running on each node in the cluster, that is responsible for reporting the health of the node and running tasks on the node. In the case of HVMs, this is also responsible for conveying instantaneous resource availability on the HVM.[4] (2) A per-job *Job Manager* (or "Application Master" in YARN terminology) that is responsible for managing the progress of the tasks in a single job. (3) A cluster-wide *Resource Manager (RM)* that receives requests for resources from the Job Managers, schedules and maps these requests to nodes, and conveys resource allocations to the job managers. Then, the Job Managers launch containerized programs on nodes based on the allocations. Additionally, we assume that Job Managers annotate their requests with resource requirements and task runtimes. These can be estimated based on input parameters and the type of computation (similar to [25, 50]). The Genomics and Seismic workloads that we consider already have resource requirement annotations. We measure sensitivity to errors in runtime estimates in §4.2.1.

SLACKSCHED consists of two key components (shown in blue): the *Scheduler* (§3.1) and the *Acquirer* (§3.2). The *Scheduler* extends the default orchestrator scheduler and exploits the diversity in task runtimes and HVM resource variations to match tasks to HVMs. The *Acquirer* interacts with the Provider and the Resource Manager to acquire and maintain a set of HVMs that are low cost, harvest more resources, and are stable enough for the workload.

[4]The hypervisor exposes the same number of logical cores to HVMs and only changes their mapping to physical cores at runtime. User programs can query the hypervisor for the core assignment anytime.

3.1 SLACKSCHED Scheduler

Typically, cluster schedulers decide on (1) the order of jobs, (2) the order of tasks within a job, and (3) the placement of tasks onto nodes. SLACKSCHED only affects the third decision and uses the default orchestrator mechanisms for the first two. The SLACKSCHED Scheduler tries to minimize preemptions and improve completion times by intelligently matching tasks to HVMs. Based on the task duration and the stability of HVMs, the Scheduler assigns longer tasks to more stable HVMs (predicted to maintain their resources for a longer time) and shorter tasks to less stable HVMs. Our design splits the operation into: (1) the *Prediction Engine* (§3.1.1) that predicts the stability of each HVM in the future, and (2) the *Match Maker* (§3.1.2) that matches tasks to HVMs based on task duration and HVM stability.

3.1.1 Prediction Engine

Challenges. For match making, we need fine granularity models that predict the resource availability of an individual HVM, e.g., "when will an HVM change its resources?" or "how many resources will it harvest?". This is very challenging since the resource availability of individual HVMs depends on several unknown and uncontrollable factors such as: (1) the arrivals and lifetimes of on-demand VMs, (2) the placement/VM allocation policy of the provider, and (3) the requested configuration of the HVM (e.g., minimum size). Even the cloud provider does not have full future knowledge, especially about when the on-demand VMs will come and go.

Prior efforts at modeling HVM resource availability [3] are not sufficient since they only make predictions at an aggregate level, rather than an individual level. For instance, they provide estimates such as "X% of HVMs will survive in the next hour", or "HVMs will expose on average *Y* number of cores in a specified time window". In addition, point prediction approaches similar to [25, 43, 78, 88], which use various machine learning models including SVMs, CNNs, and LSTMs [62, 75] to model resource availability, are not a good fit for HVM environments for two main reasons: (1) similar historic resource variations may provide widely different behaviors in the future, which makes point estimates inaccurate; (2) HVMs depict large skews in their behavior (§2.1) which makes the use of computational methods such as machine learning hard.

Key insights. Instead of making point estimations, we take an alternate approach of distribution-based predictions and conditional probabilities. This was inspired by prior work on task scheduling with unknown runtimes [63, 69, 83]. These make probabilistic estimates instead of exact predictions, leveraging the fact that the distribution of task runtimes is known

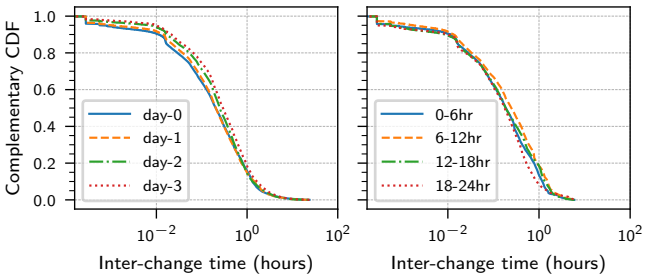


Figure 7: The inter-change-time distribution varies only slightly at different time scales. The legend lists the time period within the trace for which the CDF was computed. The historical inter-change time distribution is a good estimate of the future distribution.

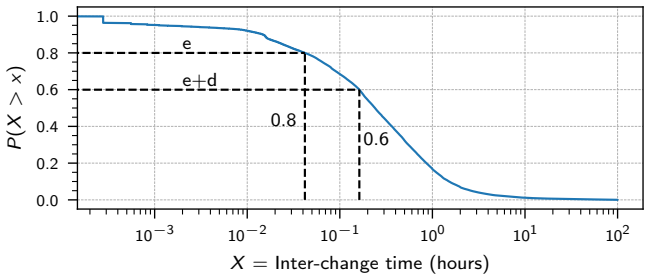


Figure 8: We use the inter-change time distribution for computing the probability that a resource change will occur during a task's lifetime. Say e time has elapsed since the HVM has seen a resource change, and d is the duration of the task. Then, from the complementary CDF, we get: $P(X \leq e+d|X > e) = 1 - P(X > e+d|X > e) = 1 - \frac{P(X>e+d)}{P(X>e)} = 1 - \frac{0.6}{0.8}$

even though the exact runtime is unknown. They proceed by computing the expected remaining runtime by conditioning on the task's progress or age. For instance, given a task has been running for 1 hour, what is the probability that it will run for 30 more minutes.

Similar to prior work, we condition on the inter-change time distribution to estimate the remaining time until the next change for a specific HVM. *We find that the inter-change time distribution is relatively stationary over time. Thus, even though the exact times when resource changes will occur is unknown, the distribution of time between changes is known.* Figure 7 shows the inter-change time distribution at multiple time scales. There are little changes in the distribution over multiple hours/days. As validated in Section 4.5, this approach generalizes to other future sources of variability in unallocated capacity, such as using renewable energy to power a data center, an emerging approach for sustainable cloud computing [12, 76, 77].

Workflow. The *Prediction Engine* maintains a rolling snapshot of the inter-change time distribution in the past D days ($D = 1$ in our implementation) for estimating the *completion probability*, i.e., the probability that a task will complete successfully on an HVM without getting preempted. We use completion probability as a proxy for HVM stability as it allows us to rank if an HVM is stable enough for a task.

We approximate the completion probability for a task and HVM as the probability that the HVM will not shrink during the lifetime of the given task. In reality, tasks can complete successfully despite witnessing resource shrinks, e.g., if the node is underutilized and has enough resources even after shrinking. However, since we only use the completion probability to obtain a relative ranking of nodes, its absolute value is of little consequence. This approximation also allows robustness to inaccuracies in task runtime estimates. Future work can consider predicting shrink size for more accurate estimation of completion probability and potentially better job execution on HVMs.

We compute completion probability in two steps. First, we estimate the likelihood that there will be a resource change event during the lifetime of a task on a particular HVM. Second, we estimate the likelihood that this resource change event will be a shrink event. Finally, the Match Maker combines these likelihoods (§3.1.2). Note, there are other ways to compute completion probability (Appendix B.2).

Likelihood of resource change. To estimate the probability of a change during the lifetime of a task, we condition on the inter-change time distribution (Figure 8). Given that e time has elapsed since the last resource change event, we compute the probability that the next change event will occur within d time from now, where d is the duration of the task.

Likelihood of shrink. To compute the probability for the next event to be a shrink, we perform an n-gram (bigram) analysis [17] on the sequence of resource changes. Specifically, we look at a historical sequence of resource changes and calculate how often a shrink occurs after a growth and how often a shrink occurs after another shrink. We use this to compute the probability that the next event will be a shrink given that the last one was a growth (or shrink) event. *We find that such shrink probabilities are also relatively stationary over time.*

3.1.2 Match Maker

To minimize task preemption likelihood, the *Match Maker* places tasks on HVMs with a high completion probability for the task. When no HVMs with available resources yield a high completion probability, the *Match Maker* may wait for occupied HVMs to free up resources ("delayed scheduling"). We now describe how we calculate the completion probability, perform delay scheduling, and our matching logic.

Completion probability. The *Prediction Engine* maintains the inter-change-time and shrinking probability distributions (§3.1.1). The *Match Maker* uses these distributions to compute the completion probability as follows:

$$
\begin{aligned}
P_c(x,e,d) &= 1 - P\left[\begin{matrix}\text{shrink occurs during}\\ \text{task lifetime}\end{matrix}\right] \\
&= 1 - P\left[\begin{matrix}\text{resource change}\\ \text{is shrink}\end{matrix} \wedge \begin{matrix}\text{change occurs}\\ \text{during task lifetime}\end{matrix}\right] \\
&= 1 - P_s(x) \cdot (1 - P(X > e+d|X > e)) \qquad (1)
\end{aligned}
$$

Table 1 shows the definition for each symbol. Recall that tasks are annotated with their estimated runtimes (§3).

Delayed scheduling. In certain situations, waiting for a more stable node (i.e., with better completion probability) might be a better choice than picking from the currently available resources. For example, rather than launching a task on a node with low completion probability, it might be better to wait for old tasks to complete on a node that is relatively stable but may be fully occupied with previous tasks. This implies a non–work–conserving schedule where the scheduling of tasks may be delayed even when resources are available.

It is challenging to decide whether to wait or run with current the best resource without future knowledge. A strawman approach is to launch tasks on an HVM only when it provides a threshold completion probability. However, this may cause the scheduler to delay indefinitely. To tackle this, we formulate the cost and benefit of waiting by estimating the expected completion time obtained from various decision choices. We derive expected completion time as follows:

$$
\begin{aligned}
&E_{wc}(x,a,e,d)\\
&= E\begin{bmatrix}\text{time spent waiting}\\ \text{for node to free up}\end{bmatrix} + E\begin{bmatrix}\text{time for completion}\\ \text{and preemptions}\end{bmatrix}\\
&= a + p\cdot d\\
&\qquad + (1-p)\cdot\left(E\begin{bmatrix}\text{time wasted}\\ \text{to preemption}\end{bmatrix} + E\begin{bmatrix}\text{time after}\\ \text{restart}\end{bmatrix}\right)\\
&= a + p\cdot d + (1-p)\cdot(w + E_{wc}(g,0,0,d)) \qquad (2)\\
&e' = e + a\\
&p = P_c(x,e',d)\\
&w = E\left(X - e' \mid (X > e') \wedge (X < e' + d)\right)
\end{aligned}
$$

This equation incorporates when will the node free up resources (a), the duration of the task (d) if it completes succesfully (with probability p), time potentially wasted (w) in case the task fails (with probability $1-p$), and the cost of rescheduling the task ($E_{wc}(g,0,0,d)$). Note that this is the worst-case expected completion time since we reschedule a task on the worst-case node that has just started (has not remained stable at all, i.e., $e = 0$). In reality, after preemption, the task may be started on a stable node. Since we know the estimated duration of tasks, their resource requirements, and how long they have been running, we can deterministically compute when the node will have enough resources to run a given task (a). The time wasted (w) is the expected time between when the task starts and when the node shrinks, given that the node does shrink during the lifetime of the task.

Workflow. The *Match Maker* uses the above formulation (Equation 2) to schedule the task on the node which gives the best expected completion time. If this node does not have resources at the moment, then it waits and reconsiders the decision at the next scheduling iteration. This automatically includes both preferring nodes with higher completion probability and considering to wait for nodes to free up. For completeness, we list the pseudo-code for the *Match Maker* in Algorithm 1 in the Appendix.

Symbol	Interpretation
P_c	Probability of completion
X	Inter-change time distribution
$x \in \{g,s\}$	Denotes whether last change event was growth or shrink. Default is growth for a newly started node
e	Time elapsed since last resource change event
d	Remaining duration of the task under consideration
P_s	Probability that the next resource change event is a shrink
E_{wc}	Expected worst case completion time of the task
a	Time between now and when the node will have enough resources to run the task under consideration. If the node currently has enough resources, then $a = 0$
w	Expected time wasted due to a shrink occurring before task completion

Table 1: Symbol definitions.

3.2 SLACKSCHED Acquirer

Challenges. Ideally, we want to maintain a set of HVMs whose stability matches the runtime of the tasks. However, in addition to task runtime information, this requires full knowledge on: (1) how unallocated resources are distributed across servers in a data center, and (2) how stable these resources are at any point in time. As a user (or cluster orchestrator), this information is not available or possible to model. Even as the provider, these metrics are only instantaneously available while HVM patterns may change over time (Figure 3).

Approach. We use a simple "exploration-exploitation" strategy to navigate the set of potential HVMs that the provider can offer to maintain a stable pool of HVMs. The Acquirer starts with a random mix of HVMs and periodically identifies the *worst* HVMs and decommissions them, gradually converging to a more stable pool of HVMs. The Acquirer defines "worst" as the most recently changed HVMs. This simple strategy works when there the HVM pool is unstable relative to the HVMs that can be returned by the provider (4.3).

In our implementation, the Acquirer runs hourly and deallocates 10% of allocated HVMs. Concurrently, it requests an equal number of new HVMs from the provider to maintain the same amount of cluster resources.[5] To avoid getting a HVM on the same server as the one just deallocated, the Acquirer first requests new HVMs and then deallocates the unwanted ones. To avoid task preemptions, it *gracefully decommissions* HVMs before deallocating them. The scheduler stops sending new tasks to the decommissioning HVMs and once all running tasks complete, the Acquirer deallocates them.

To maintain a target set of resources in the midst of load variations, the Acquirer works with a scaling policy that maintains the cluster utilization between a lower and upper bound (i.e., 60 to 80%). It requests or deallocates VMs whenever the utilization falls outside of this target range. When scaling-in, we decommission and deallocate the worst HVM first. The

[5] Currently, we consider mixes with only HVMs. One can consider a mix of both HVMs and regular VMs.

Acquirer can work with different scaling policies [8, 10], and the scaling can be decoupled from the intentional deallocations/allocations. In our implementation, scaling is triggered every hour coupled with the Acquirer. Scaling also allows maintaining resources as HVMs grow/shrink.[6] Algorithm 2 in the Appendix shows the pseudo-code for the Acquirer.

We studied different choices for the Acquirer's trigger period. We found that trigger periods in half to two hours range yield similar performance as 1 hour. Longer periods can lead to a stale cluster (with unstable HVMs) and shorter periods can operate on limited historical information (especially for HVMs allocated in the previous trigger) leading to erroneously deallocating potentially stable HVMs. In general, the trigger period should be chosen based on the range of HVM inter-change times and task runtimes.

3.3 Implementation

We implement SLACKSCHED within YARN [5] version 3.3.0. This includes the changes by [3] that make YARN aware of HVM resource variations. We added an Acquirer module that interfaces with the provider's VM request API and monitors statistics about allocated nodes. We updated Job Managers to convey task runtime estimates as part of Resource Requests and updated the matching logic to use our design (§3.1). These changes (highlighted in Figure 6) preserve compatibility with other scheduling features, e.g., reservations [27], delay scheduling [86], affinity [4], etc. Specifically for multi-dimensional packing [38], the affinity between tasks and nodes can be defined as a combination of resource match and stability match. We believe that our implementation can be easily ported to other cluster managers.

4 Evaluation

We evaluate SLACKSCHED to answer the following questions:

1. How much benefit do we gain solely from scheduling under different HVM resource profiles? (§4.2)
2. How robust is our design to various workload characteristics, operating conditions and estimation errors? (§4.2.1)
3. How much do we benefit from resource acquisition? (§4.3)
4. How does SLACKSCHED handle time varying arrival rate and compare with other VM types? (§4.4)
5. How does SLACKSCHED compare to prior techniques for addressing VM evictions? (Appendix B.1 and §5).
6. What happens under future and more extreme sources of resource variability (e.g., as a result of using renewable energy)? (§4.5)

4.1 Methodology

Setup. Since HVMs are highly variable and our method is probabilistic, to reach any conclusive results, we needed to run each experiment at a large scale: ≈ 50 jobs, where each job lasts hours and requires 100s of regular VMs. However, running such long and large-scale experiments was impractical on a real testbed. Thus, we evaluated our system using Hadoop's discrete time simulator [1, 6]. This simulator accurately mimics real-world setups, with only $\approx 1.3\%$ error on completion times and $\approx 1.5\%$ error on resource utilization [24]. The simulator runs actual YARN Resource Manager [5] code and only simulates the Node Managers, Job Managers, and clock and communication layers. Using a simulator also allows us to maintain the exact same trace of HVM availability, ensuring fair A/B testing across experiments.

Resource traces. We use two sets of production resource traces from Microsoft Azure: (1) *HVM traces:* time series of HVM resource availability that includes the time and sizes of growth/shrink events for each HVM; (2) *On-demand VM traces:* VM arrival times, lifetimes, and placement decisions made by the provider's production allocator. Our dataset includes traces from 8 clusters ($700-2000$ servers each) across 5 regions from two time periods (March 2019 and August 2021). For our experiments, we randomly select 64 HVMs for each cluster. This translates to 160–480 regular VMs in terms of resources as each HVM gets 2.5–7.5$\times$ more resources than their minimum size (§2.1).

Extensions to the simulator. To ensure different scheduling schemes witness the same HVM changes, we extend the simulator to replay HVM traces. When an HVM shrinks, containers are killed (in increasing order of their start time) until the available resources on the node exceed or equal the used resources on the node. When an HVM grows, new containers may be allocated to the node. These are the default Resource Manager behaviors. In other words, a task does not benefit from cores beyond what it asked for and is killed as soon as it gets fewer than its requested cores.

In addition, for resource acquisition experiments (that continually request new HVMs), we want every experiment to receive the same HVMs when requesting the same configuration at the same time. To ensure this, we replay the on-demand VM allocation traces to reconstruct the state of the unallocated resources. Alongside, we add a simulated HVM allocator to place HVMs on servers with unallocated resources. We study different allocation policies including random, load-balancing, and packing. Note that we use the simulated HVM allocator only for the resource acquisition experiments (§4.3, §4.4).

Workloads. Our traces are based on a collection of two workload categories running in a production environment: Seismic and Genomics [16, 22, 34, 58, 59]. We log their execution to build distributions of task runtimes, number of tasks, and resource requirements. We sample from these distributions to build traces of jobs/tasks. We model job arrival as a Poisson process (similar to [38–40, 56]) and study a variety of mean inter-arrival times.

Schemes. For scheduling, we compare SLACKSCHED (4) against the three schemes (1-3 below):

1. *CapacityScheduler* (or *CapS*): The default scheduler in

[6] Due to growth/shrinks/evictions, instantaneous cluster utilization can fall outside our target range in the time between two scaling triggers.

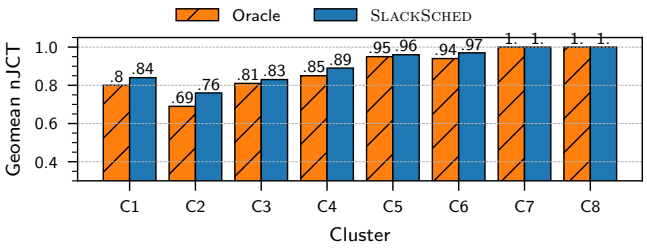


Figure 9: Normalized job completion time (nJCT) relative to *CapacityScheduler* across HVM traces from different clusters. SLACKSCHED improves JCT across different clusters.

YARN. Considers resource requirements (cores and memory) of tasks to select nodes [7].

2. *Oracle*: Uses future knowledge to schedule a task on an HVM only if it is guaranteed to complete before any future resource shrinks, and skips allocation otherwise. Skipping implicitly delegates scheduling to other nodes or later times. This provides an upper-bound for SLACKSCHED.
3. SLACKSCHEDNODELAY (or SSNODELAY): Only uses current resources to make decisions and does not wait for nodes to free up resources. It places tasks on nodes with the highest completion probability.
4. SLACKSCHED (or SLACKS): Uses expected completion time to match tasks to nodes and can wait for nodes to free up resources (§3).

Oracle is a good proxy for job completion time on a regular VM cluster that has same total resources as the HVM cluster. This is because *Oracle* does not cause any task preemptions and the clusters are sized to ensure little to no task queuing.

Metrics. We study job completion time (JCT), normalized job completion time (nJCT), and dollar cost, where,

$$\text{nJCT(Job)} = \frac{\text{JCT(Job) with scheme}}{\text{JCT(Job) with baseline}}$$

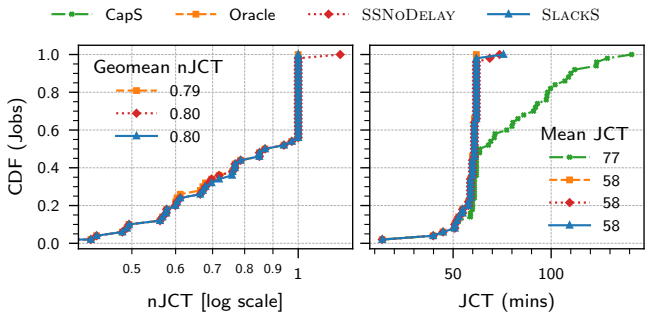
JCT distributions show absolute and tail completion times, while the normalization allows us to study the impact on jobs with different runtimes. A smaller nJCT i.e., nJCT $\in (0,1)$ is better, while nJCT $\in (1,\infty)$ is worse. nJCT of 0.7 translates to $(1-0.7)*100 = 30\%$ reduction in JCT (cf. [40]), and $1/0.7 = 1.43\times$ factor of improvement or speedup (cf. [39]). For computing nJCT, we use baseline as *CapacityScheduler* unless mentioned otherwise.

For aggregating across jobs, we study (a) mean reduction in JCT $(1 - \textit{GeometricMean}(\textit{nJCT}))$, (b) reduction in mean JCT $(1 - \frac{\text{mean JCT with scheme}}{\text{mean JCT with baseline}})$, (c) reduction in tail (90th percentile or p90) JCT.

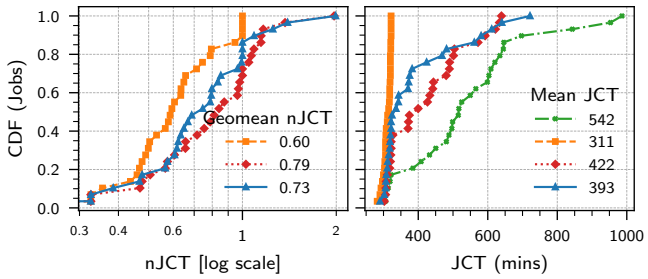
We delegate a subset of these metrics to Appendix B.5.

4.2 Scheduler Evaluation

We evaluate SLACKSCHED on the production clusters from our dataset under the same workload trace. Figure 9 shows the geomean nJCT relative to *CapacityScheduler*. Mean reduction from SLACKSCHED ranges from 0 to 24% (i.e., geomean nJCT from 1 to 0.76). Improvements from SLACKSCHED are close (within 13%) to the *Oracle*.



(a) High-activity cluster (C2, frequent resource changes) with shorter tasks.



(b) Low-activity cluster (C8, infrequent resource changes) with longer tasks.

Figure 10: SLACKSCHED's mean reduction in JCT is 20–27%. Waiting for more stable nodes provides a further 0–7.5% reduction. SLACKSCHED reduces mean JCT by 25–27% and p90 JCT by 20–44%. We use Seismic A workload (§2.2) for this experiment which has uniformly distributed task runtimes.

The frequency of resource changes relative to the task durations govern how much overhead HVMs impose and in turn govern how much benefit we can get from SLACKSCHED. We observe that benefits are greater in clusters that witness more activity from regular VMs (i.e., more HVM resource changes). Specifically, for the last two clusters (C7 and C8) which have the least amount of activity from regular VMs (§2.1), HVMs cause little slowdown for the particular workload.

We zoom into the JCT and nJCT distributions for a high-activity cluster (C2). We also study a low-activity cluster (C8) with longer tasks. These represent cases when HVMs impose non-trivial overhead. Figure 10 shows that SLACKSCHED's mean reduction in JCT is 20–27%. 40% of the jobs see more than 30% reduction in their completion times. 0–15% jobs have nJCT > 1, implying their JCT increases. Since our method is probabilistic, SLACKSCHED can make poor decisions for some individual tasks compared to a random matching decision (taken by *CapacityScheduler*). However, *on average*, SLACKSCHED's matching decisions are better than random. SLACKSCHED reduces mean JCT by 25–27% and p90 JCT by 20–44%. In §4.2.1, we vary various aspects of the workload.

Impact of "waiting" for stable nodes. Figure 10 also quantifies the impact of waiting for stable, but currently busy nodes (§3.1.2). As shown, SLACKSCHED provides an additional

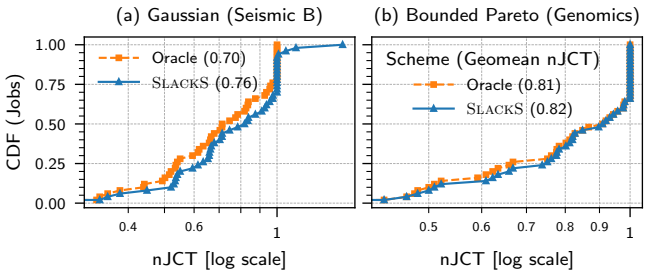


Figure 11: nJCT for workloads with different task runtime distributions. SLACKSCHED consistently improves JCT.

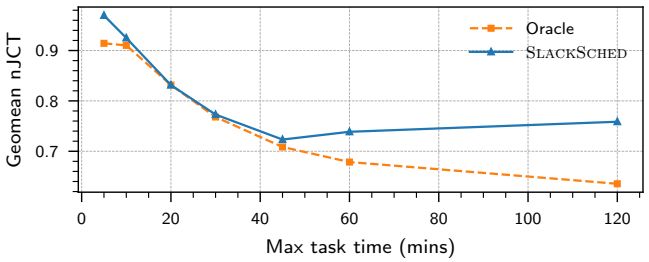


Figure 12: nJCT with varying task runtimes. SLACKSCHED is effective for a wide range of task runtimes.

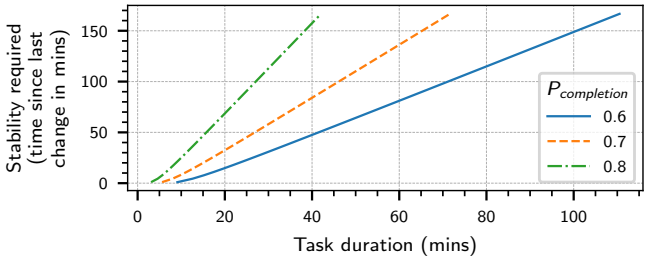


Figure 13: HVM stability required for different task durations at different confidence levels. To obtain 70% confidence in completion without preemption for a task with duration d, we roughly need a node to have been stable for $2.6 \times d$ time. We compute "stability required" using inverse CDF of inter-change-time distribution.

0–6% mean reduction in JCT compared to when this feature is disabled (SLACKSCHEDNODELAY). Specifically, for the high-activity cluster (C2), SLACKSCHEDNODELAY is already close to *Oracle* and there is little scope of further improvement from waiting.

Resource waste reduction. We compare the fraction of resource waste (i.e., work wasted due to preemptions divided by total work done). We find that *CapacityScheduler*, SLACKSCHED, and *Oracle* waste 20–40%, 3–20%, and 0% work respectively, i.e., SLACKSCHED reduces resource waste by $\approx 20\%$ compared to *CapacityScheduler*. This saving comes from the better task-to-HVM matching that avoids preemptions and reduces wasted work or resources.

4.2.1 Improvement Conditions and Robustness

SLACKSCHED's gains depend on the range and distribution of task runtimes relative to the inter-change times of HVMs. We find that SLACKSCHED provides benefits in JCT when:

- There is spatial variation in stability of HVMs and spatial variation in runtime of concurrently running tasks. Otherwise, different mappings from tasks to nodes are equivalent.
- The range of HVM inter-change times is similar to the range of task runtimes. If tasks are too short, there is not much room for improvement as there are few preemptions. If tasks are too long, preemptions cannot be avoided.

Both conditions hold for a large set of workloads and HVM resource profiles (regions/clusters) (§2). We now describe our robustness experiments that led to these findings.

Task runtime distribution. We change the task runtime distribution in accordance with different workloads (§2.2) under the same HVM resource variations (high activity cluster, C2). This is shown in Figures 10a (Uniform), 11a (Gaussian), and Figure 11b (Bounded Pareto). Workloads with higher variance (Uniform and Pareto) see more improvements from SLACKSCHED since they benefit more from matching of tasks to resource variability.

Task duration. We analyzed the impact of varying the range of task runtimes. We use uniformly distributed task runtimes between 1 and X minutes (max task time) and vary X. Figure 12 shows that the improvements in both SLACKSCHED and *Oracle* diminish with short tasks (< 20 min) as they are rarely preempted (no scope for improvement).

SLACKSCHED's improvements also diminish when tasks become too long as it becomes harder to find stable nodes. This is because SLACKSCHED relies on past history to predict the future, e.g., to schedule a task of duration d with 70% confidence, it needs a node which has remained stable for roughly $2.6d$ time (Figure 13). On the other hand, *Oracle* can identify future stable nodes even if they have been unstable in the past, allowing it to find matching nodes for long tasks.

Cluster load. We increase the load by reducing the mean inter-arrival time of jobs while keeping the same set of HVMs, task runtime distributions, and task resource requirements. Figure 14 shows that when load becomes very high ($\approx 100\%$ at 3 mins mean inter-arrival, as shown in the first data point), the benefits of SLACKSCHED diminish, since the relative number of stable nodes decrease and SLACKSCHED has a harder time finding nodes for longer tasks (same reason as shown before with Figure 13). *Oracle* still provides improvement since it has full future knowledge and can still find stable nodes.

Task runtime misestimates. SLACKSCHED uses estimates of task runtimes to compute completion probabilities and expected completion times. We evaluate SLACKSCHED's sensitivity to misestimates by injecting errors into the runtimes reported while still running tasks with their original runtimes. To inject errors, for each task, we deviate its duration estimate by a number sampled uniformly between 0 and some max percentage error. Negative error implies underestimates and positive error implies overestimates. Figure 15 shows there is little impact of misestimates on SLACKSCHED. Completion times inflate when runtimes are underestimated. This is be-

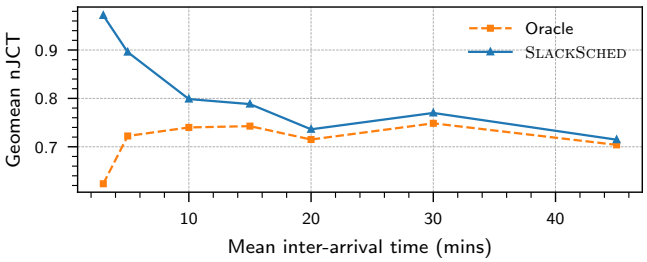


Figure 14: nJCT with varying job inter-arrival rates. SLACKSCHED improves JCT at different cluster loads.

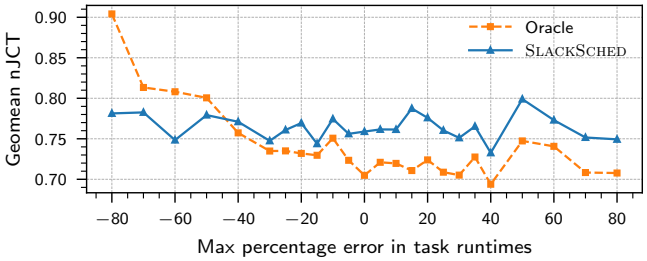


Figure 15: nJCT with varing errors in task runtimes estimates. SLACKSCHED improves JCT even under modest errors.

cause runtime underestimation results in overestimating the completion probabilities and matching tasks to nodes which may not remain stable through the lifetime of the task. We see more impact on *Oracle*, as it makes many close calls which are rendered incorrect due to inaccurate duration estimates.

4.3 Acquirer Evaluation

We study benefits from SLACKSCHED-Acquirer considering settings where there is room for improvement in job completion time between *Oracle* and SLACKSCHED-Scheduler.

Methodology. Different experiments differ in their scheduling and acquisition logic. In all cases, HVMs are continuously (minimum size chosen uniformly randomly) requested to maintain ≈ 64 HVMs worth of resources. For these experiments, we do not use the scaling policy (§3.2) and consider a constant job arrival rate. Our baseline uses the *CapacityScheduler* with no intentional HVM deallocations.

These experiments require a simulated HVM allocator and we test our system with three policies to decide which server to place an HVM on: (1) Balancing, picks the server with the most amount of unallocated resources, (2) Packing, picks the server with the least amount of unallocated resources, (3) Random, picks a random server. All policies only consider servers which have enough resources to support the minimum size of the HVM at the time of allocation.

Results. Figure 16 shows qualitatively similar improvements across different HVM allocation policies solely from the SLACKSCHED-Scheduler (16–24% mean reduction). We also obtain 8–23% mean reduction solely from the SLACKSCHED-Acquirer. The Scheduler and the Acquirer complement each other to provide a mean reduction of 27–32%.

Improvement conditions. In addition to the improvement

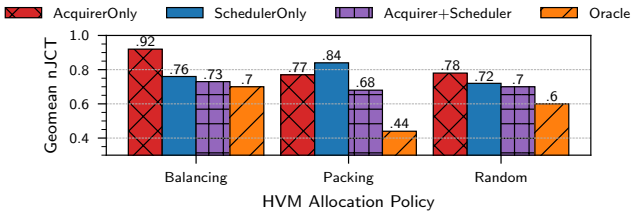


Figure 16: nJCT relative to baseline. SLACKSCHED's scheduling and resource acquisition complement each other to reduce JCT.

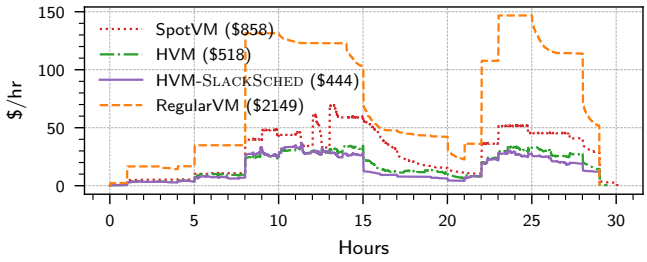


Figure 17: Cluster size over time in terms of cost ($) per hour. Acquisition logic adapts to time-varying job arrival rate. HVMs are cheaper than Spot and Regular VMs due to the discounted harvested cores. Legend lists the total cost ($) in parentheses.

conditions in §4.2.1, resource acquisition is useful when there is room for improving the stability of HVMs in the cluster. This happens if (1) there are not enough stable HVMs in the cluster and (2) there are better HVMs that can be returned by the allocation policy. For instance, when most HVMs are allocated on *buffer servers*, there is little scope to improve the HVM mix. Providers typically reserve buffer servers to satisfy sudden demand for regular VMs. In the absence of sudden demand changes for regular VMs, HVMs on buffer servers witness few resource variations and cause little to no job slowdown. In our implementation, we disallow the Balancing policy from placing HVMs on buffer servers (≈ 5% of all servers). This is because, in practice, multiple users will request HVMs from the cloud provider and a single user will only get a small portion of HVMs allocated on buffer servers. In general, we expect individual users to see HVMs that behave closer to those allocated by the Random policy.

4.4 Scaling and Cost Comparison

We study how SLACKSCHED adapts to a workload that varies over time. For such a workload, we compare the performance (JCT) and cost of maintaining homogeneous pools of HVMs, Spot VMs, and regular (on-demand) VMs.

Methodology. To generate the time-varying workload, we vary the mean job arrival rate between 1× and 4× the base rate every 6 hours. For the environment, the Acquirer maintains a homogeneous cluster of Spot VMs, HVMs or regular VMs in separate runs. Runs without SLACKSCHED (i.e., SpotVM, HVM, RegularVM in Figures 17 and 18) use *CapacityScheduler*. For these cases, the Acquirer does not intentionally deallocate servers and only uses the scaling policy to

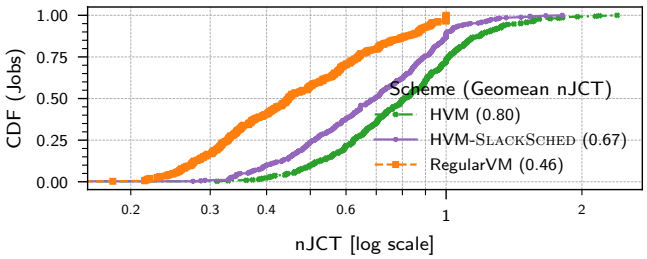


Figure 18: nJCT relative to Spot VM. SLACKSCHED improves JCT over out-of-the-box HVMs under a time-varying job arrival rate. Vanilla HVMs yield better JCT than Spot VMs.

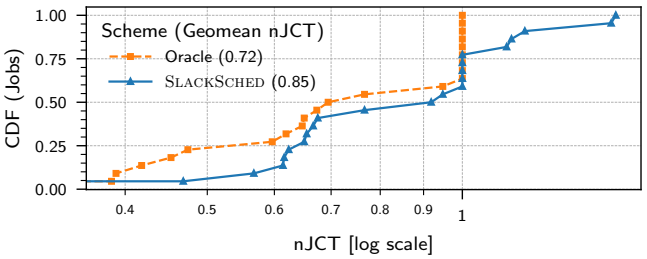


Figure 19: SLACKSCHED improves completion times even under power variations due to renewable energy sources.

tune the cluster size in response to changing job arrival rate, randomly choosing which VMs to deallocate.

We use the Random allocation policy (from §4.3) to determine the placement of HVMs and Spot VMs. Spot VMs are fully evicted if resources are needed for competing regular VMs allocations (i.e., they do not shrink like HVMs). We obtain the cost of regular and Spot VMs from [74]. For HVMs, we follow the same pricing scheme as [3, 87] and charge their minimum size at the same price as Spot VMs of the same size and charge additional harvested cores at a 50% discount.[7]

Scaling results. Figure 17 shows that the SLACKSCHED-Acquirer scales the cluster in and out in response to the changing job arrival rate for all VM types. It maintains a relatively constant core utilization over time (not shown). Across schemes, the Acquirer maintains similar core utilization and amount of resources. Different schemes have different number of preemptions and resource waste.

Cost results. Figure 17 also shows that HVMs are 40% (1.67×) and 75% (4×) cheaper than Spot and regular VMs respectively. The ≈ 14.2% difference in cost with and without SLACKSCHED ($444 vs. $518) mainly comes from (1) less resource waste, and (2) intentional deallocations that move utilization closer to the upper bound (80% threshold §3.2).

JCT results. Figure 18 shows that HVMs without SLACKSCHED already provide mean reduction of 20% compared to Spot VMs as HVMs shrink instead of getting evicted. The Scheduler+Acquirer in SLACKSCHED provide a mean reduction of 33% over Spot VMs even under a time-varying arrival rate and without incurring extra cost.

4.5 Case Study: Renewable Energy Sources

To further test the generality of our approach, we consider a future source of resource variability: renewable energy sources (e.g., wind and solar power). Power generated from renewable energy sources typically fluctuates over time as these sources are driven by weather conditions that are in turn variable over time. We study the case of HVMs that vary in resources due to a changing power supply in addition to on-demand VM arrivals/departures. In these scenarios, leveraging an unreliable resource supply is a necessity rather than an optimization.

Methodology. We use the same simulation environment and workloads as before. For generating HVM traces under renewable energy variations, we use wind power generation traces from the ELIA dataset [66]. We scale the power trace such that the cluster is fully powered at the max power in the trace. For simplicity, we also assume that cores powered on a server are proportional to the power supplied to the server. We replay the on-demand VM allocation traces and the power generation traces. Whenever power drops, we reduce the power supply to servers that have unallocated resources, effectively reducing the size of the HVMs. When no such servers exist, we evict on-demand VMs (assuming they are migrated out). When power rises, we increase the power supply to servers that have a lower supply than the max power, effectively increasing the size of the HVMs. Additionally, we relaunch previously-evicted on-demand VMs to maintain the cluster load (assuming they are migrated in).

Results. We observe that the inter-change time and shrink distributions are still relatively stationary when taken for a day. However there are more variations at smaller time scales (e.g. when taken for a window of 6 hours). Since SLACKSCHED maintains a large enough snapshot (i.e., 1 day) of the HVM distributions, it still prevents preemptions and improves completion times. Figure 19 shows that SLACKSCHED provides a mean reduction in JCT of 15%. Similar to Figure 10, ≈ 25% jobs degrade (nJCT > 1) relative to their execution with *CapacityScheduler*.

5 Related Work

Related work not covered in §3.1 can be classified into:

Checkpointing, migration, and replication. Work like [71, 80] use a combination of these techniques to mitigate the impact of preemptions caused by Spot VM evictions. These are prohibitive for long uninterruptible workloads. In Appendix B.1, we empirically show that SLACKSCHED outperforms these techniques for our target workloads by profiling their checkpointing overheads.

[7] We are only interested in price variations across space but not across time. Temporal variations in price would affect all HVMs (at least within the same region) symmetrically and would not significantly affect our acquisition decisions. Thus, we pick costs from [74] at a single point in time. In reality, Spot VM prices, and in turn HVM prices, are volatile over time and can range between 48% and 88% of the regular VM price.

Multiple markets. [20, 43, 70, 71, 80, 84] mitigate the impact of VM evictions by picking Spot VMs from different VM types/sizes and regions (i.e., markets). Different markets can have different prices and eviction rates. These efforts estimate Spot VM eviction likelihoods based on spot prices and bids. However, pricing-based prediction techniques may not necessarily predict resource changes (e.g., HVMs shrinking) which cause workload preemptions. Further, SLACKSCHED yields improvements even for a cluster of HVMs taken from a single market (i.e., the same minimum size and region) as shown in Figure 10b. For such setting, we expect multi-market techniques will perform similar to the *CapacityScheduler* as they will not distinguish between VMs taken from a single market.

[44, 70, 80, 84] use an ensemble of on-demand and Spot VMs. SLACKSCHED provides improvements without requiring on-demand VMs that cost 2–10× more than HVMs.

Bidding, pricing, and admission control. [57, 85] use bidding strategies to control VM eviction likelihoods. These may not control HVM resource variations which can preempt workloads. Further, pricing-based techniques only work when evictions and pricing are related. These methods do not generalize to flat pricing models [51, 74]. Many of these efforts are also scoped to different workloads (e.g., machine learning training [44], database queries [84]) and do not generalize to long uninterruptible workloads.

In a setting where jobs have different levels of importance, admission control [65] techniques may be useful. This is orthogonal to SLACKSCHED.

Scheduling and task placement. Most prior work assumes fixed resources over time [33, 37–40, 52, 64, 79, 86]. They leverage multi-dimensional packing, locality, fairness, and workload properties (e.g., dependencies). These ideas are orthogonal to our work and SLACKSCHED can leverage them to further improve scheduling objectives including efficiency, fairness, and completion times. However, HVMs incur large resource variations which are not addressed by this prior work.

Other proposals [3, 56, 87] adapt cluster scheduling frameworks to address resource variability. However, they are scoped to specific workloads that are less challenging than long-running uninterruptible workloads. [56] only considers elastic query processing workloads, [87] considers serverless functions with short tasks, and [3] just reacts to resource variations rather than avoiding preemptions. The closest to our work is SciSpot [51] that schedules tasks using a time-to-eviction distribution for Spot VMs. It does not consider waiting for VMs to free up resources. Waiting (delayed scheduling) provides better performance (§4.2). SciSpot also does not provide any empirical evaluation and only estimates potential for improvement using theoretical analysis. It does not consider resource acquisition and only works with bathtub-shaped time-to-eviction distributions.

Addressing underutilization. Prior work has also looked at underutilization in cloud environments [78, 88]. These try to co-locate latency critical services and batch workloads to reduce resource fragmentation and improve cluster utilization. These techniques often also leverage the fact that jobs do not use their peak resources all the time and thus oversubscribe resources. However, such oversubscription is typically only done for first party workloads and not customer facing services [3, 42, 72, 88]. Hence, providers still deal with unallocated resources [3, 88].

Serverless computing or FaaS (Function-as-a-Service) give up on the VM abstraction and allow providers more flexibility to dynamically spread computation and reduce resource fragmentation. HVMs try to preserve the VM abstraction allowing use of harvested capacity for workloads that are not suited for serverless computing, e.g., workloads that maintain state, need the abstraction of a machine, shared libraries, or operating system, or require significant software engineering effort for porting to FaaS abstractions.

While HVMs expose unallocated resources, SmartHarvest [82] also opportunistically exposes allocated but unused resources. It uses machine-learning techniques to decide when and how many resources can be harvested without harm. SmartHarvest resource variations are fine-grained (millisecond level) and would require additional scheduling solutions.

6 Conclusion

Cloud providers have started using new mechanisms like Spot VMs and Harvest VMs (HVMs) to monetize their unallocated resources. After a characterization of HVMs and workloads, we identified that prior work falls short at running long, uninterruptible workloads in such variable environments. To enable these workloads, we propose SLACKSCHED, which leverages distribution-based predictions to maintain a stable pool of HVMs and intelligently match tasks to their ideal resources. SLACKSCHED successfully enables running workloads on HVMs as if they were run on regular VMs.

We experimentally demonstrated that our proposal reduces resource waste by 20% and improves mean and tail (90th percentile) completion time by 27% and 44% respectively, at 75% lower cost than regular VMs. We also show that our system generalizes to cases where resource variations are caused by a variable power supply. We plan to contribute our code to the Apache YARN project [5].

Acknowledgments

We would like to thank anonymous reviewers, our shepherd John Wilkes, and Srikanth Kandula for feedback that helped us improve this work. We would also like to thank Peeyush Kumar for useful discussions, Philipp Witte and Roberto Lleras for help with workloads, and Shivkumar Kalyanaraman and Srinivasan Iyengar for help with the renewable energy case study.

References

[1] *[YARN-1187] Add discrete event-based simulation to yarn scheduler simulator - ASF JIRA*. [Online; accessed 17. Mar. 2021]. 2021. URL: https://issues.apache.org/jira/browse/YARN-1187.

[2] *Amazon EC2 Spot – Save up-to 90% on On-Demand Prices*. [Online; accessed 12. Sep. 2021]. URL: https://aws.amazon.com/ec2/spot/?cards.sort-by=item.additionalFields.startDateTime&cards.sort-order=asc.

[3] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *OSDI*. 2020.

[4] *Apache Hadoop 3.3.1 – YARN Node Labels*. [Online; accessed 13. Sep. 2021]. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/NodeLabel.html.

[5] Apache Software Foundation. *Apache Hadoop YARN*.

[6] Apache Software Foundation. *Hadoop*. Version 0.20.2. Feb. 19, 2010.

[7] Apache Software Foundation. *Hadoop: Capacity Scheduler*.

[8] *AWS Auto Scaling*. [Online; accessed 17. Apr. 2022]. Apr. 2022. URL: https://aws.amazon.com/autoscaling.

[9] *AWS Batch — Easy and Efficient Batch Computing Capabilities - AWS*. [Online; accessed 13. Sep. 2021]. URL: https://aws.amazon.com/batch.

[10] *Azure Autoscale | Microsoft Azure*. [Online; accessed 17. Apr. 2022]. Apr. 2022. URL: https://azure.microsoft.com/en-us/features/autoscale.

[11] *Azure Kubernetes Service (AKS) | Microsoft Azure*. [Online; accessed 13. Sep. 2021]. URL: https://azure.microsoft.com/en-us/services/kubernetes-service/#overview.

[12] *Azure Sustainability—Sustainable Technologies | Microsoft Azure*. [Online; accessed 15. Sep. 2021]. URL: https://azure.microsoft.com/en-us/global-infrastructure/sustainability/#overview.

[13] Noman Bashir, Tian Guo, Mohammad Hajiesmaili, David Irwin, Prashant Shenoy, Ramesh Sitaraman, Abel Souza, and Adam Wierman. "Enabling Sustainable Clouds: The Case for Virtualizing the Energy System". In: *SoCC*. 2021.

[14] *Batch - Compute job scheduling service | Microsoft Azure*. [Online; accessed 13. Sep. 2021]. URL: https://azure.microsoft.com/en-us/services/batch/#overview.

[15] Edip Baysal, Dan D. Kosloff, and John W. C. Sherwood. "Reverse Time Migration". In: *GEOPHYSICS* 48.11 (1983), pp. 1514–1524. eprint: https://doi.org/10.1190/1.1441434.

[16] Broad Institute. *cromwell*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/broadinstitute/cromwell.

[17] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. "Syntactic Clustering of the Web". In: *Computer networks and ISDN systems* 29.8-13 (1997), pp. 1157–1166.

[18] *Burstable performance instances*. [Online; accessed 12. Sep. 2021]. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[19] A.W. Camargo, J. Ribeiro, N. Okita, C. Benedicto, T.A. Coimbra, J.H. Faccipieri, and M. Tygel. "Fault-Tolerant Wave Propagation Assisted By Independent Checkpointing Strategy". In: 2020.1 (2020), pp. 1–5.

[20] Marcus Carvalho, Walfredo Cirne, Franciso Brasileiro, and John Wilkes. "Long-term SLOs for reclaimed cloud computing resources". In: *SoCC*. 2014.

[21] *Chameleon - Texas Advanced Computing Center*. [Online; accessed 14. Sep. 2021]. URL: https://www.tacc.utexas.edu/systems/chameleon.

[22] ChevronETC. *Examples*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/ChevronETC/Examples.

[23] Andrew A. Chien, Richard Wolski, and Fan Yang. "The Zero-Carbon Cloud: High-Value, Dispatchable Demand for Renewable Power Generators". In: *The Electricity Journal* 28.8 (2015), pp. 110–118.

[24] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. "Unearthing inter-job dependencies for better cluster scheduling". In: *OSDI*. 2020.

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *SOSP*. 2017.

[26] *CRIU*. [Online; accessed 23. Mar. 2022]. Feb. 2022. URL: https://criu.org/Main_Page.

[27] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. "Reservation-Based Scheduling: If You're Late Don't Blame Us!" In: *SoCC*. 2014.

[28] J.T. Daly. "A higher order estimate of the optimum checkpoint interval for restart dumps". In: *Future Generation Computer Systems* 22.3 (2006), pp. 303–312.

[29] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Dec. 2004.

[30] Yanlei Diao, Abhishek Roy, and Toby Bloom. "Building Highly-Optimized, Low-Latency Pipelines for Genomic Data Analysis." In: *CIDR*. 2015.

[31] *docker checkpoint*. [Online; accessed 23. Mar. 2022]. Mar. 2022. URL: https://docs.docker.com/engine/reference/commandline/checkpoint.

[32] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. "Memory-Harvesting VMs in Cloud Platforms". In: *ASPLOS*. 2022.

[33] M. R. Garey, D. S. Johnson, and Ravi Sethi. "The Complexity of Flowshop and Jobshop Scheduling". In: *Mathematics of Operations Research* 1.2 (May 1976), pp. 117–129.

[34] *GATK*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://gatk.broadinstitute.org/hc/en-us.

[35] *Genomic data processing reference architecture*. [Online; accessed 13. Sep. 2021]. URL: https://cloud.google.com/architecture/genomic-data-processing-reference-architecture.

[36] *Get started with Batch | Google Cloud*. [Online; accessed 13. Sep. 2022]. URL: https://cloud.google.com/batch/docs/get-started.

[37] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: *NSDI*. 2011.

[38] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. "Multi-resource Packing for Cluster Schedulers". In: *SIGCOMM*. 2014.

[39] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. "Altruistic Scheduling in Multi-Resource Clusters". In: *OSDI*. 2016.

[40] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. "GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters". In: *OSDI*. 2016.

[41] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, David Dion, Esaias E Greeff, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. "Protean: VM Allocation Service at Scale". In: *OSDI*. 2020.

[42] James Hamilton. *AWS Innovation at Scale*. Nov. 2014. URL: www.youtube.com/watch?v=JIQETrFC_SQ.

[43] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. "Tributary: spot-dancing for elastic services with latency SLOs". In: *USENIX ATC*. 2018.

[44] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. "Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets". In: *EuroSys*. 2017.

[45] *Helping the financial services industry transform with Google Cloud | Google Cloud Blog*. [Online; accessed 13. Sep. 2021]. URL: https://cloud.google.com/blog/topics/financial-services/helping-the-financial-services-industry-transform.

[46] *High Burst CPU Compute for Monte Carlo Simulations on AWS | Amazon Web Services*. [Online; accessed 13. Sep. 2021]. URL: https://aws.amazon.com/blogs/hpc/high-burst-cpu-compute-for-monte-carlo-simulations-on-aws.

[47] *High-Performance Computing for Financial Services | Microsoft Azure*. [Online; accessed 13. Sep. 2021]. URL: https://azure.microsoft.com/en-us/solutions/high-performance-computing/financial-services/#features.

[48] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *NSDI*. 2011.

[49] *Introducing B-Series, our new burstable VM size*. [Online; accessed 12. Sep. 2021]. URL: https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size.

[50] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Jana Kulkarni, and Sriram Rao. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *OSDI*. 2016.

[51] Jcs Kadupitiya, Vikram Jadhao, and Prateek Sharma. "SciSpot: Scientific Computing On Temporally Constrained Cloud Preemptible VMs". In: *IEEE Transactions on Parallel and Distributed Systems* (2022), pp. 1–1.

[52] Ian A. Kash, Greg O'Shea, and Stavros Volos. "DC-DRF: Adaptive Multi-Resource Sharing at Public Cloud Scale". In: *SoCC*. 2018.

[53] *Kubernetes*. [Online; accessed 13. Sep. 2021]. URL: https://kubernetes.io/docs/reference.

[54] *Kubernetes - Google Kubernetes Engine (GKE) | Google Cloud*. [Online; accessed 13. Sep. 2022]. URL: https://cloud.google.com/kubernetes-engine.

[55] *Kubernetes on AWS | AWS*. [Online; accessed 13. Sep. 2021]. URL: https://aws.amazon.com/kubernetes.

[56] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. "Dynamic Query Re-Planning using QOOP". In: *OSDI*. 2018.

[57] Sharmistha Mandal, Sunirmal Khatua, and Rajib K. Das. "Bid Selection for Deadline Constrained Jobs over Spot VMs in Computational Cloud". In: *Distributed Computing and Internet Technology*. Cham, Switzerland: Springer, Nov. 2016, pp. 118–128.

[58] Microsoft. *AzureClusterlessHPC.jl*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/microsoft/AzureClusterlessHPC.jl.

[59] Microsoft. *CromwellOnAzure*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/microsoft/CromwellOnAzure.

[60] *National Energy Research Scientific Computing Center*. [Online; accessed 14. Sep. 2021]. URL: https://www.nersc.gov.

[61] Frank Austin Nothaft, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, Arun Ahuja, Jeff Hammerbacher, Michael Linderman, Michael J. Franklin, Anthony D. Joseph, and David A. Patterson. "Rethinking Data-Intensive Science Using Scalable Analytics Systems". In: *SIGMOD*. 2015.

[62] Keiron O'Shea and Ryan Nash. "An Introduction To Convolutional Neural Networks". In: *arXiv preprint arXiv:1511.08458* (2015).

[63] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. "3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty". In: *EuroSys*. 2018.

[64] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. "Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities". In: *ACM Transactions on Economics Computation* 3.1 (Mar. 2015).

[65] Florentina I. Popovici and John Wilkes. "Profitable Services in an Uncertain World". In: *SC*. 2005.

[66] *Power generation*. [Online; accessed 12. Sep. 2021]. URL: https://www.elia.be/en/grid-data/power-generation.

[67] *Request real-time and forecasted weather data using Azure Maps Weather services*. [Online; accessed 13. Sep. 2021]. URL: https://docs.microsoft.com/en-us/azure/azure-maps/how-to-request-weather-data.

[68] *Running Geospacial Workloads On AWS*. [Online; accessed 14. Sep. 2021]. URL: https://anz-resources.awscloud.com/aws-summit-sydney-2019-analyse/room5-day1-1745-runninggeospacialworkloadsonaws-v3-3.

[69] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. "Soap: One Clean Analysis of All Age-Based Scheduling Policies". In: *SIGMETRICS*. 2018.

[70] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. "SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market". In: *EuroSys*. 2015.

[71] Supreeth Shastri and David Irwin. "HotSpot: Automated Server Hopping in Cloud Spot Markets". In: *SoCC*. 2017.

[72] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *SIGCOMM*. 2015.

[73] *Slurm Workload Manager - Documentation*. [Online; accessed 13. Sep. 2021]. URL: https://slurm.schedmd.com/documentation.html.

[74] *Spot Virtual Machines – Spot Pricing and Features | Microsoft Azure*. [Online; accessed 12. Sep. 2021]. URL: https://azure.microsoft.com/en-us/services/virtual-machines/spot/#overview.

[75] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *INTERSPEECH*. 2012.

[76] *Sustainability | Google Cloud*. [Online; accessed 15. Sep. 2021]. URL: https://cloud.google.com/sustainability.

[77] *Sustainability in the Cloud*. [Online; accessed 15. Sep. 2021]. URL: https://sustainability.aboutamazon.com/environment/the-cloud?energyType=true.

[78] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: The next Generation". In: *EuroSys*. 2020.

[79] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. "TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters". In: *EuroSys*. 2016.

[80] Prateeksha Varshney and Yogesh Simmhan. "Autobot: Resilient and Cost-Effective Scheduling of a Bag of Tasks on Spot Vms". In: *TPDS* 30.7 (2019).

[81] Jean Virieux and Stéphane Operto. "An Overview of Full-Waveform Inversion in Exploration Geophysics". In: *GEOPHYSICS* 74.6 (2009), WCC1–WCC26. eprint: https://doi.org/10.1190/1.3238367.

[82] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. "SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud". In: *EuroSys*. 2021.

[83] Adam Wierman and Misja Nuyens. "Scheduling Despite Inexact Job-Size Information". In: *SIGMETRICS*. 2008.

[84] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. "Blending On-Demand and Spot Instances to Lower Costs for in-Memory Storage". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. San Francisco, CA, USA: IEEE Press, 2016, pp. 1–9.

[85] Murtaza Zafer, Yang Song, and Kang-Won Lee. "Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs". In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 75–82.

[86] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *EuroSys*. 2010.

[87] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Sameh Elnikety Rodrigo Fonseca, Christina Delimitrou, and Ricardo Bianchini. "Faster and Cheaper Serverless Computing on Harvested Resources". In: *SOSP*. 2021.

[88] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. "History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters". In: *OSDI*. 2016.

[89] Hua-Wei Zhou, Hao Hu, Zhihui Zou, Yukai Wo, and Oong Youn. "Reverse Time Migration: a Prospect of Seismic Imaging Methodology". In: *Earth-Science Reviews* 179 (2018), pp. 207–227.

A Pseudocode

For completeness, we list the pseudocode for the algorithms used by the SLACKSCHED Scheduler and SLACKSCHED Acquirer. Algorithm 1 is used by the scheduler to match tasks to nodes, and Algorithm 2 is used by the acquirer to select nodes to deallocate.

Note, in our implementation we ensure that the allocations and deallocations by the Acquirer do not nullify those done by the scaling policy by considering the net change in number of nodes that needs to be there. We also let the scaling policy scale out the cluster after intentional deallocations instead of Acquirer directly requesting VMs after decommissioning VMs (i.e., lines 6, 7, 8 of Algorithm 2 are coupled in our implementation). The implementation can be done in a way that avoids this coupling. Decoupling would actually be needed for cases when scaling and acquisition are triggered independently.

SLACKSCHED for heartbeat-based scheduling. In some cluster managers (e.g., Apache YARN [5]) scheduling is triggered on heartbeats [7]. In these cases, the node to schedule on is implicitly decided based on the node which sent the heartbeat. However, SLACKSCHED needs to explicitly decide which node a task should be mapped to. Thus, we adapt the scheduling logic so that the scheduler can wait for heartbeats from other nodes rather than necessarily assigning tasks to the random node that sent a heartbeat. Specifically, for the node that sent the heartbeat, if the task meets a threshold completion probability, we directly schedule the task on that node, otherwise, we look at the expected completion times offered by all the other nodes and wait for more heartbeats if there are better nodes. The pseudocode for this logic is listed in Algorithm 3. This adds negligible overhead to the time complexity of the scheduler.

Algorithm 1: SLACKSCHED Scheduler

```
1  Function MatchMaker(Task t, Nodes N)
2      W ← []
3      d ← duration(t)
4      for n ∈ N do
5          x ← lastChangeDirection(n)
6          a ← nextAvailTime(n,t)
7          e ← timeSinceLastChange(n)
8          // Using formulation in §3.1
9          c ← expectedCompletionTime(x,a,e,d)
10         W.append((c,n))
11     end
12     (c*,n*) ← min(W)
13     if feasible(t, n*) then
14         schedule(t, n*)
15     end
16     // Otherwise re-visit decision
17     // at the next scheduling event
18 end
```

Algorithm 2: SLACKSCHED Acquirer

```
1  deallocateFraction = 0.1
2  Function OnEpochEnd(Nodes N)
3      N.sort()
4      // increasing order of timeSinceLastChange
5      toDeallocate ← N[: N.size() * deallocateFraction]
6      requestNVMs(toDeallocate.size())
7      gracefullyDecommission(toDeallocate)
8      // trigger scaling
9  end
10
```

Algorithm 3: SLACKSCHED Scheduler

```
1  Function ScheduleOnNodeUpdate(Tasks T, Machine m)
2      w ← []
3      x ← lastChangeDirection(m)
4      e ← timeSinceLastChange(m)
5      for t ∈ T do
6          if ¬ feasible(t, m) then
7              continue
8          end
9          d ← duration(t)
10         p ← completionProbability(x,e,d)
11         if p > threshold then
12             w.append((p,t))
13         end
14     end
15     if w is not empty then
16         (p*,t*) ← max(w)
17         schedule(t*, m)
18     end
19     else
20         N ← getAllNodes()
21         t ← shortestTask(T)
22         MatchMaker (t, N) // (Algorithm 1)
23     end
24 end
```

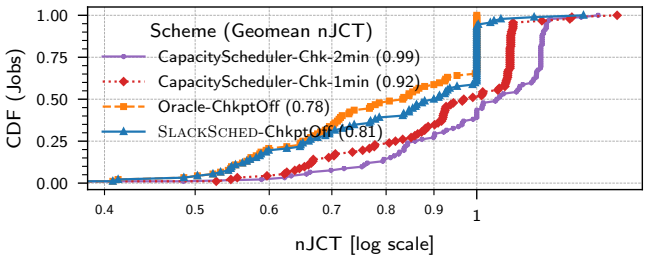


Figure 20: nJCT relative to *CapacityScheduler* without checkpointing. SLACKSCHED outperforms the checkpoint-migrate technique. We set the time-per-checkpoint as 1 min and 2 min in the two schemes with checkpointing.

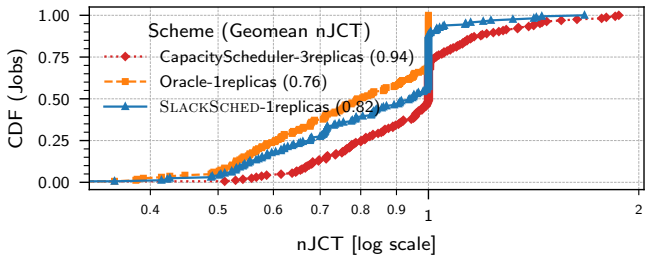


Figure 21: nJCT relative to *CapacityScheduler* without replication. Replication only slightly improves job completion time.

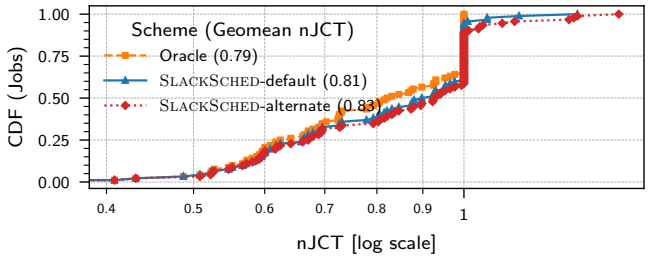


Figure 22: nJCT relative to *CapacityScheduler*. There may be multiple ways to define affinity between a task and an HVM and/or compute completion probability.

B Additional Evaluation

B.1 Comparison to prior work for Spot VMs

Prior work [28, 29, 71, 80] employs checkpointing, migration, and replication techniques to mitigate the impact of preemptions. We empirically show that these techniques are insufficient to curb the overheads imposed by HVMs on our target workloads.

Checkpointing and migration. A common approach to handle Spot VM preemptions is to periodically checkpoint tasks [28] and when they get evicted, restore them in other servers. Universal checkpointing techniques like the one docker uses based on CRIU [26, 31] only handle effects inside the container and ignores any external ones (e.g., external storage for intermediate outputs). The workloads we study do not perform checkpointing. On preemptions, tasks are simply restarted inside a new container with reinitialized external storage for intermediate results and outputs. Implementing efficient and complete checkpointing would require application specific insights and expensive engineering.

For the sake of evaluation, we measure overhead for checkpointing memory and processor state using the docker mechanism [31][8]. This checkpointing delays tasks by ≈ 1–2 minutes per checkpoint even for containers with less than 1 GB of memory[9]. This relatively large delay is because our applications rely on stateful shared libraries, open multiple TCP connections, and open a number of files in memory. These findings are consistent with [19].

To study the end-to-end impact of checkpointing, we extend our simulation framework to stall tasks being checkpointed. We delay each task by X=1,2 minute(s) per checkpoint (independent of container memory size) and run checkpoints every 10 minutes. On preemptions, tasks are restarted from the latest checkpoint and assume no overhead to migrate checkpoints. Figure 20 shows the normalized job completion time. We find that even with such optimistic checkpoint-migrate overheads, SLACKSCHED outperforms checkpoint-migrate techniques.

Replication. Another alternative to handle preemptions is to launch multiple tasks replicas and once one completes, kill the rest. This technique is common in MapReduce systems [29]. In our implementation, we launch 3 replicas for each task and kill other replicas as soon as any replica completes. Figure 21 shows that while replication reduces JCT for the *CapacityScheduler*, SLACKSCHED outperforms both *CapacityScheduler* with and without replication. In addition, replicas lead to a higher number of total preemptions and a much higher load.

B.2 Alternate method for computing the completion probability

There are other ways of computing completion probability beyond the method presented in §3.1.1. One notable way is to construct the distribution of time between consecutive resource change and resource shrink events[10] and then compute completion probability as:

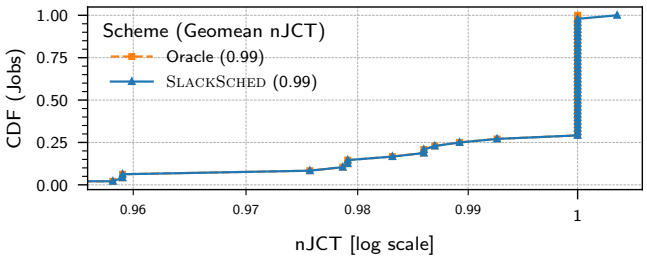
$$\begin{aligned} & P_c(e,d) \\ &= P\begin{bmatrix}\text{shrink does not occur}\\ \text{during task lifetime}\end{bmatrix} \\ &= P\begin{bmatrix}\text{shrink occurs}\\ \text{after task lifetime}\end{bmatrix} \\ &= P(Y > e+d | Y > e) \end{aligned} \quad (3)$$

where Y is the distribution of time between resource change and resource shrink events, e is the time elapsed since the

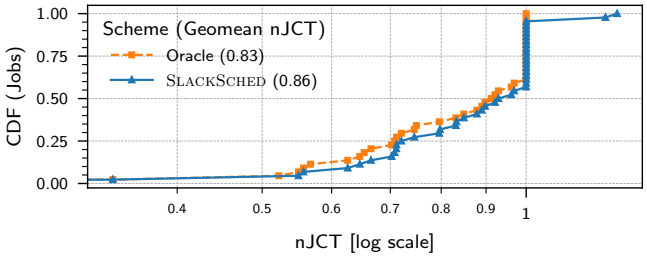
[8]These checkpoints are not restorable, a complete checkpoint would include local files and external storage/effects.

[9]The checkpointing latency (i.e., time when the checkpoint is available for restoration) is typically larger than the delay added to the task.

[10]Note, the inter-change time distribution also captures the time between two consecutive resource growth events.

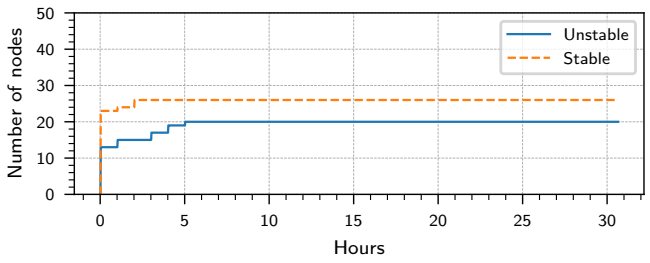


(a) Interruptible jobs.

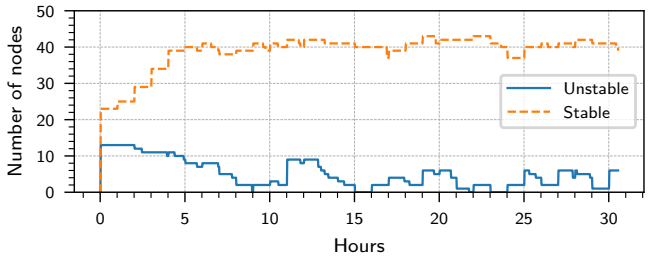


(b) Uninterruptible jobs.

Figure 23: SLACKSCHED improves JCT in a cluster serving a mix of interruptible and uninterruptible workloads.



(a) Without SLACKSCHED-Acquirer



(b) With SLACKSCHED-Acquirer

Figure 24: SLACKSCHED-Acquirer converges to a set of relatively more stable HVMs.

last resource change event, and d is the (remaining) duration of the task being considered for scheduling. We observe that using this alternate method provides similar benefits as SLACKSCHED as shown in Figure 22. The default method of SLACKSCHED provides mean reduction of 19% compared to *CapacityScheduler*, while the alternate method provides mean reduction of 17% compared to *CapacityScheduler*. The main difference is that SLACKSCHED also conditions on whether the previous resource change event was growth or shrink to compute completion probability, which gives slightly better completion probability estimates.

B.3 Mixing interruptible and uninterruptible

We study how SLACKSCHED performs in a cluster serving a mix of interruptible and uninterruptible jobs. Interruptible jobs are those that have negligible checkpointing and migration overhead.

Methodology. We simulate interruptible workloads by running checkpoints every minute and add zero overhead for checkpointing and migration. On preemption, the tasks restart from the latest checkpoint. No checkpointing or migration is done for uninterruptible jobs. The cluster serves a 50-50 mix of uninterruptible and interruptible jobs that arrive according to a Poisson process. We use the same distribution to generate task runtime and resource requirements for uninterruptible and interruptible jobs. Recall, these distributions were based on genomics and seismic workloads, which are hard to checkpoint and migrate. As such the interruptible jobs are synthetic — they do not necessarily resemble a real workload — unlike the uninterruptible jobs.

Results. Figure 23 shows the distribution of nJCT relative to *CapacityScheduler* separately for the uninterruptible and interruptible jobs. SLACKSCHED shows qualitatively similar improvement as before for uninterruptible workloads. For interruptible workloads, there is no significant difference in completion times across schemes.

B.4 Acquirer with known ground truth

To verify the operation of SLACKSCHED-Acquirer and to study its convergence properties, we study its operation on HVMs with known ground truth stability. To establish the ground truth, we generate a synthetic regular VM arrival and placement trace such that HVMs on half the servers are unstable (i.e., witness frequent regular VM arrivals and departures); while HVMs on other half are stable (i.e., witness infrequent regular VM arrivals and departures). We use the scaling policy that maintains a fixed resource budget and use the random policy for HVM allocation. We measure and compare the number of stable and unstable HVMs with and without using SLACKSCHED-Acquirer. Figure 24 shows that the Acquirer converges to a mix with a larger portion of stable HVMs.

B.5 Absolute JCT metrics

Due to space constraints we show normalized JCT for most experiments in the main text. Here we show graphs for absolute JCT. In all our experiments, we observe that mean reduction in JCT follows similar trends as reduction in mean JCT. We compute reduction in mean and p90 JCT relative to *CapacityScheduler* unless mentioned otherwise.

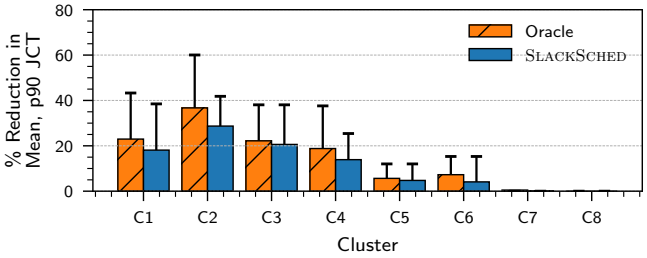


Figure 25: All clusters. cf. Figure 9. The bars correspond to mean JCT and whiskers correspond to p90 JCT.

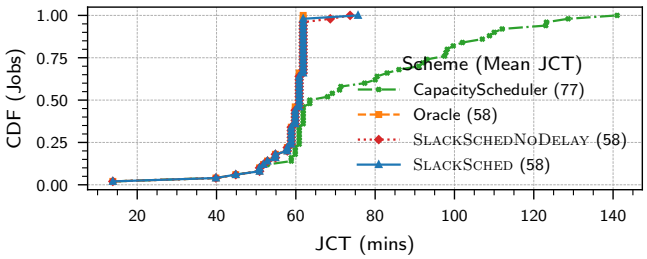


Figure 26: Stable cluster. cf. Figure 10b.

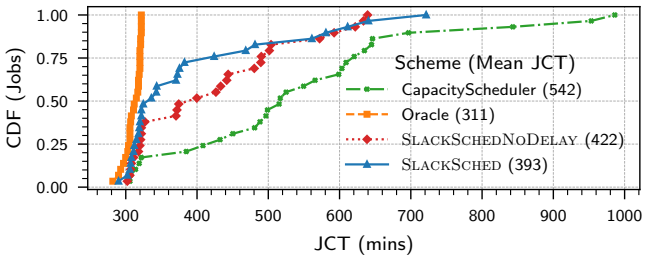


Figure 27: Volatile cluster. cf. Figure 10a.

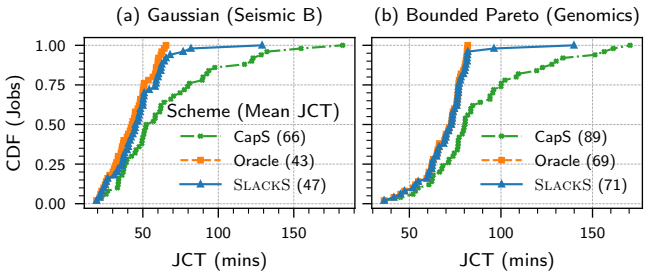


Figure 28: Different task runtime distributions. cf. Figure 11.

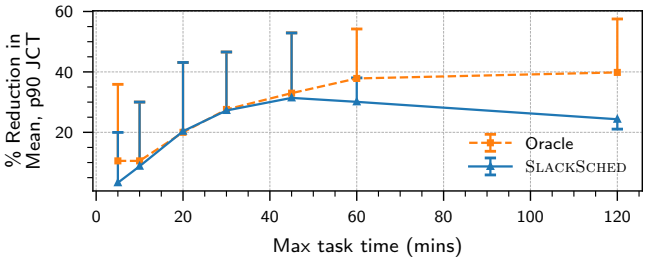


Figure 29: Varying max task time. cf. Figure 12. The lines correspond to mean JCT and whiskers correspond to p90 JCT.

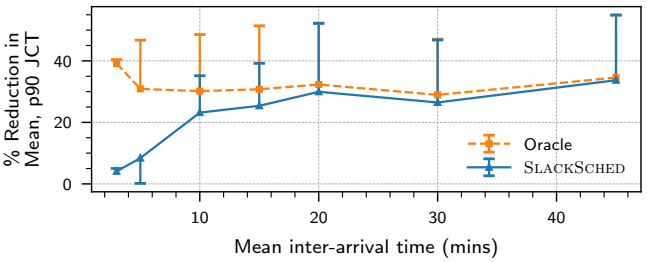


Figure 30: Varying load (job arrival rate). cf. Figure 14. The lines correspond to mean JCT and whiskers correspond to p90 JCT.

Figure 31: Robustness to errors in task runtime estimates. cf. Figure 15.

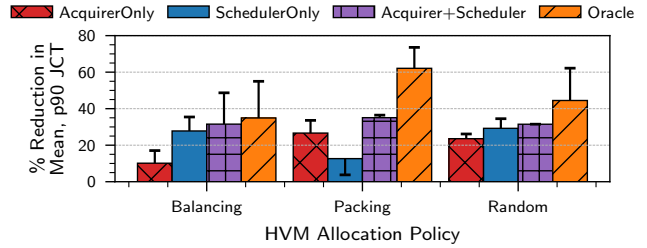


Figure 32: Resource acquisition evaluation. cf. Figure 16.

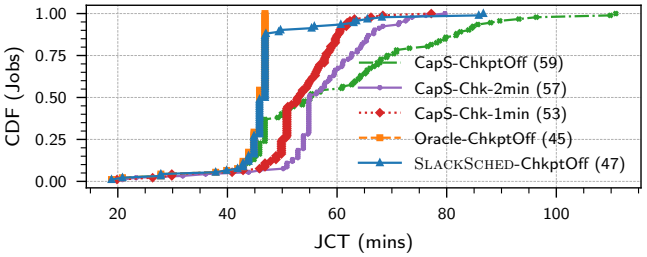


Figure 33: Comparison with checkpointing and migration. cf. Figure 20. Time in legend shows time-per-checkpoint for *CapacityScheduler*.

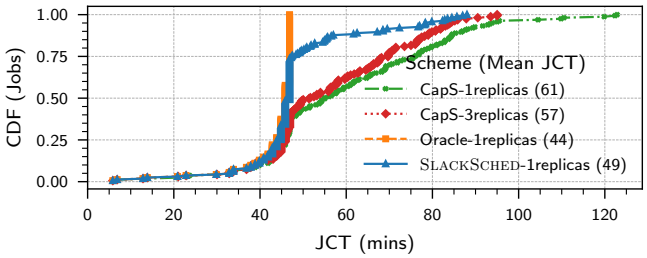


Figure 34: Comparison with replication. cf. Figure 21.

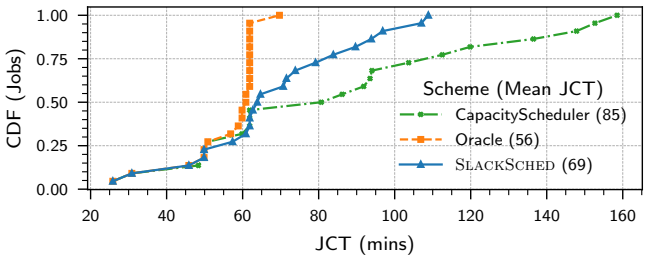


Figure 35: Renewable energy setting. cf. Figure 19.

Invisinets: Removing Networking from Cloud Networks

Sarah McClure[⋆]*, Zeke Medley*[⋆]*, Deepak Bansal*[∗]*, Karthick Jayaraman*[∗]*, Ashok Narayanan*[†]*,*
Jitendra Padhye[∗]*, Sylvia Ratnasamy*[⋆†]*, Anees Shaikh*[†]*, and Rishabh Tewari*[∗]
[⋆]*UC Berkeley* [†]*Google* [∗]*Microsoft*

Abstract

Cloud tenant networks are complex to provision, configure, and manage. Tenants must figure out how to assemble, configure, test, *etc.* a large set of low-level building blocks in order to achieve their high-level goals. As these networks are increasingly spanning multiple clouds and on-premises infrastructure, the complexity scales poorly. We argue that the current cloud abstractions place an unnecessary burden on the tenant to become a seasoned network operator. We thus propose an alternative interface to the cloud provider's network resources in which a tenant's connectivity needs are reduced to a set of parameters associated with compute endpoints. Our API removes the tenant networking layer of cloud deployments altogether, placing its former duties primarily upon the cloud provider. We demonstrate that this API reduces the complexity experienced by tenants by 80-90% while maintaining a scalable and secure architecture. We provide a prototype of the underlying infrastructure changes necessary to support new functionality introduced by our interface and implement our API on top of current cloud APIs.

1 Introduction

Almost all large cloud customers use multiple cloud providers to improve reliability and avoid provider lock-in [57]. Unfortunately, splitting a workload among large cloud providers is not as seamless as it should be. One major problem is that today's tenant networking abstractions are essentially virtualized versions of the low-level building blocks used to build physical networks and hence customers are required to craft complex topologies using subnets, virtual links, gateways, a myriad of virtual appliances, *etc.*

While an individual virtual network or simple deployment may not be overly complex, for larger tenants, reasoning about the scalability, availability, security, *etc.* of their virtual networks requires detailed knowledge and configuration of a range of networking technologies, and the problem is particularly acute when considering transit between clouds.

In response to this underlying complexity, our goal is to make multicloud architectures simple by making this inter-virtual-network transit trivial. We achieve this primarily through leveraging existing "publicly-routable but default-off" addresses for all endpoints. These addresses are publicly-routable, but the cloud provider will deny all traffic not specifically permitted by the tenant. With this choice, connectivity becomes trivial with only minor infrastructure changes.

With this connectivity assumption, we develop a new API for cloud tenants to reserve networking resources based on high-level abstractions. Ultimately, this results in a declarative approach that allows tenants to essentially associate SLOs to endpoints, without concern for *how* to achieve their networking goals. In other words, *we believe that the best way for tenants to think about networking is to not think about networking at all.*

Today's tenant networking abstractions are largely relics of the early cloud era, with a 1:1 mapping between datacenter physical network devices and cloud networking abstractions. This appealed to early cloud customers who wanted the same management experience as they shifted infrastructure from private datacenters to the cloud, but these abstractions are not fundamentally necessary nor especially accurate to what occurs in the cloud provider's network underneath. A key contribution of our paper is to show that even a modest refactoring of the existing functionality and modification of the underlying network virtualization platform can result in a far simpler higher-level API.

Such an API is beneficial to cloud providers as well as it allows them to offer their customers a seamless multicloud experience, and yet retain the ability to differentiate (*e.g.,* through performance, options, *etc.*). Furthermore, and perhaps more importantly, the API is far less complex than the low-level APIs offered today. A simpler API means fewer mistakes and thus fewer resources dedicated to customer support.

We recognize that our proposed cloud interface may not be immediately appropriate for all cloud tenants. Fortunately, the abstractions we propose can coexist with those available today so that tenants may choose the tradeoff they desire.

This paper presents the rationale (§2), design (§3-5), implementation (§6), and evaluation (§7) of a new API – which we refer to as *Invisinets* – for tenant networking. This work builds on our earlier workshop paper [41], refining and expanding the API as well as adding a complete implementation and evaluation. Our evaluation applies this API to several deployment case studies and measures several metrics to capture complexity. Accordingly, we show that Invisinets can reduce the number of network components that a tenant must interact with by ∼80%-90% in these scenarios.

2 Motivation

In this section, we discuss the status quo to motivate the need for a simpler tenant networking API. Throughout this paper, we default to Azure terminology for cloud network

components, though equivalent ones are typically available in other clouds.

2.1 Deployment Walkthrough

Consider an enterprise tenant whose workloads span multiple regions within a cloud, multiple clouds, and an on-prem deployment. To highlight the complexity that enterprise network engineers encounter, we will walk through the steps to construct such a virtual network today. At a high level, tenants must complete the following 5 steps to create a deployment.

(1) Create virtual networks. The basic construct in a tenant's network is a *virtual network* (or VPC in the terminology of AWS and GCP) which builds on the traditional concept of a subnet – a portion of the network whose hosts share a common IP address prefix. The tenant's first step, creating their virtual networks (or vnets), involves assigning IP prefixes to each vnet based on the anticipated number of instances, whether they should be IPv4 or IPv6, public or private, *etc.* These are important decisions as a particular choice (*e.g.,* IPv4 vs. IPv6) leads to a separate path down the decision tree of subsequent connectivity options. As a tenant's deployment scales, managing non-overlapping subnets across 100s of vnets becomes challenging, prompting special address planner tools [9]. Beyond address management, defining a vnet involves configuring a range of additional parameters such as security groups, ACLs, route tables, and individual VM interfaces. For simple deployments, this complexity can be hidden via default configurations but this approach rapidly breaks down with larger deployments.

(2) Connectivity in/out of the virtual network. Next, the tenant must define how instances within a vnet access resources outside the vnet. Options include setting up a "NAT Gateway" (for address translation), a "VPN Gateway" (for private VPN connectivity), or a "Virtual Network Gateway" (which can be configured for VPN or to terminate a direct link). AWS has even more gateway options, some simply to allow the virtual network to access the Internet [8]. Each gateway must be configured with the appropriate routing and access policies.

(3) Networking multiple virtual networks. Next, the tenant must interconnect these virtual network. Within a cloud, this generally requires a virtual network peering and installing the necessary routes, though these links often have some regional limitations. To connect virtual networks across clouds, VPN gateways or Internet access (via necessary gateways or security rules) will be necessary. Each of these connections come with their own specific configuration parameters.

(4) Specialized Connections. An increasingly common component in large tenant networks are dedicated connections that promise higher bandwidth, lower latency, and/or more consistent performance than seen on the public Internet (*e.g.,* ExpressRoute [44], Dedicated Interconnect [28], or Direct Connect [7]). These allow tenants to reserve a physical, dedicated link between their virtual network and a colocation facility. From there, the enterprise may complete the circuit to their on-prem resources or stitch the connection together with one from another cloud. Provisioning and managing these links requires low-level networking knowledge such as BGP configuration and coordination between the enterprise, cloud provider, and the colocation point. Since these dedicated connections are expensive, tenants might configure their routers to schedule higher priority or sensitive traffic over these links, while routing other traffic over the public Internet.

(5) Appliances. The above steps establish a basic topology and connectivity between the tenant's instances, but tenants also deploy a range of virtual appliances such as load balancers and firewalls. Each cloud offers both first-party and third-party appliances for many of these purposes. Even within the first-party selection, there are often multiple options for a single appliance. The tenant must select appliances, place them in their virtual topology, configure routing to steer traffic through the right appliances, and finally configure each appliance (*e.g.,* DPI rules, load-balancing rules, *etc.*).

Once the tenant has completed all of these steps, the job is not done. The tenant must continue to respond to changing requirements, application and network migrations, inevitable configuration mistakes, and outages caused by any other issue. When determining their procedure for confronting these issues, tenant network operators must keep all details from the above steps in mind, acting as a seasoned network expert.

2.2 Problems

We briefly highlight the main sources of complexity that we observe from the above walkthrough.

(1) Abstractions are too low-level. Provisioning and managing a virtual network involves many of the same steps as in a physical datacenter. Tenants are essentially given virtual versions of the low-level abstractions found in a physical network (*e.g.,* links, gateways, subnets) and must assemble these (which involves topology planning, routing policies, *etc.*) to achieve their higher-level intents for the overall deployment. Many of these abstractions require addressing configuration in particular to achieve basic connectivity between tenant applications/endpoints.

(2) Complex planning. Beyond determining the topology of the deployment, cloud marketplace options can make it difficult to determine the correct virtual appliance. For example, Azure offers four load balancing options and the flow chart to guide the decision is five layers deep [43] and this does not consider other third-party (*i.e.,* non-Microsoft) options. Cloud appliance marketplaces also feature third-party options (*e.g.,* firewalls from Palo Alto Networks) that vary in features and price points. A cottage industry of businesses offering answers on how to minimize cloud costs has appeared to help tenants with this problem in recent years [10, 17–19, 66].

(3) Fragmentation across clouds. As each cloud has its own similar yet different abstractions and appliances, tenants with multicloud deployments end up with siloed stacks for each cloud. This often results in teams dedicated to each cloud

with their own expertise, scripts, and approaches.

(4) Complex to maintain and evolve. As the requirements for their applications change, tenants will evolve their own deployments. Likewise, tenants must adapt as cloud providers evolve their offerings. This adds complexity and management overhead to the existing challenge of keeping applications modern and performant for tenants. Misconfigurations are common causes of network incidents [36] and outages ([35, 62] provide recent high-profile examples). With 1:1 abstractions, virtual networks suffer many of the same issues. The cloud providers also suffer from this as well, as they are obligated to provide support to a wide array of clients with unique deployments. Further, management complexity increases as the deployment size increases, so large enterprise tenants suffer significant management burdens.

In summary, tenant networks today are constructed from low-level building blocks that are unique to a given cloud. With the growing popularity of large, multicloud deployments, this complexity can compound and become even more difficult.

2.3 Current Solutions

In our experience, many enterprises undertake this complexity themselves using an array of per-vendor controllers and DIY scripts. This often requires a team that understands networking in all its gore: BGP, address management, VPN config, *etc.* These teams must understand multiple cloud environments, which change frequently and outside of their control.

Other tenants are turning to a new generation of 3rd-party multi-cloud management solutions [5, 12, 67, 68]. Some of these solutions are essentially a shim on top of the various cloud networking abstractions. They provide a unified "pane of glass" via which the tenant manages individual devices across clouds [12, 67] but do not fundamentally change the level of abstraction; *e.g.,* a key component in Aviatrix deployments is a transit router abstraction that interconnects virtual networks. Yet other 3rd-party solutions essentially run a virtual network as a service for tenants [5, 68], which allows tenants to completely outsource the problem. This shifts the burden but does not fundamentally solve it.

Anthos [22] integrates k8s and service meshes in a manner that frees app developers from having to reimplement common networking related tasks on a per-app basis. With Anthos, every service container is integrated with a "sidecar" container that implements common network-related tasks such as TLS termination, HTTP load balancing, tracing, and so forth. This clean separation between app and networking concerns gives app developers a cloud-agnostic (and hence multicloud friendly) approach to implementing network-related features. However, it is important to note that Anthos does not address the problem of network virtualization that we address here: in Anthos, sidecars are L7 proxies and (like all k8s services) they assume L3 addressing and connectivity has already been established. Implementing that L3 connectivity still requires a network engineer to set up the virtual networks, links, VPN gateways, *etc.* that we have been discussing [30, 33]. Thus we view the goals of Anthos and Invisinets as complementary: Anthos simplifies the construction of multicloud deployments for app developers, while Invisinets does the same for infrastructure operators.

2.4 It's Time for Simplification

Network virtualization technologies were originally designed to allow cloud providers to virtualize their *physical* network infrastructure [34, 53]. In this context, providing the user (in this case, the datacenter operator) with virtualized equivalents of their physical network is appropriate, and we do not question the approach.

Extending the same approach to cloud *tenants* also made sense in the early days of cloud adoption when enterprises with well-established on-prem datacenters often used the so-called "lift-and-shift" strategy: creating a networking structure that mimics the on-premises network that previously served the workload. This strategy was justifiably appealing as it allowed tenants to use familiar tools and tested configurations in their new cloud deployments. However, we see this approach as neither desirable nor necessary as tenants embrace the cloud more fully in both the scope of their deployments and in (re)designing applications for cloud environments.

Nonetheless, we recognize that certain enterprises may choose to continue with building virtual networks for reasons that range from satisfying compliance requirements (*e.g.,* with data protection laws [55,65]), to familiarity with existing tools, and the perception of greater security. Fortunately, this need not be an either-or decision: the architecture we propose can be deployed alongside existing solutions allowing tenants to choose whether and when to migrate their workloads.

Our approach requires new support from cloud providers, but we believe this is reasonable since the current situation is non-ideal even for cloud providers. The current complexity imposes a steep learning curve for onboarding new customers, and plenty of room for configuration errors that will, regardless of fault, result in unhappy customers. Simplification can decrease the number of tenant errors and therefore decreases the support burden on the cloud provider. Further, the cloud provider could likely achieve higher resource efficiency by taking control of networking and orchestration from tenants.

3 Approach

Our guiding philosophy in designing a simplified networking API is that *the right way for a tenant to think about the network is to not think about it at all*. I.e., ideally, the network should be invisible. When diagnosing the root cause of today's complexity, we arrive at the observation that the problem starts with the fact that tenant endpoints live in a private IP address space. Given private addresses, tenants must then establish (virtual) connectivity between them which necessarily implies managing subnets, constructing a virtual topology

with links, routers, and appliances, running routing protocols that must be configured, and so on.

Seeking to avoid this leads us to an alternate proposal: *can we give every endpoint a public IP address?* This makes connectivity trivial, notably even across clouds. Essentially, this allows virtual networks to reuse the connectivity of the underlying (physical) network rather than recreating it. Importantly, we must assume IPv6 so that address scarcity is not an issue.

At first glance, our proposal might seem concerning from the viewpoint of security: if any host on the Internet can reach any tenant endpoint, then surely we're exposing a tenant's endpoints to attack, including DDoS. Yet, our intuition was that security should not be a concern for public endpoints *that are hosted in the cloud*. This is because cloud providers (or CPs for brevity) have already addressed this problem with their own address management architectures. As we elaborate on in the next section, within a CP's infrastructure, when an endpoint is assigned a public IP address P[1], this address is not actually routable *within* the CP's regional datacenters. I.e., the endpoint associated with P is not actually hosted on a server with the address P. Instead, the server at which the endpoint runs has an internal/private address D and the CP uses solutions to translate P to D with appropriate security and access control checks at the point of translation.

Thus, instead of the typical public *vs.* private address tradeoff, P represents a new form of address that is publicly routable but default off (PRDO), with the important property that a packet destined to a PRDO address is delivered to the CP 's domain but will not be delivered to an endpoint until the CP has *explicitly taken action to associate P to a physical endpoint's D*. Thus our intuition was that we can leverage the PRDO addressing architecture to spare tenants from having to solve the connectivity problem in their virtual networks.

Given this, our next step was to verify our intuition and understand whether CP address management infrastructure could indeed be leveraged and extended to serve all tenant endpoints. To answer these questions, we engaged with two major CPs (Azure and GCP) and found that, perhaps surprisingly, our proposal could be supported with little modification to their existing infrastructure and raises no new scaling or security challenges. We elaborate on existing CP address management infrastructure and the implications of PRDO addressing in §4. Thus our contribution lies not in devising novel techniques or radical clean-slate designs but rather in proposing a radically simplified tenant abstraction and showing how we can repurpose existing infrastructure to implement this abstraction.

With PRDO addressing as our starting point, what API can we offer tenants? We observe that, for the vast majority of cloud tenants, the network is a means to an end: *i.e.,* tenants care that their application endpoints (*i.e.,* VMs or containers) can communicate with each other with high availability, a certain level of performance (*e.g.,* latency, throughput), security against unwanted access, and scalable mechanisms for management. These are the goals that a tenant is trying to achieve when they set up and manage a virtual network topology with firewalls, links, and routers. Hence, we design an API that allows tenants to express *what* they want the network to accomplish, rather than *how* it does so. Moreover, we note that the parameters that specify a tenant's goals are associated with how *endpoints* – VMs/containers – experience the network. Hence, our general approach is to assign PRDO addresses to all tenant endpoints and then provide tenants a high-level API that associates SLOs (for availability, security, and performance intents) with these endpoints (or groups of endpoints), thus completely eliminating today's low-level "links and boxes" abstractions.

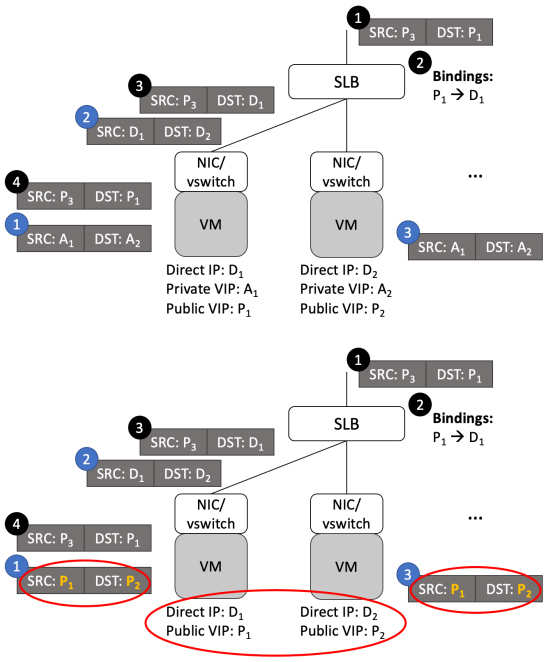


Figure 1: Cloud addressing as it is done today (top) and our proposed changes (circled, bottom). Black packets show the translation and path of packets between VMs in the same virtual network while the blue path shows an external connection using the VM's public IP.

We do not require that CPs cooperate to implement the API, nor even that all CPs adopt the API. Likewise, CPs don't have to implement identical versions of the API. Converging on a single API across all CPs would certainly be ideal but, even if each CP adopts their own flavor of the API we propose, tenants will benefit from the simplification in configuring that CP, and the high-level nature of the API will make it easier to port deployments across clouds.

4 PRDO Addressing

In this section, we first review the addressing architecture commonly implemented by CPs today (§4.1). This discussion reiterates information published before [20, 21, 54] but distills

[1]By which we mean an address from the address space the CP advertises into the Internet's routing infrastructure - *e.g.,* [27] for GCP

the aspects relevant to Invisinets. We then discuss how this addressing infrastructure can be used to support Invisinets (§4.2) and the security implications of the same (§4.3).

4.1 Addressing in Today's Clouds

A cloud provider's infrastructure involves a few different address spaces. Public IP addresses (PIPs) are drawn from public prefixes that the CP advertises into the Internet at large. CPs vary in the details but, generally speaking, a packet destined to a PIP will be delivered to a specific datacenter or region in the CP's global infrastructure.

In addition, each cloud datacenter has a private address space of "direct" IP addresses (DIP) – a DIP is the actual IP address of the physical server and the basis for routing within a datacenter fabric. It is used internally to the CP and is never exposed to the tenant, providing a layer of indirection and allowing the CP to place/move VMs as needed.

Finally, at the virtualization layer, tenants see a virtual IP (VIP) for their VM or endpoint. A VIP may be public or private (a VM may have both) but crucially is not actually assigned to the server and therefore is not routable in the datacenter fabric. Instead, it is used at the tenant-layer for virtual network-level routing, *etc.* For packets sent between two VMs with private VIPs in the same virtual subnet (A_i addresses), the source and destination fields are translated in the vswitch/NIC at the VMs; the vswitch/NIC translates from the private VIP address space to the corresponding DIPs ($A_i \leftrightarrow D_i$) [25, 54] as shown in Figure 1 with the blue series of packets. The fabric only understands how to forward traffic with DIPs and is unaware of the tenant address space (giving the virtualized layer full flexibility in assigning addresses).

Endpoints with public VIPs are assigned an IP address drawn from the CP's public address space (P_i). However, this address is not directly assigned to the VM. Instead, any incoming traffic destined to P_i is first routed through a software load balancer (SLB) in the datacenter (see Figure 1 black series of packets) that maintains a binding from P_i to D_i.[2] The mapping between public VIPs and DIPs are installed when the public IP is provisioned and associated with an endpoint. Thus the SLB advertises all public VIPs under its control and translates incoming traffic to the endpoints DIP to route to the endpoint. For traffic exiting the VM, no translation is necessary as the source is the public VIP and default routes are used to exit the datacenter.

Thus in the terminology introduced earlier, all PIP addresses (including ones assigned as public VIPs) act as PRDO addresses, requiring explicit SLB and vswitch/NIC configuration before packets can be delivered to an actual endpoint.

4.2 Applied to Invisinets

The core proposal in Invisinets is that we will no longer use private VIPs at all and instead give all endpoints a public VIP (Figure 1). Thus, tenant addresses will be drawn from a CP's public IP address space and we will leverage the CP's PRDO addressing infrastructure to ensure that a tenant's endpoints are not exposed to unwanted access.

To leverage existing CP solutions, we propose a division of labor in which tenants specify *permit lists*: for each endpoint P_i, this is the list of other endpoints that are allowed to communicate with P_i (with the possibility for extension parameters for more specificity) .[3] The CP is responsible for enforcing this permit list: ensuring that only traffic explicitly permitted to access the endpoint may do so. Any packet not cleared by the permit list should be dropped.

Ensuring these semantics to the tenant requires minimal changes to the CP's infrastructure. The CP simply programs SLB bindings for external connections only as necessary based on the tenant's permit list. We use "external" to mean outside any boundary necessary to meet the CP's addressing constraints; *e.g.,* DIPs may only be unique within a region, therefore "external" connections are those spanning beyond the region. We call the boundary within which a DIP is unique the *DIP scope*. The DIP scope determines when an address must be added to the SLB bindings as any connections to endpoints outside the DIP scope may be in overlapping address space and will not be reachable without address translation.

To support this in the datacenter fabric, we first modify the address translation for traffic which typically uses private VIPs (the blue process in Figure 1). Now, packets are translated from the public VIP space to the corresponding DIPs in the vswitch/NIC. Second, the SLB translation process for packets incoming from external endpoints must be slightly modified. The translation between public VIPs and DIPs in the SLB remains the same, but the time at which the SLB binding is installed is changed. Instead of installing when the IP is associated with the VM, the binding is only instantiated when the VM's permit list allows traffic external to the datacenter. When no mapping exists for an incoming packet, the SLB simply drops the traffic (as it does today). This provides an initial coarse layer of protection for PRDO endpoints. Importantly, each endpoint's permit list is enforced at the host as well, so dropping at the SLB is not strictly necessary but offers some defense-in-depth for endpoints.

We assume that in order to support this model, the CP can allocate addresses arbitrarily as is convenient (*e.g.,* the CP isn't required to assign addresses from predetermined prefixes of the address space). The tenant should not expect any particular addressing scheme for the IPs they are assigned. Thus, our API takes reachability (L3) between endpoints as a given from the CP and builds directly on the underlay routing and address translation rather than requiring every tenant to reconstruct their own L3 network.

[2] The SLB is often implemented as a distributed scale-out software system capable of high-speed address translation [21, 54].

[3] For convenience, we will introduce the ability to specify *groups* of endpoints in §5.

4.3 Security Implications

We argue that Invisinets does not fundamentally alter the security posture of either the tenant or CP.

Consider first the public exposure of a tenant's endpoints. Today, the tenant has 3 types of endpoints: (1) public (*e.g.,* a VM associated with a public IP), (2) "semi-private " (*e.g.,* a VM with only a private VIP, but behind a VPN/NAT that restricts access to the VM), and (3) private (*e.g.,* a VM with a private VIP). Under our proposal, nothing changes for endpoints in (1). For endpoints in (3), they are equally unavailable to outside connections in Invisinets (since no SLB bindings will be programmed for them). [4] For endpoints in (2), access is still limited however Invisinets does change *how* this access control is implemented. In Invisinets, access to a semi-private endpoint is restricted by the permit list and SLB bindings; effectively, we're replacing the tenant's VPN/NAT with the CP's SLB/NAT infrastructure.[5]

This change does however, modify the trust boundary between the tenant and the CP. Today, the tenant trusts the cloud provider to implement the network architecture they specify faithfully. In Invisinets, the tenant trusts the cloud provider to construct the necessary network architecture to achieve their connectivity requirements and accordingly trusts that any connectivity not explicitly permitted will not be enabled.

From a CP viewpoint, a potential concern is that the larger number of allocated public IP addresses increases the risk or impact of a DDoS attack on their infrastructure. We found that this was not a concern for CPs because they currently advertise entire IP prefixes, independent of what subset of those addresses are actually allocated: *e.g.,* Azure advertises $>1.4 \cdot 10^{20}$ [46] while GCP advertises $> 7.1 \cdot 10^{26}$ IPv6 addresses [27]. Any of these addresses can be used as the target of a DDoS attack and this remains unchanged with Invisinets.

Furthermore, CPs today deploy cloud DDoS mitigation systems [45] to thwart high-volume traffic attacks targeting a specific endpoint (since these might overwhelm specific links in their WAN). Notably, only one public IP is sufficient for an attacker to attempt such an attack and hence Invisinets does not materially increase the likelihood of such attacks. A final concern might be attacks that spread traffic over multiple endpoints. However, such attacks are already possible today (given the large advertised address space) and handled through filtering at the destination datacenter where SLB mappings reveal what traffic is valid. In such cases, attack traffic is carried over the CP's wide-area links, however the extensive use of load-balancing in these networks is effective in this case (since the attack comprises many smaller flows) and hence the increase in backbone traffic volume is not problematic, akin to a general increase in valid traffic volume.

[4] However, if a private endpoint today is in a virtual network which spans across more than one DIP scope, it would be installed in the SLB.

[5] In some cloud SLB implementations [21], tenant-level NATs are implemented in the SLB. In this case, the only change is that each endpoint has its own binding in the cloud SLB.

API	Description
`request_eip(vm_id)`	Grants endpoint IP
`request_sip()`	Grants service IP
`bind(eip, sip)`	Binds EIP to SIP
`set_permit_list(eip, permit_list)`	Sets access list for EIP
`annotate(eip, middlebox)`	Adds middlebox to EIP's traffic
`set_qos(region, dest, bandwidth)`	Sets (region, dst) BW allowance
`set_qos_class(class, five_tuple)`	Defines tenant QoS class
`tag(eip, tag)`	Associates endpoint with tag

Table 1: Proposed cloud tenant network API.

5 API Design

In this section, we outline the Invisinets API and and describe how it achieves each of its goals. The design implications for each piece of the API are discussed as necessary. We expect the listed arguments to be the minimum required parameters and deliberately leave room for cloud providers to differentiate their services with additional extension parameters. A complete list of our proposed API is shown in Table 1.

5.1 Connectivity

Rationale. As explained in §3, we modify the CP infrastructure to support PRDO addressing for all endpoints and enable trivial connectivity. By giving all endpoints publicly-routable IP addresses, we can abandon the virtual network altogether. Accordingly, tenants are not obligated to construct the networks to facilitate communication outside of a given virtual network, as is required by the inherent isolation of the virtual network abstraction.

API. Connectivity between the tenant's VMs/storage endpoints/*etc.* in the same cloud, across clouds, and to their on-prem network (provided they expose public endpoints) is trivially achieved given that tenant instances have public IP addresses. Thus our basic `request_eip` API allows the tenant to request and receive an *endpoint IP address (EIP)* for each of its instances. A tenant must be prepared to treat its EIP as a flat address with no assumptions about whether its addresses can be aggregated, drawn from certain prefixes, *etc.* This gives providers flexibility in assigning addresses from their overall pool (*e.g.,* to maximize the ability to aggregate for routing, *etc.*) and with effective tagging mechanisms should not affect tenants in any way (since tenants are no longer configuring routing with mechanisms such as BGP).

Design. As discussed in §3, the infrastructure necessary to support PRDO addresses requires only minor modifications to the SLB in the datacenter fabric and otherwise the existing vswitch/NIC functionality is sufficient. When an external endpoint is added to a permit list, only then is the EIP of the local endpoint installed in the SLB.

IPv6 will be necessary to support PRDO-only addressing as the IPv4 address space is too small to feasibly give all cloud endpoints an address. Since these addresses will be allocated arbitrarily from the tenant's perspective, grouping mechanisms will be critical to managing a flat address space. We address this need directly in §5.5.

5.2 Availability

Rationale. Tenants often build highly-available services using multiple backend instances. The service is associated with a service IP address (SIP) and an in-network load balancer maps traffic destined for SIP to an available backend instance. We'd like to support this use-case without requiring that tenants engage with the lower-level details of load balancers.

API. We allow tenants to request a SIP and introduce a *bind* API that allows tenants to associate EIPs with a SIP (Table 1). This SIP address is globally routable, however, traffic destined for the SIP is routed / load-balanced across the EIPs bound to it and we place the responsibility of load balancing on the cloud provider. Hence, the `bind` call allows the tenant to inform the cloud provider of how load-balancing should be implemented, with optional parameters that guide load-balancing policy (*e.g.,* associating a weight with each EIP, akin to weighted fair queuing policies).

Design. Requesting SIPs can be supported today and is in fact somewhat similar to service mesh integrations with cloud load balancers today [29]. The `bind` call only requires changes to the interface tenants use request load balancing services.

5.3 Security

Rationale. In §4.3, we addressed the security implications for the network fabric. We now discuss how tenant-level security concerns are addressed in the Invisinets API. We assume the tenant is primarily concerned about service-level attacks and targeted resource exhaustion. To address potential attacks, our architecture will implement tenant-level security in two main pieces: middlebox annotations on endpoints and network-level permit lists. Both of these are specified by the tenant but implemented/enforced by the CP.

Today, tenants protect their services through a combination of per-VM/virtual network permit lists and by deploying various security appliances such as first-party firewalls and third-party DPI systems (*e.g.,* [52]). Network permit lists are specified by the tenant but implemented by the cloud provider (typically through filtering in the vswitch/NIC at each endpoint) while security appliances may be deployed and managed by the tenant. Together, these mechanisms protect the tenant from both service-level attacks (*e.g.,* intrusion or exfiltration attacks caught by DPI firewalls or proxies) as well as resource exhaustion attacks that specifically target the tenant (*e.g.,* overwhelming the tenant's service).

Ideally, we would remove middleboxes from the cloud offerings altogether and instead implement security measures in an API gateway in the service itself [6, 63]. However, we acknowledge that some applications may have strict security requirements that demand middleboxes and therefore we expand our API to include those available today.

API. The per-host tenant-level permit lists can be naively implemented at the endpoint with the access lists used today such as Network Security Groups in Azure. To the tenant, our API will look similar to these access lists. To permit from another host, the tenant simply uses the `set_permit_list` function to update the given endpoint's allowed hosts.

The tenant may use the `annotate` API to apply the desired middlebox to an endpoint's traffic. The cloud provider is responsible for the instantiation and placement of the middlebox and the routing necessary to direct the relevant traffic. The tenant will provide the type of middlebox (could be their own VM as seen with some third-party network appliances today) and configure it as done today. Additional parameters to the `annotate` API could specify a subset of the traffic to be sent through the middlebox (*i.e.,* by destination) or specify the ordering of a series of middleboxes.

Design. We propose a two-pronged approach to protect tenants from attacks. First, we allow tenants to continue their use of cloud middlebox offerings by annotating endpoints. Therefore, they may continue to use their DPI firewalls, IDS/IPS appliances, *etc.* as they do today. However, the tenant does not have to manage the placement of these appliances in their networks and route relevant traffic. The cloud provider will install the necessary routing in the vswitch/NIC. By including security-focused middlebox functionality in our API, tenants can continue their defense-in-depth best practices as they do today. Secondly, we propose that the cloud provider protect the tenant from network-level resource exhaustion attacks by reusing the same infrastructure it already implements to protect itself. In addition to the above, we assume the cloud provider will continue to enforce the tenant's permit list through filtering at the endpoint's vswitch/NIC. These permit lists are essentially available today as NSGs (Azure) and Security Groups (AWS), so we adopt the underlying architecture unchanged.

5.4 Performance

Rationale. Today, cloud providers offer rather limited network performance/QoS guarantees. Tenants are generally not promised any minimum bandwidth and are instead throttled above a certain threshold. However, some tenants seeking high availability and reliability may reserve a dedicated link [7, 28, 44] which the tenant must then provision, configure and operate as discussed in §2.

The abstraction of a dedicated link is fundamentally at odds with our goal of a high-level endpoint-centric API since a link implies a topology that incorporates it and routing that steers select traffic over the link. Our goal is to avoid this complexity and hence we instead ask: *can we approximate the benefits of these dedicated links without obligating the tenant to worry about the many details they do today?* [6]

The point-to-point link abstraction offered today requires coordination between the entities on either end of the link and

[6] One might ask whether the performance of dedicated links justify their cost and complexity in the first place. In Appendix 11, we present early results showing that these links may not always offer a performance benefit but leave a full evaluation to future work.

is not offered directly between clouds. To avoid the coordination and low-level configuration of direct links, our proposal is to approximate the *effect* of dedicated links by guaranteeing some amount of dedicated egress bandwidth to another domain to tenants who purchase it at a predefined granularity (in this paper, we will assume regional granularity). We hope that since clouds already have an incentive to be highly connected with one another, this guaranteed bandwidth when leaving a region is enough to roughly estimate the effect of dedicated links stitched together at a colocation facility. With this overall goal, we then ask how to provide such an abstraction to the tenant as a service.

The service model we assume is in terms of bandwidth reservations, rather than point-to-point links with a specified bandwidth as is available today. Tenants will specify to the cloud provider their desired amount of dedicated egress bandwidth to another domain (*e.g.,* another cloud) for some region. Then, we seek to make the traffic management necessary to use this link as simple as possible by allowing tenants to define traffic classes and map them to strictly ordered priorities. These priorities will define which traffic gets to use the dedicated bandwidth if/when the aggregate traffic for the tenant in that region is greater than the allowed dedicated bandwidth.

API. Ultimately, the tenant will define their own traffic classes (in terms of five-tuples). The CP will then label traffic as necessary and map these classes to their own high ("dedicated") and low ("best-effort") priority classes in the fabric of the datacenter. To the tenant, reserving the regional aggregate egress bandwidth will simply require calling `set_qos` and setting the priority of traffic via `set_qos_class`.

Design. Our `set_qos` API is based on the assumption that clouds are reasonably well-connected with one another so that dedicated egress bandwidth between a cloud region and another domain can approximate a direct link between the two clouds. If not, congestion between the clouds could impact the available bandwidth beyond the control of either CP. Further, we require that the CP can classify egress traffic into reserved bandwidth packets and best effort-packets. Reserved-class packets are guaranteed to not experience congestion on egress (up to their reserved bandwidth) while best-effort may.

In offering the `set_qos` API, the CP has two primary goals: (1) enforce that the tenant does not consume more than its aggregate egress guarantee and (2) make it easy for the tenant to use all of their promised bandwidth without requiring low-level traffic engineering (as is required today to utilize dedicated links). Since the bandwidths are offered at a per-tenant, per-destination-domain, per-region level, the CP must monitor usage across multiple endpoints in a region and enforce the cumulative bandwidth limit at each endpoint. In doing so, there will be a tradeoff between reactivity and cost as enforcing the limit strictly will impose higher overheads. Scalability is also a challenge as performing distributed rate limiting across all tenant endpoints (in the 10s of millions) must be done with minimal resource consumption.

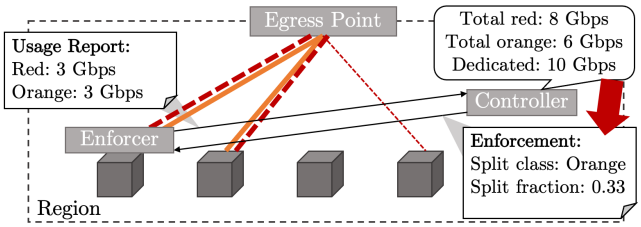


Figure 2: QoS enforcer example. The tenant sends traffic in two classes: red (dashed) and orange (solid), prioritized in that order. The reserved egress bandwidth is 10Gbps.

Our approach to enforce the `set_qos` API is as follows: we assume that tenants define traffic classes each with a different priority level and the cloud provider marks their traffic accordingly. The cloud provider then determines whether a particular traffic class (or what fraction of it) should be assigned reserved vs. best-effort bandwidth based on the tenant's current traffic demand aggregated across all its endpoints (collected at the host and reported to the controller). This mapping between tenant traffic classes and reserved vs. best-effort bandwidth is computed by a per-tenant *QoS controller*. The QoS controller communicates the appropriate mappings to a *QoS enforcer module* at each endpoint (implemented in the NIC or vswitch) which accordingly marks egress packets and rate limits reserved priority flows according to their current bandwidth allocation. Finally, the egress router classifies packets based on these markings and (priority) schedules them accordingly. Hence, the additional infrastructure that our API imposes on the cloud provider is the per-tenant QoS controllers and the QoS enforcer modules (Figure 2).

Using per-tenant controllers mitigates potential scaling issues since each controller need only scale to the number of endpoints per tenant (*vs.* all endpoints) per region, thus dividing the regional rate limiting into reasonably-sized problems.

Figure 2 shows an example of this design for a single tenant. The tenant has two traffic classes, red and orange, which are prioritized in that order. The communication between only one enforcer and the controller is shown for clarity, though each VM would participate. The controller determines that the red class does not use the full egress reservation and allows 1/3 of the orange traffic into the dedicated class. For a more formal discussion of the QoS controller, see Appendix 12.

This model requires cloud providers to engage with an underlying capacity provisioning problem (*i.e.,* how does one ensure that the total reserved bandwidth across all customers is actually there?). For the purpose of this API, we assume that the CP has some policy for installing and allocating bandwidth, but we do not address the specifics of the policy.

5.5 Grouping

Rationale. One noticeable advantage of today's abstractions over a purely endpoint-centric view is that virtual networks can serve as a helpful and simplifying grouping mechanism (*e.g.,* to apply an identical policy or configuration to all VMs

running a Spark job). Without a method to group hosts, reasoning at an endpoint-level may be difficult, especially since tenants are not given EIPs from a continuous address space.

We support the convenience of grouping via tags that can be associated with EIPs such that tenants may use tags in place of a list of EIPs in permit lists. This ensures cleaner semantics for the tenant while the cloud provider can "compile" these to IP addresses for filtering on packets at the end host associated with the permit list. Similar features, called service tags, are available today for Network Security Groups [48] while AWS offers general purpose tagging across resources [59]. Our use of tags is limited to EIPs as they are known across clouds while specific resources are not.

API. To provide the benefits of grouping, we adopt a method for tagging as a means of convenience for tenants in our API with `tag`. Here, tenants may associate an EIP with a given tag as shown in Table 1. Tags can be used in other APIs such as `set_permit_list` in place of addresses.

6 Implementation

To demonstrate the feasibility of our proposed API itself, we have developed an implementation using existing APIs for AWS and Azure to expose our simplified API ; *i.e.,* it effectively builds a shim layer on top of existing APIs. As discussed in earlier sections, most of our API requirements map onto existing abstractions and hence can be implemented in a straightforward manner. The key new addition is the QoS API for which we provide a partial implementation as described below. This approach – *i.e.,* building on top of existing APIs – is unfortunately deeply tied to specific cloud implementations. In the long term, we hope/expect the shim layer to thin over time as clouds provide the Invisinets API as a first-party implementation.

Implementing the QoS API requires three infrastructure components: (i) per-host enforcement modules, (ii) QoS controllers, and (iii) modifications to egress routers to appropriately classify and prioritize traffic. We built a prototype of the first two components but lack the access to realize (iii); in this sense, our implementation of the QoS API is only a partial one. At the same time, such classification and priority scheduling is standard on high-end routers and hence we anticipate no problem in realizing the API more fully in production.

For the enforcer module, the rate limiting is done through Linux `tc` operations and a bpf filter implemented as a bcc program [3]. This approach was chosen to fit easily into any host-based approach, though in a production deployment we expect this logic would reside in the hypervisor or virtual switch [25]. The bcc programs monitor outgoing traffic volumes on a per-traffic-class basis and reports to a reporting process on the host. The host sends usage to a per-tenant controller via RPCs [31]. The controller calculates per-host mappings to each traffic class and reports back to all hosts which reported in the last time interval. The host process then inserts this data into bpf tables for the rate limiter to observe. The rate limiter classifies traffic based on the tenant to cloud provider class mapping. The cloud provider classes correspond to classes in a `tc` hierarchical token bucket which rate limits the egress traffic from the node.

7 Evaluation

In our evaluation, we focus on answering two questions: (i) to what degree does Invisinets simplify a tenant's experience when networking their workloads? and, (ii) is implementing the Invisinets API technically feasible for cloud providers?

We evaluate simplicity using a two-pronged approach. First, we consider three sample tenant deployments and compare the complexity of implementing each deployment using Invisinets versus doing so with existing solutions (§7.1). This approach allows us to do a deep-dive evaluation for specific deployment scenarios and solutions. For a broader view on tenant deployments, we also analyzed 677 publicly available deployment scenarios as captured by their Terraform files on GitHub and quantify the extent to which existing network abstractions contribute to the overall setup and configuration complexity that a tenant faces, and the extent to which Invisinets can remedy this complexity (§7.2).

When considering the feasibility of implementing Invisinets, we focus primarily on scalability. This is because, as discussed earlier, cloud providers already implement (close approximations for) the individual building blocks needed to implement our Invisinets API – *e.g.,* address translators, load balancers, rate limiters. Hence, the main question is whether extending their use of these components to a larger fraction of their tenant pool will create new and problematic scaling bottlenecks. Thus, in §7.3, we evaluate scalability using a combination of system microbenchmarks and deployment statistics from cloud providers.

7.1 Evaluating Simplicity via Case Studies

Methodology. We compare the complexity of tenant networking using Invisinets versus existing solutions. For the latter we consider: (i) a DIY tenant that writes scripts directly atop existing "first-party" cloud APIs, and (ii) Aviatrix [12] as representative of third-party multi-cloud solutions.[7]

Given that there is no best practice for measuring simplicity, we propose a metric which we believe is reasonable, though we do not claim that it perfectly captures all notions of complexity. We measure simplicity in terms of the number of network components that the tenant must consider within three main categories: network boxes, links, and configuration points. Network *boxes* refers to device abstractions such as transit gateways, VPN devices, firewalls, and so forth. *Links* refers to various forms of virtual link abstractions including dedicated egress links (*e.g.,* Azure ExpressRoute [44]), vnet

[7] As mentioned earlier, Aviatrix offers tenants a management layer built atop per-cloud abstractions and optimized cloud appliances. Tenants view their multicloud deployments through a "single pane of glass", but must still be fluent in per-cloud building blocks as well as Aviatrix-specific constructs.

peerings [42], and private internal links [47]. Finally, *configuration points* counts any abstraction that exposes network configuration parameters for the tenant to consider. We count the number of configuration points (*e.g.,* a firewall) rather than lines of code or configuration (*e.g.,* number of firewall rules) since the latter can often be arbitrarily scaled in our scenarios and hence may be misleading.

We further clarify certain aspects of how we account for configuration points. First, every box is also counted as a configuration point: *e.g.,* a gateway is both a box (must be placed in a topology) and a configuration point (must be configured with routes, tunnels, *etc.*). Second, while boxes are always configuration points, the inverse is not true: *e.g.,* abstractions such as virtual networks or subnets are not boxes but do require configuration and hence are counted as configuration points. Invisinets in particular has multiple configuration points but few boxes so both measures are needed for a fair comparison. Finally, we ignore security groups/permit lists and endpoint IP addresses when counting configuration points as these are present in all solutions and depend on the number of endpoints which can be scaled arbitrarily.

We compare solutions using three case studies: one is a validated design published by Aviatrix [11] and the other two were defined by us to represent extremes of simple *vs.* sophisticated deployments.

Case Study 1: A Simple Tenant Network. We start with a rather contrived, simple deployment in which a single VM in one cloud (Cloud A) must communicate with a service in another cloud (Cloud B) in a shared address space as shown in Figure 3a. The service in Cloud B uses two VMs which are load balanced. Table 2 shows the complexity of implementing this scenario with each of the three solutions we consider.

Metric	First-Party	Aviatrix	Invisinets
Boxes	3 (VPN, LB)	3 (GW, LB)	0
Config. Point	7 (vnet, subnet, VPN)	7 (vnet, subnet, VPN)	1 (SIP)

Table 2: Complexity analysis for a Simple Tenant Network. In parentheses, we list the "top three" abstractions in terms of their contribution to complexity (see Fig. 3 for abbreviations).

We see that even this simple scenario incurs non-trivial complexity with existing solutions. With the DIY approach, tenants must still consider virtual network gateways, a load balancer, backend pools, local network gateways, route propagation parameters, and more, leading to a total complexity of 10 network components. Aviatrix is similar, though it uses 2 Aviatrix gateways in lieu of the first-party VPN gateways.

Invisinets, however, completely eliminates the virtual network layer and instead requires just one configuration point (the SIP in Cloud B), allowing the deployment to be expressed in just 9 lines of code as shown below:

```
eip1 = cloud_a.request_eip(vm1_id, name="vm1")
eip2 = cloud_b.request_eip(vm2_id, name="vm2")
eip3 = cloud_b.request_eip(vm3_id, name="vm3")
sip = cloud_b.request_sip(name="service")
bind(eip2, sip)
bind(eip3, sip)
set_permit_list(eip1, [eip2, eip3])
set_permit_list(eip2, [eip1])
set_permit_list(eip3, [eip1])
```

By contrast, our DIY script using the AWS and Azure Python APIs requires over 45 lines of code (snippets shown in Appendix 13) even assuming IPs and the underlying virtual network have already been provisioned.

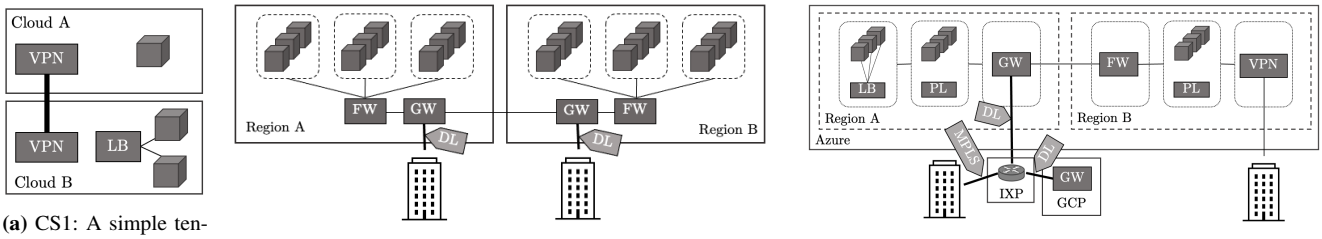
Case Study 2: (Aviatrix) Multi-Region Design. We consider a design from Aviatrix [11], as seen in Figure 3b. This deployment involves 3 virtual networks running different services in two different cloud regions. They are connected to one another via a transit virtual network which contains firewalls for security. These transit virtual networks also contain direct links to on-prem datacenters. Table 3 summarizes the complexity costs in implementing this design.

Metric	First-Party	Aviatrix	Invisinets
Boxes	4 (GW, FW)	18 (GW, FW)	2 (FW)
Links	9 (peering, DL)	2 (DL)	0
Config. Point	29 (peering, subnet, vnet)	36 (GW)	2 (egress BW)

Table 3: Complexity analysis for the Aviatrix design. See Fig. 3 for abbreviations.

We see that, compared to our first case study, complexity rises significantly with both the DIY and Aviatrix solutions. The majority of this complexity comes from the gateway and virtual network peering abstractions. Interestingly, Aviatrix incurs greater complexity than a DIY implementation due to its recommended redundant gateways, though first-party peerings cannot be redundant. In contrast, Invisinets requires only 4 network components (egress reservations to approximate the direct links and middlebox annotations for firewalls), which represents a more than 90% reduction in complexity relative to the DIY and Aviatrix solutions. Creating this deployment with the available first-party APIs would take over 45 lines of code even assuming all instances and their IPs have been provisioned and the out-of-band coordination to set up the direct links has been performed. In addition, we do not count lines only defining configuration, otherwise the script is well over 200 lines. A significant portion of this code sets up the necessary routes to get traffic from each virtual network to the appropriate firewall and/or gateway.

Case Study 3: A Heterogeneous Tenant Network. Our third scenario showcases a network deployment that involves a range of connectivity requirements, as shown in Figure 3c. This scenario is representative of virtual networks constructed by larger cloud tenants, though we chose to scale it down for understandability. For the sake of demonstration, in Figure 3c, the Azure network is depicted in some detail while the GCP deployment is unrealistically simple. In this scenario, the tenant's cloud deployments in GCP and Azure are each connected to the tenant's on-prem datacenter via direct links to an Internet exchange point where the tenant has reserved a virtual router and an MPLS link to their datacenter. In the Azure deployment, the ExpressRoute terminates at a VPN gateway which must reside in its own subnet [49]. From there, user-defined routes send traffic to the appropriate subnet or



(a) CS1: A simple tenant network.

(b) CS2: Aviatrix multi-region design [11].

(c) CS3: A heterogeneous tenant network.

Figure 3: Generalized (not taking the form of any particular current approach) network topologies for each case study. (LB = load balancer, GW = gateway, FW = firewall, PL = private link, DL = Direct Link)

onto the ExpressRoute for egress traffic to GCP or On-Prem. The other subnets in the virtual network contain VMs and a private link to an Azure service such as storage on the left and load-balanced VMs on the right. In another virtual network in Azure, a VPN gateway tunnels to a branch office. Ingress traffic is directed to a subnet containing a firewall which then allows traffic to reach VMs or another private link to a Microsoft service. Table 4 summarizes the complexity costs in this design. A script to create this deployment (again, ignoring instances, IPs, and some ExpressRoute setup) would require over 80 lines of code. While coming in the form of many different components, achieving basic connectivity via gateways, links, *etc.* continues to be the main source of complexity.

Metric	First-Party	Aviatrix	Invisinets
Boxes	6 (GW, FW, LB)	6 (GW, FW, LB)	2 (FW)
Links	5 (DL, PL, MPLS)	5 (DL, PL, MPLS)	0
Config. Point	22 (vnet, subnet, GW)	22 (vnet, subnet, GW)	4 (egress BW, FW)

Table 4: Complexity analysis for the heterogenous tenant network. See Fig. 3 for abbreviations.

7.2 Terraform Complexity

For a broader perspective, we now consider the impact of Invisinets on a larger pool of deployment scenarios. Specifically, we evaluate the scope for simplification in the Terraform files that a network administrator maintains. Terraform [32] is a popular language for specifying cloud infrastructure as declarative code directly using the first-party abstractions.

Methodology. We scraped GitHub for Terraform files creating virtual networks, selecting files that mention an AWS VPC and omitting files that only defined Terraform variables or outputs. For a scrape conducted on 9/15/2022, our filtering yielded 677 files for analysis.[8] We use these configurations since they are publicly available but recognize that they may not fully represent the tenant use-cases that Invisinets targets (*e.g.,* mid/large scale enterprises). We believe our analysis could be easily repeated on large production deployments.

Results. We parse each Terraform file, identifying whether a code block corresponds to a virtual network component. Table 5 lists the components we identified and how often they occurred in our files.[9] We also calculate the lines of code within each of these code blocks and show the total per-component line count in the last column of Table 5. Unsurprisingly, VPCs are the most common component. Beyond this, we see that abstractions used to establish basic connectivity – *e.g.,* Internet Gateway, Subnet, Route Table, Route – constitute a significant fraction of the networking abstractions that an administrator must deal with. Moreover, even complex routing abstractions such as Transit Gateways are not uncommon. These findings thus support our thesis that an approach such as Invisinets, which altogether eliminates the need for virtual topologies and their routing configurations, can substantially simplify networking configurations.

Virtual Network Component	Occurrence Count	Line Count
VPC	2,493	26,731
Route Table	1,839	13,317
Subnet	1,514	14,677
Security Group	456	9,704
Internet Gateway	445	2,802
Route	339	2,184
NAT Gateway	209	1,661
Network ACL	141	2,841
Transit Gateway	82	587
VPN Gateway	40	316
Load Balancer	22	207
Network Interface	16	123
VPN Peering Connections	5	27
Customer Gateway	4	58
VPN Route	2	10
VPN Connections	1	13

Table 5: Breakdown of Terraform lines removed (across all 677 scraped files) by virtual network component.

The lines of code identified in Table 5 can be viewed as an upper bound on the lines of code that can be omitted by using Invisinets. To evaluate whether this is a significant portion of the overall Terraform configuration, Figure 4 shows a histogram of the percentage of lines-of-code that can be omitted from the Terraform files we consider.

While the fraction of omitted configuration lines may be surprising, these resource definitions rarely carry information critical to the four tenant goals around which the Invisinets API is designed. Instead, many of these lines specify details of unnecessary abstractions such as virtual networks and the gateways required to reach external endpoints. (We show an

[8] Github API search results are limited to the first 1000 results. Additional filtering on returned files reduced the number further.

[9] We believe this estimate is approximately equivalent to the set of network components (boxes, links, and configuration points) that we measure in §7.1.

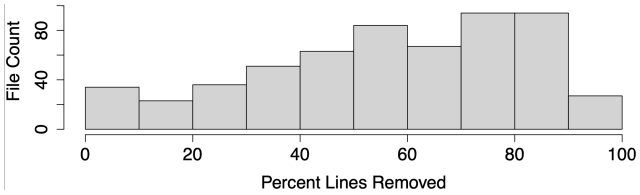


Figure 4: Histogram of percent lines of Terraform resource files that can be omitted with Invisinets API.

example of such a configuration in Appendix 14). Accordingly, the streamlined Invisinets API does not require much of this configuration.

We note that our estimates for omitted code in this section are an upper bound since we do not account for the lines of configuration that would be *added* by using Invisinets's API. Nonetheless, our evaluation in the previous section suggests that Invisinets requires far fewer (90+% fewer) components to configure and hence we expect to be left with significant savings. Moreover, the permit lists that Invisinets requires contain information that is necessary even without Invisinets; *e.g.,* in the form of VPN access lists, firewall rules, NSG configurations, and so forth.

7.3 Scalability

As mentioned earlier, we focus on the scalability of the Invisinets API. Implementing the Invisinets API implies change along two main fronts: address management to support PRDO-only addressing, and enforcing per-tenant egress bandwidth reservations. We consider each in turn.

PRDO addressing. PRDO-only addressing changes the number of endpoints that are assigned public IP addresses and hence a natural question is whether cloud providers have a large enough public IP addresses to give one to every VM. From discussions with Azure operators, we learnt that Azure advertises over $1.4 \cdot 10^{20}$ addresses in total [46] and hosts O(10M) VMs. Thus we can safely conclude that Azure has sufficient public addresses to support PRDO addressing. Similarly, GCP has over $7.1 \cdot 10^{26}$ advertised addresses in total [27], presumably plenty to allocate to all endpoints.

The next scalability concern may be the impact on the address translation infrastructure – specifically, the vswitch/NIC and SLBs used to translate PRDO addresses to internal private addresses and to enforce permit lists. Fortunately, the PRDO-only infrastructure does not increase the number or complexity of vswitch/NIC lookup operations, as every packet is translated to DIPs even today. Similarly, in considering the impact on the SLB, we note that Invisinets does not change the number of internal endpoints that need to be reachable from endpoints outside the cloud provider (defined as semi-private addresses in §3) since this depends on the tenant's workload requirements rather than the addressing architecture. Today, the bindings and permit-list rules for these endpoints are programmed in per-tenant VPNs/NATs while Invisinets implements the same in the cloud provider's SLB. Since cloud SLBs [21, 54] are already designed for elastic horizontal scaling, we do not anticipate any scaling challenges due to semi-private addresses.

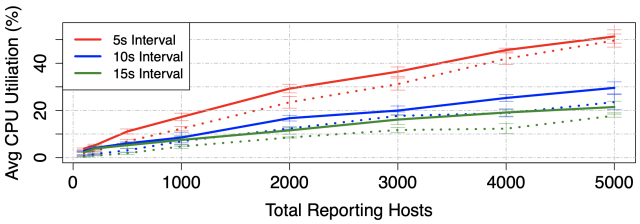


Figure 5: CPU utilization of QoS controller for different reporting intervals over 5 experiments. The report collection (solid) is the bottleneck over aggregation and reporting (dotted).

QoS Enforcer. Invisinets adds per-tenant, per-region QoS controllers to a cloud provider's infrastructure. We now benchmark the QoS components of our implementation to demonstrate that these QoS controllers could reasonably scale in a production setting. Our implementation consumes additional network bandwidth for communication between the QoS controller and enforcer modules. (The volume of tenant traffic remains unchanged with only ToS fields modified to reflect the class of traffic.) Using 32 traffic classes (chosen to be expressive), we record a worst-case overhead of only 800 bytes per host in one reporting interval for one reservation. For a VM with a 5 Gbps link, communication with the QoS controller consumes less than 0.00001% of its link bandwidth when reporting at 10 second intervals. At the QoS controller, this overhead is $n \times 800$ bytes per interval where n is the number of VMs per tenant per region. Discussions with a major cloud provider $CP-X$ revealed that, for their deployments, n is under 50,000 per region in the worst case, and $O(10)$ on average. Hence, for a 10 second reporting interval, the total bandwidth consumed at the QoS controller is around 32Mbps in the worst-case and far lower for average tenant sizes.

Next, we measure the CPU utilization at our QoS controller for increasing numbers of reporting hosts, shown in Figure 5. Our QoS controller is comprised of two main processes: a gRPC [31] server to collect bandwidth reports and an aggregation process to calculate the mapping and report back to hosts. We run our QoS controller on a t3.small instance in AWS with 2 vcpus, 2 GB of memory, and 5 Gbps bandwidth, chosen to ensure that network capacity is not a bottleneck. One hundred t2.small instances were used as hosts, with many host processes per instance. As shown in the figure, stressing our 2vCPU instance requires 1000s of reporting hosts. We believe this is a reasonable expectation for region-scale deployments since the average number of VMs per tenant is $O(10)$. Depending on the interval, with enough hosts, the collection process cannot keep up with incoming reports at which point the controller size (*i.e.,* vcpus) can be scaled as necessary. Scaling up to support a tenant with potentially millions of hosts in one region would pose an additional challenge in aggregating bandwidth measurements; we leave that challenge to future work.

Finally, we consider the total resource consumption due

to QoS controllers (across all tenants). In Azure, there are roughly an average of 16,000 tenants in each region. Accordingly, we expect the QoS enforcer to consume fewer than 1,600 cores in each region on average (since the total utilization for 100 hosts and 5 s intervals is <10%).

Despite the additional packet-processing logic due to the enforcer, no significant overheads are seen in processing host traffic. We measured the latency and throughput of outbound traffic from a host with the QoS enforcer enabled and disabled. Neither metric demonstrated a noticeable overhead. Throughput measurements using iperf [1] between two t2.medium instances for 10 second flows produced an average of 64.9 Mbps across 100 flows with the rate limiting (though set to allow many Gbps to not interfere) and 64.8 Mbps without. Latency was tested with 200 pings to a remote host. With the rate limiter enabled, the average was 0.46 ms ($\sigma = 0.12$) while the average was 0.48 ms ($\sigma = 0.258$) without it.

Thus we conclude that Invisinets's QoS infrastructure introduces little overhead for cloud providers.

8 Limitations

We acknowledge that Invisinets may not satisfy infrastructure requirements for all cloud tenants. Despite the layers of protection from the cloud provider, the tenant may deem public IPs to be an unacceptable risk for their endpoints. While addresses in Invisinets are default-off, this choice removes the layer of isolation provided by virtual network private address spaces. We note, however, that default-off semantics are present in both Invisinets and today's virtual network abstractions. Only the placement of these parameters differ (*e.g.,* endpoint permit list vs. gateway rules). In addition, the workload may have security requirements defined in terms of today's networking abstractions. Accordingly, we do not expect this interface to appeal to all tenants and target specifically cloud-native applications.

As mentioned in 5.4, our QoS API's ability to simulate the effects of a direct link is dependent on the level of connectivity between the clouds. This assumption may be particularly precarious in areas with smaller cloud footprints.

Since our primary goal with the Invisinets API was to simplify the tenant interface, our evaluation is inherently limited. Complexity is a multifaceted issue and cannot be completely captured in any one metric. We proposed a variety of metrics to evaluate simplicity as fairly as we could, though the values are not necessarily exact.

9 Related Work

We build on the extensive literature on cloud and network virtualization [20, 24, 53, 54, 60]. As mentioned in §5, most of our API leaves this underlying architecture unchanged or extends it in relatively straightforward ways.

Distributed rate limiting has also been extensively studied [37, 39, 56, 58, 64]; *e.g.,* with work on optimizing end host traffic shaping [58, 64]. Systems such as [39, 56] make network-wide rate limiting decisions similarly to the model we adopt, though [56] primarily seeks to limit traffic aggregates rather than provide a minimum guarantee. Such guarantees are provided by [40] with strategic resource placement while [37] modifies the end host networking stack.

We take the large body of work in network verification [14, 23, 26, 36, 38] as evidence of the overwhelming management complexity in networking. While verification has focused more on datacenter environments, virtual networks suffer similar complexity since they use similar abstractions. Generally, compiling high-level network intents into low-level device configs remains difficult and error prone [15].

This paper joins others in advocating a forward-looking view of cloud networking though prior proposals focused on performance guarantees [13, 50, 51] or declarative control [16] rather than the simplification we target. In [4], a platform to unify cloud services is developed by creating a substrate infrastructure across clouds. The vision in [61] takes the extreme stance of proposing a unified interface for all clouds. Invisinets can be viewed as an instantiation of their vision for networking, though we focus less on unification and instead hope that simpler APIs across clouds will result in some homogenization over today's highly-siloed APIs.

10 Conclusion

To simplify networking for cloud tenants, we proposed a declarative and endpoint-centric API which takes L3 connectivity as a given and removes the burden of deep networking knowledge from tenant operators. We acknowledge that our model may not initially meet the requirements of all tenants. However, our proposed API can coexist with existing abstractions and, in fact, can provide a spectrum of simplicity where deployments may include both today's building blocks as well as our proposed architecture. Our API requires consideration of fewer network components and obviates the need for network topologies to be constructed at all. Supporting this simple API requires minimal infrastructural changes to cloud datacenters and the new systems should be easily scaled. We believe the Invisinets API for virtual networking follows the evolution seen in cloud compute and storage from virtual replicas of physical components to higher-level services.

References

[1] iperf. https://iperf.fr/.

[2] tcpping. http://www.vdberg.org/~richard/tcpping.html.

[3] BPF Compiler Collection (BCC), 2022. https://github.com/iovisor/bcc.

[4] M. Alaluna, E. Vial, N. Neves, and F. M. V. Ramos. Secure and dependable multi-cloud network virtualization. In *Proceedings of the 1st International Workshop on*

Security and Dependability of Multi-Domain Infrastructures, XDOMO'17, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Alkira. Alkira, 2022. https://www.alkira.com/.

[6] Amazon Web Services. Amazon api gateway, 2022. https://aws.amazon.com/api-gateway/.

[7] Amazon Web Services. Aws direct connect, 2022. https://aws.amazon.com/directconnect/.

[8] Amazon Web Services. Connect your vpc to other networks, 2022. https://docs.aws.amazon.com/vpc/latest/userguide/extend-intro.html.

[9] Amazon Web Services. Vpcs and subnets, 2022. https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html.

[10] Aurea. Cloudfix, 2022. https://cloudfix.com/.

[11] Aviatrix. Aviatrix multi-region high-availability cloud network design. https://aviatrix.com/aviatrix-multi-region-high-availability-cloud-network-design/.

[12] Aviatrix. Aviatrix, 2022. https://aviatrix.com/.

[13] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 242–253, New York, NY, USA, 2011. Association for Computing Machinery.

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[15] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations: Brief reflections on abstractions for network programming. *SIGCOMM Comput. Commun. Rev.*, 49(5):104–106, nov 2019.

[16] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: A cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[17] Cerba. Densify, 2022. https://www.densify.com/.

[18] CloudBolt Software. Cloudbolt cost security management platform, 2021. https://www.cloudbolt.io/kumolus/.

[19] CloudZero. Cloudzero, 2022. https://www.cloudzero.com/.

[20] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, Apr. 2018. USENIX Association.

[21] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, Mar. 2016. USENIX Association.

[22] J. L. Eric Brewer. Application modernization and the decoupling of infrastructure services and teams, 2019. https://services.google.com/fh/files/blogs/anthos_white_paper.pdf.

[23] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 217–232, Savannah, GA, Nov. 2016. USENIX Association.

[24] A. D. Ferguson, S. Gribble, C.-Y. Hong, C. Killian, W. Mohsin, H. Muehe, J. Ong, L. Poutievski, A. Singh, L. Vicisano, R. Alimi, S. S. Chen, M. Conley, S. Mandal, K. Nagaraj, K. N. Bollineni, A. Sabaa, S. Zhang, M. Zhu, and A. Vahdat. Orion: Google's Software-Defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98. USENIX Association, Apr. 2021.

[25] D. Firestone. Vfp: A virtual switch platform for host sdn in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2017*, March 2017.

[26] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, May 2015. USENIX Association.

[27] Google. cloud.json, 2022. https://www.gstatic.com/ipranges/cloud.json.

[28] Google. Dedicated interconnect overview, 2022. https://cloud.google.com/network-connectivity/docs/interconnect/concepts/dedicated-overview.

[29] Google. Load balancer overview, 2022. https://cloud.google.com/anthos/clusters/docs/multi-cloud/aws/how-to/load-balancers.

[30] Google. Secure and encrypted communication between anthos clusters using anthos service mesh, 2022. https://cloud.google.com/architecture/encrypt-secure-communication-between-multiple-anthos-clusters-concept.

[31] gRPC Authors. gRPC, 2022. https://grpc.io/.

[32] Hashicorp. Terraform, 2022. https://www.terraform.io/.

[33] Istio Authors. Vpn connectivity, 2019. https://istio.io/v1.1/docs/setup/kubernetes/install/multicluster/vpn/.

[34] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[35] S. Janardhan. Update about the october 4th outage, oct 2021. https://engineering.fb.com/2021/10/04/networking-traffic/outage/.

[36] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[37] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. EyeQ: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, Lombard, IL, Apr. 2013. USENIX Association.

[38] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, Apr. 2012. USENIX Association.

[39] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 1–14, New York, NY, USA, 2015. Association for Computing Machinery.

[40] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 467–478, New York, NY, USA, 2014. Association for Computing Machinery.

[41] S. McClure, S. Ratnasamy, D. Bansal, and J. Padhye. Rethinking networking abstractions for cloud tenants. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 41–48, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Microsoft. Virtual network peering. https://learn.microsoft.com/en-us/azure/virtual-network/virtual-network-peering-overview, year = 2022, lastaccessed = March 2, 2023.

[43] Microsoft. Overview of load-balancing options in azure, 2021. https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/load-balancing-overview.

[44] Microsoft. What is azure expressroute?, 2021. https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction.

[45] Microsoft. Azure ddos protection standard overview, 2022. https://docs.microsoft.com/en-us/azure/ddos-protection/ddos-protection-overview.

[46] Microsoft. Azure ip ranges and service tags – public cloud, 2022. https://www.microsoft.com/en-us/download/details.aspx?id=56519.

[47] Microsoft. Private link, 2022. https://azure.microsoft.com/en-us/products/private-link/.

[48] Microsoft. Virtual network service tags, 2022. https://docs.microsoft.com/en-us/azure/virtual-network/service-tags-overview.

[49] Microsoft. What is vpn gateway?, 2022. https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways.

[50] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *SIGCOMM Comput. Commun. Rev.*, 42(5):44–48, sep 2012.

[51] J. C. Mogul and J. Wilkes. Nines are not enough: Meaningful metrics for clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 136–141, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Palo Alto Networks. Palo alto networks, 2022. https://www.paloaltonetworks.com/.

[53] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[54] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. 43(4):207–218, aug 2013.

[55] PCI Security Standards Council. Pci security standards council, 2022. https://www.pcisecuritystandards.org/.

[56] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 337–348, New York, NY, USA, 2007. Association for Computing Machinery.

[57] D. Ramel. Research brief summarizes trends in multicloud deployments, October 2019. https://virtualizationreview.com/articles/2019/10/21/cloud-trends.aspx.

[58] A. Saeed, N. Dukkipati, V. Valancius, T. Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end-hosts. In *ACM SIGCOMM 2017*, 2017.

[59] A. W. Services. Tagging aws resources, 2022. https://docs.aws.amazon.com/general/latest/gr/aws_tagging.html.

[60] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *Commun. ACM*, 59(9):88–97, aug 2016.

[61] I. Stoica and S. Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Strickx, Tom and Hartman, Jeremy. Cloudflare outage on june 21, 2022, jun 2022. https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/.

[63] The Kubernetes Authors. Ingress controllers, 2021. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

[64] K. To, D. Firestone, G. Varghese, and J. Padhye. Measurement based fair queuing for allocating bandwidth to virtual machines. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, page 14–19, New York, NY, USA, 2016. Association for Computing Machinery.

[65] U.S. Department of Health and Human Services Office for Civil Rights. Hipaa administrative simplification, March 2013.

[66] Virtana. Virtana, 2022. https://www.virtana.com/.

[67] VMWare. Multi cloud operations: Visibility & control, 2022. https://www.vmware.com/cloud-solutions/multi-cloud-ops.html.

[68] Volterra. Volterra, 2022. https://www.volterra.io/.

Appendix

11 Benefits of dedicated links.

Further, we conducted a small experiment to determine the benefit of these dedicated links. We provisioned an ExpressRoute from Azure in Northern California to a colocation facility in Silicon Valley and connected it to Direct Connect to AWS in Northern California. A diagram is shown in Figure 6. Both dedicated links were 50Mpbs and the virtual router in the colocation facility can handle up to 500Mbps. In parallel to this dedicated connection between clouds, we connected two hosts in each deployment via the public Internet. We collected throughput measurements using iperf [1] every 5 minutes for a week and performed tcppings [2] every minute. We found that at these low bandwidths, the primary performance benefit is consistent throughput (see summary in Table 6). Notably, the latency can even be worse across these dedicated links, though not significantly so considering the variability.

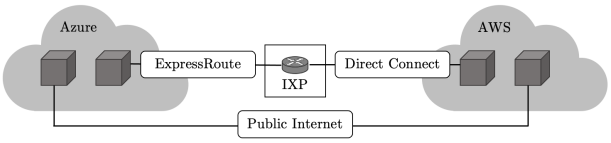


Figure 6: Direct link measurement setup.

Client Cloud	Direct Link?	Measurement	Mean	Std. Dev.
AWS	Yes	Throughput	51 Mbps	0.23 Mbps
AWS	No	Throughput	169 Mbps	85 Mbps
Azure	Yes	Throughput	50 Mbps	1.4 Mbps
Azure	No	Throughput	117 Mbps	80 Mbps
AWS	Yes	Latency	4.89 ms	1.74 ms
AWS	No	Latency	3.38 ms	1.73 ms
Azure	Yes	Latency	12.53 ms	9.36 ms
Azure	No	Latency	7.45 ms	4.39 ms

Table 6: Summary of direct link measurements.

12 QoS Controller

More formally, the job of a QoS controller is as follows. Each tenant, t, can have n classes of traffic, decreasing in priority: $C_1, C_2 \ldots$ Traffic in these classes must be mapped to the cloud provider best-effort and dedicated classes B and D. The tenant reserves some dedicated egress bandwidth, r_t. To map the classes, each host x reports x_i^t, the bandwidth tenant t consumed in class C_i every interval of k seconds to the per-tenant controller. Every interval, the controller calculates the total bandwidth for each class, C_i. Starting with the highest priority class, C_1, the controller adds x_i^t to the running total dedicated bandwidth, x_D, and maps the class to the dedicated cloud provider class, D. When $x_D > r_t$, the controller maps the current class C_s to a split class, S. All subsequent classes $C_{j>s}$ are mapped to B. The fraction of the bandwidth used by C_i which would fit into the reserved bandwidth $f_s = (x_D - r_t)/C_s$ is calculated as well. The controller sends f_s and C_s to all hosts that reported in the previous interval.

The (enforcement module at) hosts then install the calculated mapping between tenant and cloud provider classes. Traffic belonging to $C_i < C_s$ is simply marked as reserved (D) while traffic in $C_i > C_s$ is marked as best-effort (B). For traffic belonging to C_s, the host calculates the maximum allowed dedicated bandwidth in the split class $b_d = f_s \cdot x_i^t$. This value is used for per-flow admission to D for traffic in C_s. When a new flow in C_s arrives, a timestamp is recorded. After m time has passed, the flow is eligible for promotion to D. Its bandwidth over the last m seconds, b_{flow}, is compared against b_d. If $b_{flow} < b_d$ the flow is mapped to D and b_d is updated ($b_d = b_d - b_{flow}$), otherwise the flow is mapped to B. This evaluation is performed every m seconds for each flow in C_s which has not been admitted to D. Accordingly, this process requires per-flow state, though only proportional to the number of flows in C_s during a single reporting interval.

13 Case Study 1 Code

In Figure 7 is an excerpt of the code required to implement Case Study 1 using first-party cloud APIs.

```
def setup():
    vnet = network_client.virtual_networks.get(RESOURCE_GROUP, vnet_name)
    eip = network_client.public_ip_addresses.get(RESOURCE_GROUP, EIP1_NAME)
    eip1_config = network_client.\
        network_interface_ip_configurations.get(RESOURCE_GROUP,
                                                get_nic_name_from_ipconf_id(
                                                    eip.ip_configuration.id),
                                                get_resource_name_from_id(
                                                    eip.ip_configuration.id))

    eip = network_client.public_ip_addresses.get(RESOURCE_GROUP, EIP2_NAME)
    eip2_config = network_client.\
        network_interface_ip_configurations.get(RESOURCE_GROUP,
                                                get_nic_name_from_ipconf_id(
                                                    eip.ip_configuration.id),
                                                get_resource_name_from_id(
                                                    eip.ip_configuration.id))

    backend_address1 = \
        LoadBalancerBackendAddress(name="endpoint1-backend",
                                   virtual_network=SubResource(id=vnet.id),
                                   ip_address=eip1_config.private_ip_address)
    backend_address2 = \
        LoadBalancerBackendAddress(name="endpoint2-backend",
                                   virtual_network=SubResource(id=vnet.id),
                                   ip_address=eip2_config.private_ip_address)

    params = PublicIPAddress(location=LOCATION,
                             sku=PublicIPAddressSku(name="Standard"),
                             public_ip_allocation_method="Static",
                             public_ip_address_version="IPv4")
    poller = network_client.\
        public_ip_addresses.begin_create_or_update(RESOURCE_GROUP,
                                                   "lb-ip",
                                                   params)
    wait_for_complete(poller)
    lb_ip = poller.result()
    ip_config = FrontendIPConfiguration(name="lb-ipfrontend",
                                        public_ip_address=lb_ip)

    # Set up probes and backend pools
    probe = Probe(name="lb_probe", protocol="Tcp", port=80,
                  interval_in_seconds=5, number_of_probes=2)
    backend_pool = BackendAddressPool(name="lb-pool-1",
        load_balancer_backend_addresses=\
        [backend_address1, backend_address2])

    # Create a new LB
    # ...
```

Figure 7: Excerpt of the code necessary to setup the Azure side of Case Study 1.

14 Terraform Example

Below is a code snippet from one of our scraped Terraform files.

```
resource "aws_subnet" "mgmt_subnet2" {
  vpc_id            = aws_vpc.vpc_sec.id
  cidr_block        =
  var.security_vpc_mgmt_subnet_cidr2
  availability_zone = var.availability_zone2
  tags = {
    Name = "$mgmt-subnet2"
  }
}

# Routes
resource "aws_route_table" "data_rt" {
  vpc_id = aws_vpc.vpc_sec.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw_sec.id
  }
  route {
    cidr_block         = var.spoke_vpc1_cidr
    transit_gateway_id =
    aws_ec2_transit_gateway.TGW-XAZ.id
  }
  route {
    cidr_block         = var.spoke_vpc2_cidr
    transit_gateway_id =
    aws_ec2_transit_gateway.TGW-XAZ.id
  }
  tags = {
    Name = "$data-and-mgmt-rt"
  }
}
```

Figure 8: Code snippet from a scraped Terraform file.

Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs

John Thorpe[†♣] Pengzhan Zhao[†♣] Jonathan Eyolfson[†] Yifan Qiao[†] Zhihao Jia[‡]
Minjia Zhang[§] Ravi Netravali[*] Guoqing Harry Xu[†]
UCLA[†] *CMU*[‡] *Microsoft Research*[§] *Princeton University*[*]

Abstract

DNN models across many domains continue to grow in size, resulting in high resource requirements for effective training, and unpalatable (and often unaffordable) costs for organizations and research labs across scales. This paper aims to significantly reduce training costs with effective use of preemptible instances, i.e., those that can be obtained at a much cheaper price while idle, but may be preempted whenever requested by priority users. Doing so, however, requires new forms of *resiliency* and *efficiency* to cope with the possibility of frequent preemptions – a failure model that is drastically different from the occasional failures in normal cluster settings that existing checkpointing techniques target.

We present Bamboo, a distributed system that tackles these challenges by introducing *redundant computations* into the training pipeline, *i.e.*, whereby one node performs computations over not only its own layers but also over some layers in its neighbor. Our key insight is that training large models often requires *pipeline parallelism* where "pipeline bubbles" naturally exist. Bamboo carefully fills redundant computations into these bubbles, providing resilience at a low cost. Across a variety of widely used DNN models, Bamboo outperforms traditional checkpointing by **3.7**× in training throughput, and reduces costs by **2.4**× compared to a setting where on-demand instances are used.

1 Introduction

DNNs are becoming progressively larger to deliver improved predictive performance across a variety of tasks, including computer vision and natural language processing. For instance, recent language models such as BERT [66] and GPT [50] already have a massive number of parameters, and their newer variants continue to grow at a rapid pace. For example, BERT-large has 340 million parameters, GPT-2 has 1.5 billion, and GPT-3 increases to 175 billion; the next generation of models embed upwards of trillions of parameters [17].

Of course, model growth also entails larger training costs. For instance, GPT-3 consumes several thousand petaflop/s-days, costing over $12 million to train on a public cloud (needing hundreds of GPU servers) [6]. Unfortunately, such costs are prohibitive for small organizations. Even for large tech firms, training today's models incurs an exceedingly high monetary cost that eventually gets billed to the training department. While pretrained models may be reused and fine-tuned for different applications, training new models is often required to keep pace with changing or emerging workloads and datasets.

Although there exists a body of work on improving the training of large models [38, 39, 26, 9, 7, 11, 12, 18, 54, 53, 64, 72, 24, 28, 31], existing techniques focus primarily on *scalability* and *efficiency*, with monetary costs often being neglected. However, when *affordability* and *accessibility* are considered, resource usage becomes a key concern and none of these techniques were targeted at improving *cost-efficiency* (*e.g.*, performance-per-dollar) for training.

Preemptible Instances. This paper explores the possibility of using preemptible instances—a popular class of cheap cloud resources—to reduce the cost of training large models. There are several kinds of preemptible instances. For example, major public clouds provide *spot instances* with a price much cheaper than *on-demand instances*—*e.g.*, the hourly rate of a GPU-based spot instance is only ∼30% of that for its on-demand counterpart on Amazon EC2 [3]. As another example, large datacenters often maintain certain amounts of compute resources that can be allocated for any non-urgent tasks but will be preempted as urgent tasks arise [41, 5]. Similarly, recent ML systems [27, 69, 4] allow training jobs to use inference-dedicated machines to fully utilize GPU resources but preempts those machines when high-priority inference jobs arrive. The presentation of this paper focuses on spot instances, but we note that our techniques are generally applicable to any type of preemptible resources.

Despite their substantial cost benefits, preemptible instances pose major challenges in reliability and efficiency due the frequent and unpredictable nature of their preemptions. When and how many instances get preempted depends primarily on the number of priority jobs/users in a cluster. In a public spot market, preemption can also result from the market price exceeding the user's bid price. While price-based preemption can be avoided via a high bid price (*e.g.*, the on-demand price), capacity-based preemption is unavoidable. Preemption patterns vary drastically across clouds and even across families/zones on the same cloud (§3).

Given the unpredictable nature of spot instances, users can often only run short, stateless jobs and simply restart these jobs if they get preempted. Model training, on the contrary, is stateful and time-consuming. Discarding the state (*e.g.*, learned weights) upon each instance preemption not only

♣ Contributed equally.

wastes computation but also prevents training from making progress. Checkpointing-based techniques can reduce wasted computation to a degree, but still spend a significant fraction of the training time (*e.g.*, **77%** when training GPT-2 with 64 EC2 spot instances, see §3) on restarting and redoing prior work in the presence of frequent preemptions [20, 21]—a largely different scenario compared to conventional clusters where failures are rare.

Bamboo. This paper presents Bamboo, a distributed system that provides resilience and efficiency for DNN training over preemptible instances. Bamboo supports both pipeline parallelism and (pure) data parallelism with the same approach. Since pipeline parallelism is a more complex and general approach (for training large models), our discussion focuses on pipeline parallelism; we briefly discuss our support for pure data parallelism in §B. Bamboo currently does *not* support model parallelism.

Redundant Computation. Key to the success of Bamboo is a set of novel techniques centered around *redundant computation* (RC), inspired by how disk redundancies such as RAID [45] provide resilience in the presence of disk failures. A training system that uses pipeline parallelism runs a set of data-parallel pipelines, each training on a partition of the dataset. Each node[1] in a data-parallel pipeline performs (forward and backward) computations over a shard of NN layers with a *microbatch* of data items [24]. Bamboo lets each node in each data-parallel pipeline carry its own shard of layers as well as its successor's shard. Each node performs *normal* computation over its own layers and *redundant* computation over its successor's layers. The reason why we use a neighbor node (as opposed to a random node) to run RC is to exploit data locality in pipeline parallelism (see §5). Upon a node preemption, its predecessor has all the information (*e.g.*, layers, activations) needed for the training to progress; continuing training requires running a failover schedule on the predecessor node without wasting prior computations.

At first glance, running RC on every node appears infeasible due to concerns with both time and memory. Bamboo overcomes these challenges by taking into account pipeline characteristics to carefully reduce/hide these overheads.

First, to minimize the time overhead from RC, Bamboo leverages a key insight that *bubbles* [24, 51] inherently exist in systems using synchronous pipeline parallelism (§2). Bubbles are idle times on each node due to the gaps between the forward and backward processing of microbatches (Figure 1). Bamboo schedules the forward redundant computation (FRC) on each node asynchronously into the bubble. FRC entails a node doing the forward pass over its successor's layers using the output of its own active layers and is the main way Bamboo achieves redundancy. For the part of FRC that cannot fit into the bubble, Bamboo overlaps it with the normal computation. As a result, FRC incurs a tolerable overhead (*i.e.*, no extra communication is needed due to locality, and it can overlap with normal computation), and hence Bamboo performs it *eagerly* in each epoch. If a node is preempted during a forward pass, the pipeline continues after a node rerouting step whose overhead is negligible.

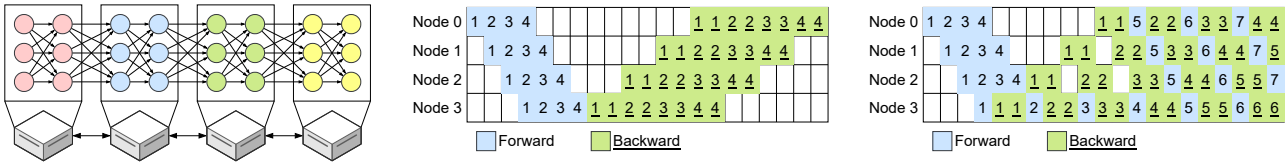
In addition to FRC, the system must find a way to generate the redundant version of the intermediate data related to backwards passes for the successor node. This can be accomplished by using the backward redundant computation (BRC), or a backwards pass over the node's redundant layers (its successor's layers). Unfortunately, for BRC, ~~such~~ a corresponding bubble does not exist. Eager BRC would require much extra work and data-dense communication on the critical path, which could delay training significantly (§5). As such, Bamboo runs BRC *lazily* only when a preemption actually occurs. If a node is preempted in a backward pass, continuing the pipeline requires a pause for the node's predecessor to perform BRC to restore the lost state. However, since FRC is performed eagerly, when BRC runs, much of what it needs is already in memory, keeping pauses short.

Second, performing RC increases each node's GPU memory usage. Note that the major source of the memory overhead is storing intermediate results (activations and optimizer state) from FRC, *not* the redundant layers, which take only little extra memory. To mitigate the memory issue, we leverage Bamboo's unique way of performing RC described above. Note that the purpose of saving intermediate results of a forward pass is that these results are used by its backward computation. However, in Bamboo, BRC is performed lazily upon preemptions and the intermediate results of FRC are thus not needed in normal backward passes. Hence, Bamboo swaps out the intermediate results of each node's FRC into the node's CPU memory, leading to substantial reduction in GPU memory usage. These results are swapped back into GPU memory for BRC only upon preemptions.

Bamboo continues normal training with the help of RC in the presence of non-consecutive preemptions, *i.e.*, preempted instances are not neighbors in the same data-parallel pipeline. Once consecutive instances are preempted, RC can no longer provide resilience. More redundancies could be added to provide stronger resilience, but this would incur (compute and communication) overheads that are too significant to hide. Instead, based on our empirical observation that most concurrent preemptions come from the same allocation group (*e.g.*, a zone), Bamboo takes care to ensure that consecutive nodes in each pipeline come from different zones, minimizing the chance of consecutive preemptions at a small ($<$5%) overhead (see §6.5).

Results. We built Bamboo atop DeepSpeed [51] and evaluated it by training 6 representative DNN models using EC2 spot clusters comprised of p3 instances. Compared to a baseline using on-demand instances, Bamboo delivers a **3.6**$\times$ cost reduction. Bamboo also outperforms a checkpointing approach by **3.7**$\times$. We developed a simulation

[1] In this paper, "instance" and "node" both refer to a spot instance.



(a) Pipeline parallelism (b) GPipe scheduling (c) PipeDream scheduling

Figure 1: Illustration of pipeline parallelism on a 4-node cluster: (a) the model is divided into 4 shards, each with 2 layers; (b) and (c) show the scheduling of two recent systems GPipe [24] and PipeDream [38].

framework that takes preemption traces from real spot clusters and training parameters to simulate how training progresses with larger numbers of nodes. A deep-dive with BERT across a wide range of preemption probabilities shows that the *value* (*i.e.*, performance-per-dollar) Bamboo provides stays constant and is much higher (**2.48**×) than that of on-demand instances. Bamboo is publicly available at https://github.com/uclasystem/bamboo.

2 Background

This section discusses necessary background for parallelism strategies. *Data parallelism* keeps a replica of an entire DNN on each device, which processes a subset of training samples and iteratively synchronizes model parameters with other devices. This strategy is often used with models that can fit entirely within a single GPU and used to both increase throughput or expand to batch sizes that cannot fit within a single GPU. Data parallelism can be combined with pipeline and/or model parallelism to train large models that do not fit on a single device. *Model parallelism* [13] partitions model operators across training devices. For example, the weights for a single matrix multiplication may reside across two separate GPUs, each performing a part of the full computation and then combining the results. This technique allows the model to expand beyond a single GPU by reducing the memory requirements of each operator. However, efficient model parallelism algorithms are extremely hard to design, requiring difficult choices among scaling capacity, flexibility, and training efficiency. As such, model-parallel algorithms are often architecture- and task-specific.

Pipeline parallelism [38, 24, 71] has gained much traction recently due to its flexibility and applicability to a variety of neural networks. Pipeline parallelism divides a model at the granularity of layers and assigns a shard of layers to each device. Figure 1(a) shows an example where the model is partitioned into four shards and each worker hosts one shard (with two layers). Each worker defines a computation stage and the number of stages is referred to as the *pipeline depth* (*e.g.*, 4 in the example). One worker only communicates with nodes holding its previous stage or next stage. Each input batch is further divided into *microbatches*. In each iteration, each microbatch goes through all stages in a forward pass and then returns in an opposite direction in a backward pass. There are often multiple microbatches residing in the pipeline and different nodes can process different microbatches in parallel to improve utilization.

A key challenge in efficient pipeline parallelism is how to schedule microbatches. GPipe [24] schedules forward passes of all microbatches before any backward pass, as shown in Figure 1(b) where each node processes four microbatches. This approach leaves a "bubble" (*i.e.*, white cells) in the middle of the pipeline, leading to inefficient use of compute devices. PipeDream [38] proposes the one-forward-one-backward (1F1B) schedule to interleave the backward and forward passes, as shown in Figure 1(c). 1F1B can reduce the bubble size and the peak memory usage.

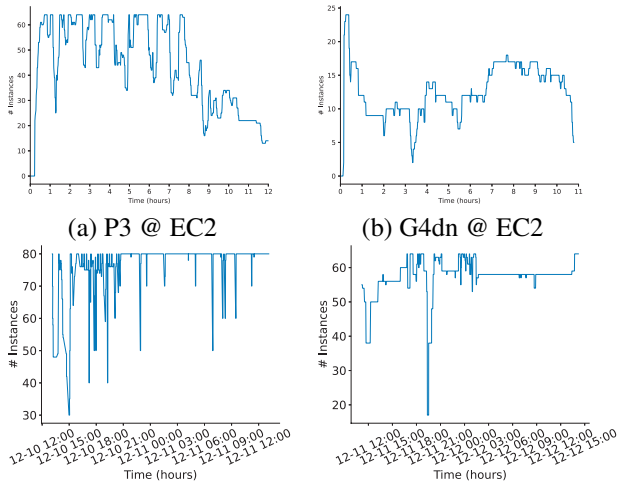
However, even with carefully-designed schedules, the pipeline bubble is still hard to eliminate. A fundamental reason is that it is extremely difficult to find the optimal layer partitioning to have each stage processed at the same rate. There exists a body of algorithms proposed recently to optimize layer partitioning and most of them are model- and hardware-specific [38, 16]. These algorithms are often time-consuming for large models, unsuitable for preemptible instances where the number of nodes keeps changing [2].

PipeDream [38] proposes asynchronous pipelining to eliminate the bubble—a node is allowed to work with stale weights to reduce the wait time. However, asynchronous microbatching introduces uncertainty in model convergence. In general, the effectiveness of synchronous v.s. asynchronous training is still open to debate. Furthermore, asynchronous training introduces inconsistencies in model state, which can create a more significant convergence issue when training occurs on preemptible instances, due to the need of frequent reconfigurations. For example, under synchronous microbatching, a reconfiguration can be performed at the end of each optimizer step (*i.e.*, parameter update), and hence the reconfigured pipelines can start with the up-to-date parameters. This is impossible to do under asynchronous microbatching.

As a result, we built Bamboo atop synchronous microbatching where model state is always consistent. Instead of attempting to reduce the bubble, we explore an orthogonal direction—how to leverage the bubble to run RC efficiently.

3 Motivation

This section motivates Bamboo from two aspects: (1) high preemption rates and unpredictability of spot instances, and (2) high performance overheads of strawman approaches.



(a) P3 @ EC2 (b) G4dn @ EC2

(c) n1-standard-8 @ GCP (d) a2-highgpu-1g @ GCP

Figure 2: Preemptions traces for a target cluster of size 64 instances on EC2 and 80 instances on GCP. Each graph shows a full-day trace for a GPU family in a cloud.

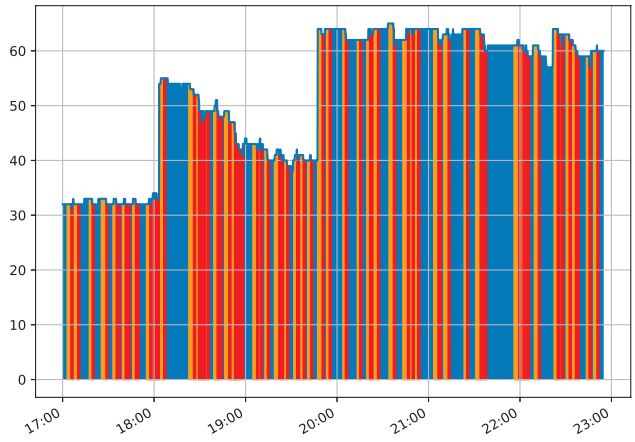


Figure 3: Training GPT-2 using checkpointing/restart with an autoscaling group of 64 P3 spot instances. Each color represents time spent in a distinct state,including Blue: training actively made progress; Orange: the cluster made progress that was then wasted; and Red: cluster restarting.

Preemptions of Spot Instances. We first studied failure models with spot instances on major public clouds. Figure 2 shows a set of real preemption traces collected from running spot instances in two public clouds: Amazon EC2 and Google Cloud Platform (GCP). For EC2, we used two GPU families: P3 (NVIDIA V100 GPUs with 32GB of memory) and G4dn (NVIDIA T4 GPUs with 16GB of memory). For GCP, we used `n1-standard-8` (NVIDIA V100 GPUs with 16GB GRAM) and `a2-highgpu-1g` (NVIDIA A100 GPUs with 40GB GRAM). For each family, we collected traces for a 24-hour window. In each experiment, we used an autoscaling group to maintain a cluster of 64 with an exception of `us-east1-c` in GCP, whose cluster size is 80. The autoscaling group, provided by each cloud, automatically allocates new instances upon preemptions to maintain the size (though without any guarantee).

From both families, node preemptions and additions are frequent and bulky (*i.e.*, many nodes get preempted at each time). This can make a checkpointing-based approach restart many times in a short window of time, leading to large inefficiencies (discussed shortly). Furthermore, both preemptions and allocations are unpredictable. While the autoscaling group attempts to allocate new nodes to maintain the user-specified size, allocations are committed incrementally; new allocations are mixed with preemptions of existing instances, making the spot cluster an extremely dynamic environment.

To understand the nature of the nodes that are preempted at the same time, we carefully analyzed two 24-hour preemption traces collected respectively from EC2 and GCP. For the EC2 trace, preemptions occur at 127 distinct timestamps, each of which see many preempted nodes. Of these 127 timestamps, only 7 see preemptions from multiple zones; at each of the remaining 120 timestamps, all nodes preempted come from the same zone. A similar observation was made on the GCP trace (12 out of 328 timestamps see cross-zone preemptions). These results confirmed the observations made by existing works [21, 20]: preemptions tend to be independent based on each individual spot market and each availability zone has a different and independent spot market—this is because each availability zone maintains capacity separately and therefore capacity preemptions in one zone are not associated with capacity preemptions in another.

These observations motivate our design—even with 1-node redundancies, Bamboo can recover from a majority of preemptions if consecutive nodes are not preempted at the same time; we maximize this possibility with a best-effort approach that makes consecutive nodes in each pipeline come from different zones. Although this may increase communication costs, it does not lead to visible performance impacts for Bamboo because Bamboo only sends (small amounts of) activations data between nodes.

Strawman #1: Checkpointing. We next show why a technique based on checkpointing and restarting does not work. We developed a new checkpointing system on top of DeepSpeed [51], providing checkpointing and restarting functionalities similar to TorchElastic [47] and Varuna [2]. We modified DeepSpeed to checkpoint *continuously* and *asynchronously*. In particular, each worker moves a copy of any relevant model state to CPU memory whenever the state is generated; the CPU then asynchronously writes it to remote storage so that training and checkpointing can fully overlap. During restarting, our system automatically adapts the prior checkpoints to the new pipeline configurations.

To understand how well this technique performs, we used it to train GPT-2 over 64 p3.2xlarge GPU spot instances on EC2. We profiled the training process and collected the check-

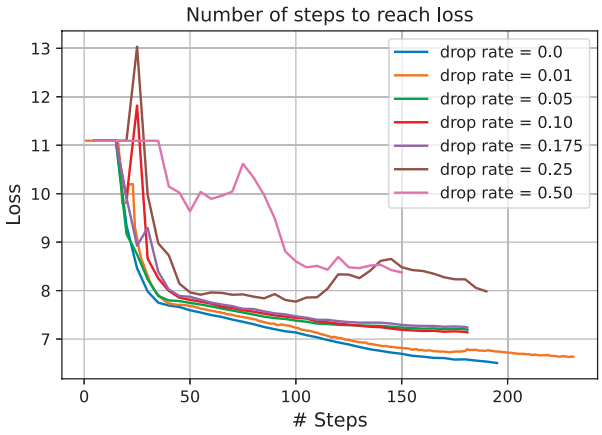


Figure 4: Effects of sample dropping under different rates.

pointing times, reconfiguration overheads, and total execution time. Figure 3 reports these results. The blue sections represent the times the system spent making actual progress for training. The red sections represent the times on reconfiguring (*i.e.*, restarting) while the orange sections show the times for wasted work—the computation that was done but not saved in checkpoints; the system ended up redoing these computations after restarting. This is because preemptions often occur during checkpointing, and hence, the system must roll back to a previous checkpoint. Frequent rollbacks slows down the training significantly. Note that systems such as Varuna and TorchElastic share this property and would have similar training patterns when facing regular preemptions. As shown, although checkpointing itself can be done efficiently, the restarting overheads (*i.e.*, for adapting existing checkpoints to new pipeline configurations) and the wasted computations take **77%** of the training time.

Strawman #2: Sample Dropping. An alternative approach that has shown promise is to take advantage of the statistical robustness of DNN training and allow some samples to be dropped so that training can continue without significant loss of accuracy [67, 36]. These techniques are also known as *elastic batching* because dropping samples is equivalent to changing the effective batch size at a training iteration (with the learning rate dynamically adjusted).

In the case of pipeline parallelism, we implemented sample dropping by suspending a pipeline upon losing an instance while letting other data-parallel pipelines continue to run. The system performs optimizer steps with the gradients of whichever data-parallel pipelines are able to complete that training step. Learning rate was adapted linearly with respect to the effective batch size to make sure that the only effect on the accuracy is the lost samples, but *not* a mismatch between hyperparameters and training configurations. In doing so, the training can continue for sometime without a reconfiguration (which is needed upon allocations).

We conducted a set of experiments to simulate the effect of sample dropping on model accuracy with a range of drop rates. Note that we could not obtain these results with the actual spot instances because we could not control the preemption rate. We ran a pre-training benchmark with GPT-2 using 16 on-demand instances from the same EC2 family, which form four data-parallel pipelines, each with four stages. To consider a range of different failure models, we used different rates of preemption to generate preemption events. Upon a preemption event, we randomly selected a pipeline and zero out the pipeline's gradients in that iteration. We measured the model's evaluation accuracy every 5 training steps. These results are shown in Figure 4 where each curve represents the function of the number of steps needed to reach a given loss for a particular drop rate.

Similarly to checkpointing, sample dropping works well for low preemption rates, but when frequent preemptions occur, many samples can be lost quickly and its impact on model accuracy quickly grows to be too significant to overlook. While this experiment was not an exact recreation of a sample dropping scenario, these results represent an *under-approximation* of the effect of the actual sample dropping (which can lose more accuracy than reported by Figure 4). This is because the actual sample dropping rate should be higher than the instance preemption rate—a preempted instance would likely be down for some time and consecutive samples would be dropped in a real setting. Note that training samples are shuffled before loading; hence, the effects of randomly dropping consecutive samples (*i.e.*, the actual scenario) and dropping random samples sporadically (*i.e.*, our experiment) should be similar.

4 Overview

Goal and Non-Goal. Our goal is *not* to automatically determine the cheapest way to train a given model (*e.g.*, which parallelism model can lead to the largest cost savings). Instead, Bamboo aims to enable efficient and preemption-safe training over cheap spot instances.

User Interface. To use Bamboo, a user specifies two system parameters D and P, as they normal would to use other pipeline-parallel systems, where D is the number of data-parallel pipelines and P is the pipeline depth. Due to the need of storing redundant layers, Bamboo requires a larger pipeline depth P than a normal pipeline-parallel system such as PipeDream [38]. We observed, empirically, that to avoid swapping data between CPU and GPU memory on the critical path, Bamboo's pipeline should be $\sim$1.5$\times$ (see §6.4) longer than an on-demand pipeline due to the extra memory needed to (1) hold the redundant layers and (2) accommodate potential pipeline adjustments. Given that spot instances are much cheaper (*e.g.*, 3-4$\times$ on EC2) than on-demand instances, training with 1.5$\times$ more nodes still leads to significantly reduced costs. While we recommend 1.5$\times$ more nodes, the number of active instances in a cluster is often much smaller due to preemptions and incremental allocations.

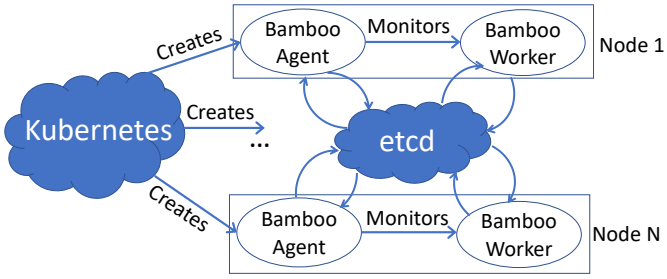


Figure 5: Bamboo runs one agent process per node (*i.e.*, spot instance). An agent monitors worker processes (each running a training script) that use our modified DeepSpeed. All workers and agents coordinate through `etcd` [42].

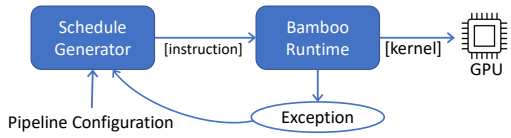


Figure 6: Bamboo worker.

$P \times D$ will be the size of the spot cluster Bamboo attempts to maintain throughout training. Preemptions can cause Bamboo to reduce the pipeline depth and/or the number of pipelines; in such cases, Bamboo would request more instances to bring the size of the cluster back to $P \times D$. However, Bamboo would never try to scale the training beyond $P \times D$. In other words, P and D are the *upper bound* of the pipeline depth and number of pipelines. It is important to note that the goal of the autoscaling framework we build for Bamboo is to adjust the pipelines *passively* in response to node preemptions and additions that we cannot control, rather than *proactively* finding an optimal cluster configuration to achieve better performance. This distinguishes Bamboo from existing works on autoscaling distributed training [43, 2, 25], whose goal was to find better configurations.

System Overview. Figure 5 shows an overview of our system. We built Bamboo on TorchElastic [47] and DeepSpeed [51]. In particular, we built the Bamboo agent, which runs on each node to kill/add a data-parallel pipeline, on top of TorchElastic. The agent monitors a Bamboo worker process on the same node, which is a DeepSpeed application enhanced with our support for redundant computation. Bamboo workers run D data-parallel pipelines that use an `all-reduce` phase to synchronize weights at the end of each iteration. Our spot instances are managed by Kubernetes [33], which is configured to automatically scale by launching a Bamboo agent on each new allocation. Our agents communicate and store cluster state on `etcd` [42], a distributed key-value store.

Each Bamboo worker uses a runtime to interpret the schedule, which produces a sequence of instructions, as shown in Figure 6. The schedule is generated statically based on the stage ID of the current worker and pipeline configurations, including the depth of pipeline and total number of micro-batches. The instructions consist of a computation component (*i.e.*, forward, backward, and apply gradient), and a communication component (*i.e.*, send/receive activation, send/receive gradient, and all-reduce). The Bamboo runtime interprets these instructions by launching their corresponding kernels on GPU. Communication instructions can fail due to preemptions. Upon a failure, the runtime throws an exception and falls back to use a failover schedule.

5 Redundant Computation

For ease of presentation, our discussion focuses on one node running one stage in the pipeline. Support for multi-GPU nodes will be discussed shortly.

Preemption of a node is detected by its neighboring nodes in the same pipeline during the execution of communication instructions. If a node on one side of the communication is preempted, the node on another side will catch an IO exception due to broken socket and update cluster state on `etcd`. Bamboo detects preemptions based on socket timeout. Although we could let a node to be preempted actively notify its neighbors in the grace period before the preemption, the length of this period varies across different clouds and hence Bamboo does not use it currently.

Since the victim node communicates with two nodes in the pipeline, both of its neighbors can catch the exception. The observed exception will be shared between these two nodes through `etcd`. This two-side detection is necessary for Bamboo to understand which node fails and generate the failover schedule. In addition to the two neighbors, nodes in other pipelines involved in the `all-reduce` operation also need to be informed. To safely perform `all-reduce`, each node participating in `all-reduce` reads the up-to-date cluster state on `etcd` and, if another pipeline has a failure, waits until the failure is handled.

5.1 Redundant Layers and Computation

To quickly recover from preemptions, Bamboo replicates the model partition on each worker node in each data-parallel pipeline. Instead of saving these replicas to a centralized remote storage (like checkpointing), Bamboo takes a *decentralized* approach by letting each node replicate its own model partition (*i.e.*, layer shard) on its predecessor node in the same pipeline. The first node has its layer replica stored on the last node in the pipeline. Conceptually, the last node is considered the "predecessor" of the first node. For simplicity of presentation, we use *forward stage IDs* to identify nodes, that is, a node that runs the forward stage $n+1$ is always considered as a successor of a node running the forward stage n (although in the backward pass, $n+1$ is a stage before n).

Our key idea is to let each node run normal (forward and backward) computation over its own layers and redundant (forward and backward) computation over the replica layers for its successor node. Let FRC_n^m/BRC_n^m denote the forward/backward redundant computation that is per-

formed on node m for node n, respectively. In Bamboo, $n = (m+1)\ mod\ P$ where P is the pipeline depth. Let FNC_n/BNC_n denote the forward/backward normal computation on node n. In Bamboo's pipeline, FRC^n_{n+1}/BRC^n_{n+1} is exactly the same computation as FNC_{n+1}/BNC_{n+1}, working with the same model parameters and optimizer states. To enable the last node to perform RC for the first node, we let it fetch input samples directly.

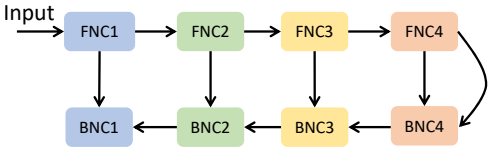


Figure 7: Dependencies between normal pipeline stages.

Why Neighboring Nodes? Due to our focus on pipeline parallelism, Bamboo performs RC on predecessor nodes to exploit *locality* for increased efficiency. To see this, we first need to understand the dependencies between different (backward and forward) pipeline stages that a microbatch goes through, as illustrated in Figure 7. For each forward stage FNC_n, it depends only on the output of its previous stage FNC_{n-1}. However, for each backward stage BNC_n, it has two dependencies: one on the output of stage BNC_{n+1} and a second on its corresponding forward stage FNC_n. The first is a *hard* dependency without which BNC_n cannot be done, while the second is a *soft* dependency primarily for efficiency—intermediate results produced by FNC_n can be reused to accelerate BNC_n. Without such cached results, BNC_n has to recompute many tensors (*i.e.*, tensor rematerialization [8]), leading to inefficiencies.

Figure 8 shows dependencies on an RC-enable pipeline where each node performs both normal and redundant (backward and forward) computation. Here solid/dashed arrows represent inter/intra-node dependencies. By running FRC for node $n+1$ on node n, *locality benefit* can be clearly seen because FRC only creates intra-node dependencies, which do not incur any extra communication overhead. However, in a backward pass, such a locality benefit does not exist for BRC^n_{n+1}, which requires the output of BNC_{n+2} and incurs much extra communication. This motivates our eager-FRC-lazy-BRC design which does not perform BRC until a preemption occurs and hence eliminates the extra communication cost in normal executions.

Note that we could also perform FRC lazily, but this would significantly increase the pause time for recovery. This is because (1) recovering from preemptions at both forward and backward pass now require a pause; and (2) lazy FRC would not produce intermediate results that can be used to speed up BRC and hence BRC's pause would be much longer. Since FRC can be scheduled in the pipeline bubble and overlap with FNC, performing it eagerly is a better choice.

The careful reader may think of an alternative approach that places node n's layer replica on node $n+1$ as opposed to

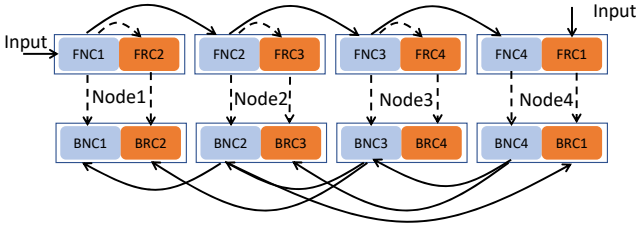


Figure 8: Dependencies between RC-enabled pipeline stages: solid/dashed arrows represent inter/intra-node dependencies; for simplicity, FRC_n/BRC_n in the figure represents FRC^{n-1}_n/BRC^{n-1}_n.

node $n-1$ (*i.e.*, its successor rather than its predecessor). This approach is symmetric to our design in that it turns inter-node dependencies for BRC into intra-node dependencies, but intra-node dependencies for FRC into inter-node dependencies. As a result, it eliminates the extra backward communication at the cost of increased forward communication. However, unlike Bamboo's design that can use lazy BRC to eliminate the extra backward communication, it is not as easy to eliminate the extra forward communication with lazy FRC—if FRC is not done eagerly in each iteration, BRC (regardless of whether it is eager or lazy) must perform tensor re-materialization, which incurs a long delay.

Level of Redundancy. As with any redundancy-based systems, the more redundancies, the higher level of resilience. For example, since Bamboo performs redundant computations only for one node, it cannot provide resilience when preemptions occur on consecutive nodes in a pipeline, in which case a reconfiguration is needed (see §A). However, enabling RC for multiple nodes can significantly increase the FRC time, making it much longer than what the bubbles can accommodate. Furthermore, the locality benefit (*i.e.*, FRC only incurs intra-node dependency) does not hold anymore, because FRC now depends on the outputs of multiple nodes. This can slow down the training substantially.

Takeaway. Storing each node's replica layers on its predecessor and running eager-FRC-lazy-BRC achieves low-overhead RC for pipeline parallelism. While this design does not support consecutive preemptions, Bamboo takes care to make consecutive nodes come from different zones. As discussed in §3, if multiple preemptions occur at the same time, the preempted nodes are highly likely to be from the same zone. As a result, our node assignment reduces the chance of consecutive preemptions, making RC effective for most preemptions. Although cross-zone data transfer can incur an overhead, this overhead is negligible (*e.g.*, <3%), as reported in Appendix §6.5, because in pipeline-parallel training, each node only passes a small amount of activation data to its neighbors.

We refer to the preempted node as the *victim node*, and the node saving the replica of the victim as its *shadow node*.

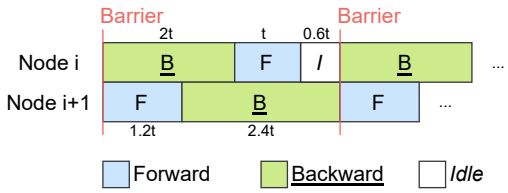


Figure 9: A closer examination of the pipeline bubble. Here we assume the forward pass on node i and $i+1$ takes time t and $1.2t$, respectively. Hence, a bubble of $0.6t$ exists before each communication barrier.

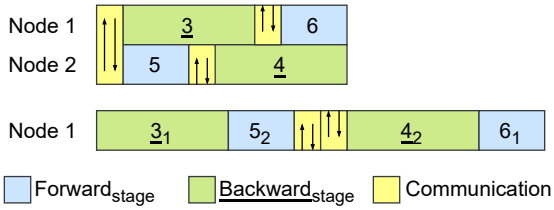


Figure 10: An example of merged instruction sequences in a failover schedule. We use PipeDream's 1F1B schedule as shown Figure 1(c), and assume node 2 is the victim node and node 1 is the shadow node.

5.2 Schedule Redundant Computation

It is straightforward to see that RC incurs an overhead in both time and memory. We propose to (1) schedule FRC into the pipeline bubble to reduce forward computation overhead, (2) perform BRC lazily to reduce backward computation/communication overhead, and (3) offload unnecessary tensors to CPU memory to reduce memory overhead.

Eager FRC. As discussed in §2, the pipeline bubble can come from either imperfect scheduling or unbalanced pipeline partitioning. To illustrate, consider Figure 9 with PipeDream's 1F1B schedule. Suppose there are two consecutive nodes in the pipeline where both the forward and the backward computation of node $i+1$ run $1.2\times$ slower than those of node i. The communication between these two nodes serves as a barrier. Since node i runs faster, it always reaches the barrier earlier and waits there until node $i+1$ arrives. This wait period is where we should schedule FRC.

Bamboo builds on the 1F1B schedule (Figure 1(a)) due to its additional efficiency compared to GPipe's schedule (Figure 1(b)). However, even for 1F1B, bubbles widely exist in a pipeline—as a microbatch passes different pipeline stages, the later a stage, the longer the (backward and forward) computation takes. This is because for the 1F1B schedule, the number of active microbatches in a later stage is always smaller than that in an earlier stage. In Figure 1 (c), for example, node 1 has 3 active microbatches while node 2 only has 2. Consequently, later stages often consume less memory. To balance memory usage, the layer partition on a later node is often larger that that on an earlier node in the pipeline, and hence a later stage runs slower. A detailed analysis of bubble size can be be found in Appendix §C.1.

Scheduling. Based on this observation, we schedule FRC on a node before the node starts communicating with its successor node. This is where a bubble exists. In cases where the FRC cannot fit entirely into the bubble (*i.e.*, for the last four stages in Figure 14), we overlap FRC and FNC as much as we can. However, for the same microbatch, FRC^n_{n+1} depends on FNC_n and they cannot run in parallel. To resolve this dependency issue, we focus on different microbatches for FNC and FRC. That is, Bamboo schedules FNC_n for the k-th microbatch and FRC^n_{n+1} for its previous $(k-1)$-th microbatch to run in parallel. Since there is no dependency between them, their executions can overlap.

To reduce memory overhead, Bamboo follows a well-known principle to offload less frequently used tensors to CPU memory. Specially, since BRC is *not* performed in normal training passes, FRC's outputs and intermediate results are not needed until a preemption occurs and BRC is triggered. As a result, we swap out these data after FRC is done for each microbatch on each node. These data take the majority of FRC's memory consumption; swapping them out significantly reduces FRC's GPU memory usage [52]. However, we leave the redundant weights in GPU memory for efficient FRC because these weights are needed for FRC on each microbatch.

Lazy BRC and Recovery. BRC is executed by a *failover schedule* which a node runs when detecting its successor node fails. In particular, for the current iteration, all the lost gradients must be re-computed, while for the following iterations, all instructions of the victim node must be executed by its shadow node (until a reconfiguration occurs). Nodes that originally communicate with the victim node are transparently rerouted to the shadow node. The failover schedule is generated by merging the schedules of the victim and shadow node. In particular, a schedule consists of a sequence of instructions and we divide it into two groups—(1) continuous communication instructions, which is placed at the head of a group and (2) computation instructions that can be executed without remote data dependencies.

When the two instruction groups (from the victim and shadow nodes) are merged, the instructions are interleaved with the following rules. (1) Communication instructions are still placed in the beginning of the merged groups. (2) Communications that used to be inter-node between the victim and the shadow are removed. (3) External communications from the victim node are first performed. (4) Computation instructions are ordered such that backward computation is always executed earlier; after the backward computation is done, the memory occupied by intermediate results is freed. Figure 10 shows an example of merged instruction sequences if node 2 is the victim node and node 1 is the shadow node.

Model	Dataset	Samples	D	P
ResNet-152 [22]	ImageNet [32]	300,000	4	12
VGG-19 [63]	ImageNet [32]	1,000,000	4	6
AlexNet [32]	ImageNet [32]	1,000,000	4	6
GNMT-16 [68]	WMT16 EN-De	200,000	4	6
BERT-Large [15]	Wikicorpus En [15]	2,500,000	4	12
GPT-2 [49]	Wikicorpus En [15]	500,000	4	12

Table 1: Our models, datasets, pipeline configurations.

Support for Multi-GPU Nodes. Bamboo's RC works for multi-GPU settings—this requires replicating all layers that belong to the GPUs of one node in the GPUs of its predecessor node. In other words, we use "group replicas" as opposed to individual replicas. However, in the presence of frequent preemptions, using multi-GPU would yield poorer performance—losing one node (with multiple GPUs) is equivalent to losing multiple nodes in the single-GPU setting. Our evaluation (§6) shows that it is much harder to allocate new multi-GPU nodes during training than single-GPU nodes.

Once Bamboo loses too many nodes or there are many idle nodes (*i.e.*, new allocations) waiting to join the pipelines, Bamboo launches a reconfiguration. Details of the reconfiguration process can be found in §A.

Support for Pure Data Parallelism. Bamboo supports pure data parallelism (without model partitioning). Due to space constraints, here we briefly discuss how it is supported. We use the same redundant computation strategy—Bamboo replicates the parameter and optimizer state of each node on a different node and uses these replicas as redundancies to provide quick recovery. For pure data parallelism, there is no bubble time to schedule RC. Eager FRC would be equivalent to overbatching (*i.e.*, each node processes its original minibatch plus a redundant minibatch). To reduce the FRC overhead and make RC fit into the GPU memory constraints, we over-provision spot instances (by 1.5×, in the same way as discussed in §5) to make each node process a smaller batch.

Enabling eager FRC doubles the batch size. However, it results only in a ∼1.5× increase in the computation time due to the parallelism provided by GPUs. This overhead can be effectively reduced by slightly over-provisioning (1.5 $\times D$) nodes, increasing the degree of parallelism and decreasing the impact of overbatching. This enables us to run FRC eagerly without incurring much overhead (*i.e.*, <10%).

Once Bamboo loses too many nodes or there are many idle nodes (*i.e.*, new allocations) waiting to join the pipelines, Bamboo launches a reconfiguration. Details of the reconfiguration process can be found in Appendix §A.

6 Evaluation

Bamboo is implemented in ∼7K LoC as a standard Python library. We evaluated Bamboo by pretraining a range of popular vision and language models, as shown in Table 1. For the first four (smaller) models that were also used in PipeDream [38] (which actually used smaller versions of these models), we took the values of D (the number of data-parallel pipelines) and P_{demand} (pipeline depth) from PipeDream [38]'s configurations.

As discussed earlier in §4, to avoid swapping Bamboo needs 1.5× more instances for each pipeline and hence each P reported in Table 1 equals $1.5\times P_{demand}$. For BERT and GPT2, we used 4 and 8×1.5=12 as D and P. We have also evaluated with another pipeline depth $P_h = P_{demand} \times \frac{Price_{demand}}{Price_{spot}}$; these results can be found in §6.2.

We trained these models on a spot cluster from EC2's p3 family where each instance has V100 GPU(s) with 16GB GPU memory and 61GB CPU memory. Each on-demand instance costs $3.06/hr per GPU while the price of its spot counter-part (at the time of our experiments) is $0.918/hr. Our evaluation uses two on-demand baselines: (1) p3 instances each with four V100 GPUs (Demand-M) and (2) p3 instances each with a single GPU (Demand-S). For both baselines, the pipeline configuration was the same and all nodes were obtained from one availability zone.

For all experiments, we trained each model to a target validation accuracy, which is a particular number of samples for the model. We did not train them to higher accuracies because large models take a huge amount of time to train (*e.g.*, weeks) to reasonable accuracies; using such a large amount of resources (even spot instances) goes beyond our financial capabilities. Furthermore, Bamboo uses synchronous training where the time per iteration is fixed; hence, training for extended time would not change our results.

For on-demand instances, we used the largest per-GPU minibatch that fits in one GPU's memory—anything larger yields out-of-memory exceptions. This ensures that we hit peak achievable FLOPs on a single device. For data-parallel runs with n workers, the global minibatch size is $n \times g$ where g is the minibatch size. The global minibatch sizes we used are consistent with those used by the ML community and reported in the literature for these models. We used a per-GPU minibatch size of 256 per GPU for VGG-19, 512 for AlexNet, 2048 for ResNet-152, 32 for GNMT-16, 256 for BERT-Large, and 256 for GPT-2. For microbatch size, we always selected a small value and tuned it for different models/configurations. We trained the vision models with an initial learning rate of 0.001, respectively, with a vanilla SGD optimizer [29]. For language models, we used the Adam optimizer [30] with an initial learning rate of $6e^{-3}$. We used half (fp16) precision in all our experiments.

6.1 Training Performance and Costs

Overall Performance. To thoroughly and deterministically evaluate Bamboo's performance over spot instances under different preemption rates, we first ran a 48-node cluster (*i.e.*, the configuration for ResNet, BERT, and GPT) and a 32-node cluster (*i.e.*, for VGG, AlexNet, and GNMT) on AWS and collected a 24-hour preemption trace for each. On these traces, the *hourly preemption rate* varies significantly, ranging from

Model	System	Time (Hours)	Throughput	Cost ($/hr)	Value
ResNet	D-M	2.78	30.00	97.92	0.31
	D-S	2.60	32.00	97.92	0.33
	B-M	[4.31, 5.31, 10.14]	[19.35, 15.69, 8.22]	[44.33, 40.01, 37.21]	[0.43, 0.39, 0.22]
	B-S	[**3.85**, 4.29, 6.87]	[**21.67**, 19.41, 12.13]	[**42.23**, 40.39, 36.72]	[**0.51**, 0.48, 0.33]
VGG	D-M	1.41	197.00	48.96	4.02
	D-S	1.66	167.00	48.96	3.41
	B-M	[2.98, 3.67, 4.33]	[93.34, 75.75, 64.22]	[21.31, 19.55, 18.43]	[4.38, 4.11, 3.48]
	B-S	[**1.81**, 2.22, 2.83]	[**153.31**, 124.88, 98.21]	[**20.19**, 19.28, 18.36]	[**7.59**, 6.48, 5.35]
AlexNet	D-M	0.77	359.00	48.96	7.33
	D-S	0.78	336.00	48.96	6.86
	B-M	[1.02, 1.34, 1.93]	[271.06, 207.43, 143.57]	[21.31, 19.55, 18.43]	[12.72, 10.61, 7.79]
	B-S	[**0.82**, 0.86, 0.99]	[**340.32**, 321.65, 280.42]	[**20.19**, 19.28, 18.36]	[**16.86**, 16.68, 15.27]
GNMT	D-M	2.06	27.00	48.96	0.55
	D-S	2.31	24.00	48.96	0.49
	B-M	[3.98, 5.13, 8.78]	[13.95, 10.82, 6.33]	[21.31, 19.55, 18.43]	[0.65, 0.55, 0.34]
	B-S	[**2.94**, 3.41, 6.31]	[**18.92**, 16.31, 8.8]	[**20.19**, 19.28, 18.36]	[**0.94**, 0.85, 0.48]
BERT	D-M	5.89	118.00	97.92	1.21
	D-S	6.43	108.00	97.92	1.10
	B-M	[9.75, 12.31, 16.66]	[71.22, 56.41, 41.68]	[44.33, 40.01, 37.21]	[1.61, 1.41, 1.12]
	B-S	[**7.02**, 8.3, 11.46]	[**98.87**, 83.70, 60.59]	[**42.23**, 40.39, 36.72]	[**2.34**, 2.07, 1.65]
GPT	D-M	4.34	32.00	97.92	0.32
	D-S	4.63	30.00	97.92	0.30
	B-M	[7.83, 9.92, 12.04]	[17.73, 14.00, 11.54]	[44.33, 40.01, 37.21]	[0.40, 0.35, 0.31]
	B-S	[**4.64**, 6.12, 10.08]	[**29.92**, 22.68, 13.78]	[**42.23**, 40.39, 36.72]	[**0.71**, 0.56, 0.38]

Table 2: Comparisons between training with DeepSpeed over on-demand instances and Bamboo over spot instances. For Bamboo, we trained each model three times, and their results are explicitly listed in the form of $[a, b, c]$ for the 10% (average), 16%, and 33% preemption rates, respectively.

no preemption all the way to 16 nodes preempted (33%), with an average rate of 4-6 nodes per hour (8-12%). To account for such changes, we extracted from each trace three segments, each with a different hourly preemption rate: 10%, 16%, and 33%. We used AWS' fleet manager to trigger preemptions by replaying these segments. Note that if we were to run Bamboo over the uncontrolled spot cluster, there would be no way to enable a direct comparison.

We trained ResNet, BERT, and GPT by replaying the three segments from the 48-node trace, and VGG, AlexNet, and GNMT by using the segements from the 32-node trace. These results are reported in Table 2. In addition to the time and monetary costs, we used a metric called *value*, which measures performance-per-dollar. Value is computed as $V = \frac{T}{C}$ where T is the training throughput, measured in terms of the number of samples per second, and C is the monetary cost per hour. Throughout the evaluation, we used both value and throughput as our metrics.

Our first observation is Demand-M slightly outperforms Demand-S due to reduced cross-node communication. However, the difference is marginal as the amount of data (*i.e.*, only activations) transferred over the network is small. Bamboo-S significantly outperforms Bamboo-M (*i.e.*, **1.4**× higher throughput and **1.5**× higher value) because (1) multi-GPU nodes are subject to more GPU failures with the same number of preemptions and (2) it is much harder to to allocate new nodes in a timely fashion.

For Bamboo-S, the results in each bracket of the form [a, b, c] show Bamboo's performance under the three preemption rates. The higher the preemption rate, the worse Bamboo's throughput and value. Given that the average preemption rate is ∼10%, the first number in each bracket (highlighted) represents Bamboo's performance on the used spot cluster. On average, Bamboo's throughput (under the 10% preemption rate) is 15% lower than DeepSpeed running over $D \times P_{demand}$ instances. There are three major reasons.

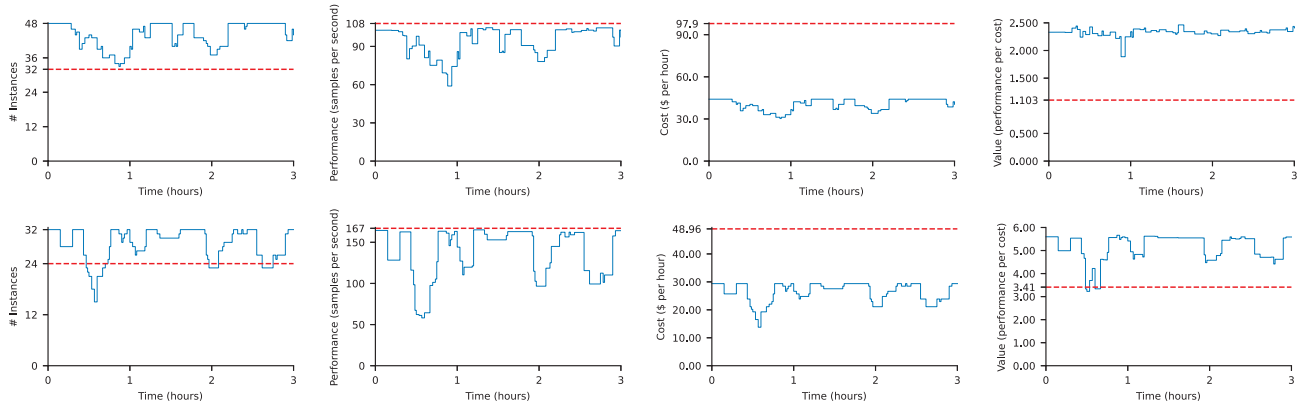
First, the number of active instances in the spot cluster is actually lower than the requested size $D \times P$. For ResNet, for example, the average number of instances throughout the training is only 25.58 although the requested cluster size is 48 (and the on-demand cluster always has 32 nodes). The autoscaling group keeps attempting to add new instances but the total number of active instances only reaches the requested size for a small period of time.

Second, Bamboo's reconfiguration contributes to reduced throughput—these overheads vary with environments and take an average of 7% of the total training time.

Third, the time for each iteration increases due to eager FRC. This is the major source of overhead for language models such as GPT-2. A detailed evaluation of RC's overhead can be found in §6.4.

Despite the small throughput reduction, Bamboo delivers an overall of **1.95**× higher value compared to training with on-demand instances. The benefit in value remains clear for five models (ResNet, VGG, AlexNet, BERT and GPUT) even when the preemption rate increases to 33% (*i.e.*, the worst-case segment of the collected trace).

To have a closer examination of Bamboo-S' training, we showed the traces for BERT-large and VGG-19, and plotted them in Figure 11. The two rows show (a) preemption traces (under the 10% rate), (b) training throughputs, (c) monetary costs, and (d) values, for BERT-large and VGG-19, respectively. Since Bamboo-M underperforms Bamboo-S, we focus on Bamboo-S in the rest of the evaluation.



(a) Trace (b) Training Throughput (c) Monetary Cost (d) Value

Figure 11: Bamboo's training performance for BERT (top) and VGG (bottom), compared to on-demand instances (red lines).

Prob.	Prmt (#)	Inter. (hr)	Life (hr)	Fatal Fail. (#)	Nodes (#)	Thruput	Cost ($/hr)	Value
0.01	8.50	2.08	15.20	0.06	45.18	87.99	41.11	2.10
0.05	48.15	0.44	10.14	0.23	43.65	76.35	39.73	1.90
0.10	99.77	0.23	6.71	0.29	41.69	72.12	37.94	1.88
0.25	276.52	0.10	3.13	1.04	35.80	60.12	32.58	1.82
0.50	709.83	0.06	1.49	5.98	26.96	40.37	24.53	1.59

(a) Results of simulating training BERT *until completion*; each preemption probability ran 1,000 times.

Prob.	Thruput	Cost ($/hr)	Value
0.01	54.87	90.73	0.60
0.05	50.66	87.43	0.58
0.10	49.18	83.23	0.59
0.25	40.59	71.24	0.57
0.50	26.24	53.05	0.49

(b) Simulation results of training BERT-large with pipeline depth P_h (which is $3.3\times P_{demand}$).

Table 3: Simulation results for more configurations.

6.2 Different Failure Models

This section demonstrates Bamboo's ability to affordably train large DNNs across a wide range of failure models. To this end, we developed an offline simulation framework that takes as input (1) the preemption probability (including preemption frequency and the number of preemptions in each bulk), (2) per-iteration training time, and (3) Bamboo's recovery and reconfiguration time, automatically calculating training performance, costs, and values. Here we focus on BERT-large and simulated its training until completion.

We experimented using 5 different preemption probabilities (*i.e.*, preemption rate per hour), and kept the preemption probability constant throughout the entire run (as opposed to replaying traces). To mimic realistic spot instance creation and preemption, we randomly generated different creation probabilities per hour and also randomly picked zones for allocations. For each preemption probability, Table 3a reports the average numbers of preemptions, intervals (*i.e.*, average time, in hours, between preemption events), average lifetime of an instance (in hours), average numbers of fatal failures (which require a restart from a checkpoint), average numbers of instances in the cluster, throughput (*i.e.*, #samples per second), costs, and values, across 1,000 simulations.

Our simulations show that Bamboo's values match our real-world runs as just reported in §6.1. Further, regardless of the preemption probability, the value of Bamboo remains stable and is constantly higher than that of training with on-demand instances (which is 1.1). This is because most preemptions can be quickly recovered without introducing much overhead. The higher the preemption probability, the less the active instances running training jobs; this is the major source of the performance slowdown. However, the cost is reduced also proportionally, leading to stable values.

Simulation for P_h. To understand the tradeoff in choosing P, we experimented with another value of P for BERT-large: P_h, which is $\frac{3.06}{0.918} \times P_{demand}$. This configuration represents the *upper-bound* of the spot training resources that can be obtained within the cost of training with P_{demand} on-demand instances (while D remains unchanged). Note that in practice the number of active instances can barely reach the requested size and hence the cost of using a spot cluster of size $P_h \times D$ is often still much lower than training with an on-demand cluster of size $P_{demand} \times D$.

To avoid incurring a large monetary cost, we used the same simulator to run this experiment. These results are reported in Table 3b. As shown, using P_h actually decreases both throughput (compared to 84 under P in Table 2) and value (due to significantly increased costs). This is because using too a large pipeline leads to poorer partitioning, underutilized resources and inferior performance.

6.3 Comparisons with Other Systems

We have reported the performance of training GPT-2 with asynchronous checkpointing and restart in Figure 3—the checkpointing-based approach spent only 23% on actual training, while Bamboo increases this percentage to **84%**. In fact, as shown in Table 3a, even for the preemption rate of 0.5, there are only 5.98 fatal failures that would require checkpoint-

ing/restart under Bamboo. On the contrary, a checkpointing-based approach would need to restart the pipeline for every one of the 709.83 preemptions. Similarly, sample dropping significantly slows down the training when the preemption rate increases, as shown in Figure 4.

Varuna. Varuna [2] is a system developed concurrently with Bamboo to enable training on spot instances. As with other existing techniques, Varuna provides resilience with checkpointing. We set up Varuna on the same spot cluster on AWS EC2 as we used in § 6.1. We ran Varuna with a $D \times P$ pipeline (*i.e.*, the same as on-demand instances) because Varuna does not use redundancies and hence not need to over-provision resources.

We trained BERT on Varuna with the same configurations, including the same datasets, model architectures, float precision, preemption rates, and hyperparameters. Varuna hung under the 33% preemption rate. For the 10% and 16% preemption rates, comparisons between Varuna and Bamboo-S are reported in Figure 12. As shown, Bamboo-S outperforms Varuna by **2.5**× and **2.7**× in throughput, respectively, under the 10% and 16% rates; and by **1.67**× and **1.64**×, in value, under the two rates. Note that value benefits are lower than throughput benefits due to Varuna's use of fewer instances. To understand the cause of Varuna's slowdown, see §3. Varuna follows a similar pattern, having to frequently restart and redo lost computations.

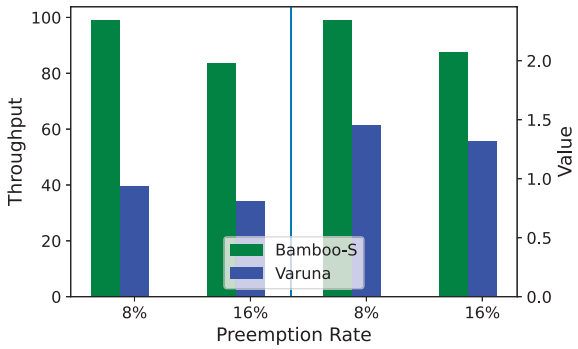


Figure 12: Throughput and value for Bamboo-S and Varuna running BERT at different preemption levels. Varuna hung at the 33% preemption rate.

6.4 Microbenchmarks of Redundant Computation

To fully understand the overhead introduced by RC, we compared time and memory among three versions of RC: eager-FRC-lazy-BRC (EFLB, Bamboo's approach), eager-FRC-eager-BRC (EFEB), and lazy-FRC-lazy-BRC (LFLB), when training BERT and ResNet. Since the focus here is the RC overhead, we ran this experiment over on-demand instances.

Table 4 reports RC's time overheads for the three RC settings. As expected, LFLB incurs the lowest per-iteration overhead because neither FRC nor BRC is performed with normal training iterations. The ∼7% overhead comes primarily from the extra code executed to prepare for a failover schedule. However, the recovery time is much longer under LFLB than the other two settings (discussed shortly). On the contrary, EFEB has the highest per-iteration overhead due to the eager execution of both FRC and BRC. The overhead incurred by EFLB, as used in Bamboo, is slightly higher than LFLB but much lower than EFEB. This is because eager FRC does not incur extra communication overhead and much its computation overhead can be hidden by scheduling it into the pipeline bubble and overlapping it with FNC.

Redundancy Mode	BERT	ResNet
Lazy-FRC-Lazy-BRC	7.01%	7.65%
Eager-FRC-Lazy-BRC (Bamboo)	19.77%	9.51%
Eager-FRC-Eager-BRC	71.51%	64.24%

Table 4: Time overhead with different RC settings.

Another interesting observation is the overhead for ResNet is lower than for BERT. This is because ResNet's layer partitioning is much more imbalanced than that of BERT (which is a transformer model where most the middle layers are equivalent). As a result, the bubble in ResNet's pipeline is much larger and hence it can accommodate a more significant fraction of FRC.

Eager FRC incurs an overall ∼1.5× overhead in GPU memory (that is why Bamboo recommends creating pipelines with 1.5× more nodes) while lazy FRC does not incur any memory overhead.

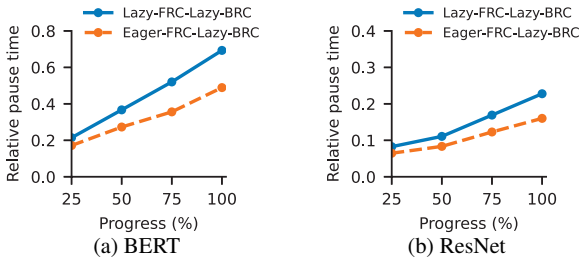


Figure 13: Relative pause time for BERT and ResNet under different RC settings. Bamboo runs into a pause when a pipeline stops training and waits for the shadow node to recover the lost state on the victim node.

To understand the pause time under these different RC settings, Figure 13 shows the relative pause time (*i.e.*, the actual pause time relative to the time of each training iteration without preemptions). As shown, lazy FRC reduces pause time by ∼35% despite the slightly higher per-iteration overhead it introduces. In summary, eager-FRC-lazy-BRC strikes the right balance between overhead and pause time.

Model	Config	Throughput	Total Transferred Bytes
BERT	Spread	148.923	16.39 GiB
BERT	Cluster	151.124	16.39 GiB
VGG19	Spread	160.12	11.213 GiB
VGG19	Cluster	165.77	11.213 GiB

Table 5: Comparison of throughput when running across availability zones compared to running within a single zone.

6.5 Cross-Zone Communication

Because Bamboo allocates workers across availability zones to minimize the probability of reconfigurations, we measured the overhead incurred by cross-zone communication. We ran Bamboo in two configurations: (1) with nodes distributed across all zones (*i.e.*, Spread) and (2) in a single availability zone with AWS' "Placement Group" option set to "Cluster" (*i.e.*, Cluster), and measured their performance differences. As reported in Table 5, the differences between these two configurations are quite low (*i.e.*, usually less than 5%). This demonstrates Bamboo's choice of assigning nodes from different availability zones as consecutive nodes in each pipeline has little impact on training performance.

7 Related Work

Parallel Training. Data parallelism [28, 14, 32, 7, 12, 72, 35, 72] is the most common parallelism model that partitions the dataset and trains on each partition. The learned weights are synchronized via either an all-reduce approach [7] or parameter servers [35, 10]. Model parallelism [14, 31, 43, 60, 62] partitions the operators in a DNN model across multiple GPU devices, with each worker evaluating and performing updates for only a subset of the model's parameters for all inputs. Recently, pipeline parallelism [24, 38, 71, 65] has been proposed to train large models by partitioning layers across workers and uses microbatches to saturate the pipeline. Popular DL training libraries such as DeepSpeed [51] and Megatron [40] support 3D parallelism, which combines data parallelism, model parallelism, and pipeline parallelism to train models at extremely large scale with improved compute and memory efficiency. Furthermore, DeepSpeed offers ZeRO-style data parallelism [52], which partitions model states across GPUs and uses communication collectives to gather individual parameters when needed.

Elastic Training. Distributed training experiences frequent resource changes. There are a number of systems [43, 21, 47, 23, 48, 25] built to provide elasticity for training over changing resources. TorchElastic [47] is a PyTorch [44]-based tool that can dynamically kill or add data-parallel workers. Huang *et al.* [23] considers elasticity for declarative ML on MapReduce, which does not work for modern deep learning workloads. Litz [48] is a system that provides elasticity in the context of CPU-based machine learning using the parameter servers. Or *at al.* [43] presents an autoscaling system built on top of TensorFlow [1] and Horovod [55], which dynamically adapts the batch size and reuses existing processes.

Exploiting Spot Instances. Proteus [21] exploits dynamic pricing on public clouds in order to lower costs for machine learning workloads through elasticity. Since Proteus does not explicitly consider modern deep learning workloads, Proteus simply reprocesses the input of a preempted node with another node. Varuna [2] is a system built concurrently with Bamboo for distributed training over spot instances. However, Varuna focuses on elasticity, not quick recovery from preemptions. Bamboo, on the contrary, is designed specifically to deal with frequent preemptions.

There exists a body of work on enabling low latency and/or SLO guarantees when using preemptible spot instances. Tributary [20] is an elastic control system that exploits preemptible resources to reduce cost with SLO guarantees. Kingfisher [59] proposes a cost-aware resource acquisition scheme that uses integer linear programming to determine a service's resource footprint among a heterogeneous set of non-preemptible instances with fixed prices. Flint [56] is a system that runs batch-based data-intensive jobs on transient servers. SpotCheck [58] selects spot markets to acquire instances in while always bidding at a configurable multiple of the spot instance's corresponding on-demand price. BOSS [70] hosts key-value stores on spot instances by exploiting price differences across pools in different data-centers. ExoSphere [57] is a virtual cluster framework for spot instances. These systems are all orthogonal to Bamboo that is built specifically for deep learning training.

GPU Scheduling. There is also a large body of work on GPU scheduling [61, 69, 74, 46, 37, 19, 39, 40, 34, 73] for ML workloads. These techniques are orthogonal to Bamboo —they all focus on efficiency and throughput while Bamboo aims to perform redundant computation at a low cost.

8 Conclusion

Bamboo is the first distributed system that uses redundant computation to provide resilience and fast recovery for training DNNs over preemptible instances. An evaluation with 6 representative models shows that Bamboo provides a much higher value than (1) training on on-demand instances and (2) training with checkpointing/restart on spot instances.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Yiting Xia for her feedback. This work is supported by NSF grants CNS-1703598, CNS-1763172, CNS-1907352, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, CNS-2147909, CNS-2152313, CNS-2151630, CNS-2140552, and CNS-2153449, ONR grant N00014-18-1-2037, a Sloan Research Fellowship, and research grants from Cisco.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: a system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys*, 2021.

[3] AWS. Amazon ec2 spot instances pricing. https://aws.amazon.com/ec2/spot/pricing/, 2021.

[4] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin. PipeSwitch: fast pipelined context switching for deep learning applications. In *OSDI*, pages 499–514, 2020.

[5] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Lightweight preemptible functions. In *USENIX ATC*, pages 465–477, 2020.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. In *NIPS*, 2020.

[7] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. In *ICLR Workshop Track*, 2016.

[8] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost, 2016.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: efficient primitives for deep learning, 2014.

[10] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[11] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25, 2019.

[12] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: scalable deep learning on distributed GPUs with a gpu-specialized parameter server. In *EuroSys*, 2016.

[13] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, Lake Tahoe, Nevada. Curran Associates Inc., 2012.

[14] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1223–1231, 2012.

[15] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL: http://arxiv.org/abs/1810.04805.

[16] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, and W. Lin. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP*, pages 431–445, 2021.

[17] W. Fedus, B. Zoph, and N. Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *CoRR*, 2021.

[18] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training ImageNet in 1 hour. *CoRR*, abs/1706.02677, 2017. URL: http://arxiv.org/abs/1706.02677.

[19] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: a gpu cluster manager for distributed deep learning. In *NSDI*, pages 485–500, 2019.

[20] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *USENIX ATC*, pages 1–14, 2018.

[21] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons. Proteus: agile ML elasticity through tiered reliability in dynamic resource markets. In *EuroSys*, 2017.

[22] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[23] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, pages 137–152, 2015.

[24] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018. URL: http://arxiv.org/abs/1811.06965.

[25] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. In *NSDI*, pages 721–739, 2021.

[26] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica. Checkmate: breaking the memory wall with optimal tensor rematerialization. In *MLSys*, pages 497–511, 2020.

[27] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *USENIX ATC*, pages 947–960, 2019.

[28] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. In *MLSys*, 2019.

[29] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952.

[30] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization, 2014.

[31] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL: http://arxiv.org/abs/1404.5997.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[33] Kubernetes: an open-source system for automating deployment, scaling, and management of containerized applications. https://kubernetes.io/, 2021.

[34] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: opening the black box of machine learning prediction serving systems. In *OSDI*, pages 611–626, 2018.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[36] H. Lin, H. Zhang, Y. Ma, T. He, Z. Zhang, S. Zha, and M. Li. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. *CoRR*, 2019.

[37] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: fair and efficient GPU cluster scheduling. In *NSDI*, pages 289–304, 2020.

[38] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*, pages 1–15, 2019.

[39] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *OSDI*, pages 481–498, 2020.

[40] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia. Efficient large-scale language model training on gpu clusters using Megatron-LM. In *SC*, 2021.

[41] A. Newell, D. Skarlatos, J. Fan, P. Kumar, M. Khutornenko, M. Pundir, Y. Zhang, M. Zhang, Y. Liu, L. Le, B. Daugherty, A. Samudra, P. Baid, J. Kneeland, I. Kabiljo, D. Shchukin, A. Rodrigues, S. Michelson, B. Christensen, K. Veeraraghavan, and C. Tang. RAS: continuously optimized region-wide datacenter resource allocation. In *SOSP*, pages 505–520, 2021.

[42] Operating etcd clusters for Kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/, 2021.

[43] A. Or, H. Zhang, and M. Freedman. Resource elasticity in distributed deep learning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, volume 2, pages 400–411, 2020.

[44] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: an imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, 2019.

[45] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, pages 109–116, 1988.

[46] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[47] PyTorch Developers. TorchElastic. 2021. URL: https://pytorch.org/docs/stable/distributed.elastic.html.

[48] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: elastic framework for High-Performance distributed machine learning. In *USENIX ATC 18*, pages 631–644, 2018.

[49] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. In 2019.

[50] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language Models are Unsupervised Multitask Learners, 2019.

[51] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: memory optimizations toward training trillion parameter models. In *SC*, 2020.

[52] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[53] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech dnns. In *ICASSP*, pages 235–239, 2014.

[54] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, Sept. 2014.

[55] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018. URL: http://arxiv.org/abs/1802.05799.

[56] P. Sharma, T. Guo, X. He, D. E. Irwin, and P. J. Shenoy. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *EuroSys*, 2016.

[57] P. Sharma, D. Irwin, and P. Shenoy. Portfolio-driven resource management for transient cloud servers. In *SIGMETRICS*, page 59, 2017.

[58] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: designing a derivative IaaS cloud on the spot market. In *EuroSys*, 2015.

[59] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *ICDCS*, pages 559–570, 2011.

[60] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. A. Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. In *NIPS*, 2018.

[61] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *SOSP*, pages 322–337, 2019.

[62] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR*, 2019.

[63] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2015.

[64] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, 2005.

[65] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *OSDI*, pages 495–514, 2021.

[66] I. Turc, M. Chang, K. Lee, and K. Toutanova. Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation. *CoRR*, 2019.

[67] T. Wang, J. Huan, and B. Li. Data dropout: optimizing training data for convolutional neural networks. In *ICTAI*, pages 39–46, 2018.

[68] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL: http://arxiv.org/abs/1609.08144.

[69] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: dynamic scaling on GPU clusters for deep learning. In *OSDI*, pages 533–548, 2020.

[70] Z. Xu, C. Stewart, N. Deng, and X. Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *INFOCOM*, pages 1–9, 2016.

[71] B. Yang, J. Zhang, J. Li, C. Ré, C. R. Aberger, and C. D. Sa. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *MLSys*, 2019.

[72] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: an efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX ATC*, pages 181–193, 2017.

[73] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *SoCC*, pages 390–404, 2017.

[74] Q. Zhang, R. Zhou, C. Wu, L. Jiao, and Z. Li. Online scheduling of heterogeneous distributed machine learning jobs. In *MobiHoc*, pages 111–120, 2020.

A Pipeline Reconfiguration

Reconfiguration introduces a much longer pause to the training process than recovering using RC. The goal of reconfiguration is to rebalance pipelines so they can withstand more failures as training progresses and continue to yield good performance. Reconfiguration also attempts to allocate more instances to maintain the cluster size. As shown in §3, asynchronous checkpointing is very efficient (but frequent restarting is not), and hence, Bamboo periodically checkpoints the model state. These checkpoints will not be used unless Bamboo restarts the training from a rare fatal failure (*i.e.*, too many nodes are preempted so that training cannot continue).

Reconfiguration Triggering. Reconfiguration is triggered immediately when (1) consecutive preemptions occur simultaneously and (2) Bamboo determines that there is an urgent need to rebalance the pipelines at the end of an optimizer step. To do (2), the workers retrieve the cluster state from `etcd`, allowing them to see how many preemptions have occurred and in which pipeline they have occurred. They can also see how many workers are currently waiting to join the next rendezvous.

There are two main conditions for triggering reconfiguration at the end of an optimizer step: (a) the cluster has gained enough new nodes to reconstruct a new pipeline, and (b) Bamboo has encountered many preemptions and is close to a critical failure in the next step (*e.g.*, encountering another preemption would cause us to suspend training), in which case we must pause the training to allocate more nodes.

Reconfiguration Policy. Bamboo attempts to maintain the pipeline depth P specified by the user. Therefore, our top priority at a reconfiguration is to reestablish a full pipeline of depth P. In this case, if we have had F failures and J ($> F$) nodes are waiting to join the cluster (*i.e.*, new allocations arrive as Bamboo runs on the "spare tire"), we can fully recover all pipelines to depth P. The remaining $(J-F)$ nodes are placed in a standby queue to provide quick replacement upon future failures. However, if the number of nodes joining is smaller than F, we may end up having a number of N nodes such that $N\%P \neq 0$. In this case, instead of creating asymmetric pipelines (which complicates many operations), we move some nodes into the standby queue and decrease the total number of data-parallel pipelines. A final case is that the number of nodes joining, together with those in the standby queue, can form a new pipeline, and in this case we add a new pipeline to the system. In all these cases, the redundant layers are redistributed among the set of nodes participating in the updated pipelines.

How to Reconfigure. Once a reconfiguration is triggered, each node must be assigned a new stage (with new layers, state, and redundancies); it also needs to figure out if it will need to send or receive model and optimizer state from other nodes. Whichever nodes hits the rendezvous barrier first decides the new cluster configuration and puts the decision on `etcd` for all other nodes to read. To minimize the amount of data sent in layer transfer, Bamboo transfers layers in such a way that each node can reuse its old model and optimizer state as much as possible.

B Support for Pure Data Parallelism

Bamboo supports pure data parallelism (without model partitioning). Due to space constraints, here we briefly discuss how it is supported. We use the same redundant computation strategy—Bamboo replicates the parameter and optimizer state of each node on a different node and uses these replicas as redundancies to provide quick recovery. For pure data parallelism, there is no bubble time to schedule RC. Eager FRC would be equivalent to overbatching (*i.e.*, each node processes its original minibatch plus a redundant minibatch). To reduce the FRC overhead and make RC fit into the GPU memory constraints, we over-provision spot instances (by 1.5×, in the same way as discussed in §5) to make each node process a smaller batch.

Enabling eager FRC doubles the batch size. However, it results only in a ∼1.5× increase in the computation time due to the parallelism provided by GPUs. This overhead can be effectively reduced by slightly over-provisioning ($1.5\times D$) nodes, increasing the degree of parallelism and decreasing the impact of overbatching. This enables us to run FRC eagerly without incurring much overhead (*i.e.*, <10%).

C Additional Experiments

C.1 Bubble Size

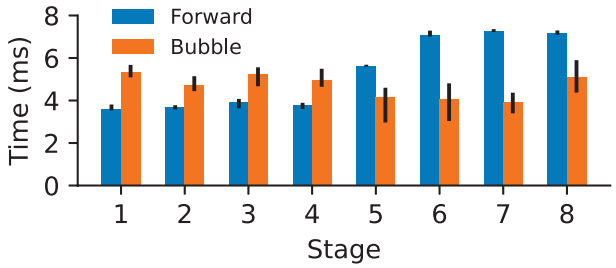


Figure 14: Comparison between bubble sizes and forward computations.

We measured the sizes of the pipeline bubble and forward computation of BERT with the same configuration as mentioned in Section 6, running on on-demand instances each with a single GPU. We manually inserted a barrier before each peer-to-peer communication, treating the time spent on the corresponding NCCL kernel as the bubble size. These results are reported in Figure 14.

To make memory evenly distributed across stages, more layers are placed on the last few stages. This explains the growth of forward computation. In this pipeline, for the first 4 stages, the bubble time is long enough to fit the entire FRC

(*i.e.*, the bubble at stage 1 should run the forward computation for stage 2). For the last 4 stages, the bubble time is shorter than the forward computation time—it can still cover ∼60% of its FRC. The rest of the FRC on these nodes is run in parallel with their regular forward computation, as discussed in §5.2.

C.2 Bamboo for Pure Data Parallelism

We ran two relatively small models such as VGG and ResNet using pure data parallelism with 8 workers (*i.e.*, we partition the data but not the model). For Bamboo, we similarly over-provisioned 1.5× additional workers. We implemented another baseline *Checkpoint*, which periodically checkpoints model state for each worker and restarts the worker on another node when its original node is preemption. We used the same global batch size for these models as reported in §6. The comparisons between Bamboo, *Checkpoint*, and on-demand training are shown in Table 6.

Note that our implementation of *Checkpoint* assumes that there is always a standby node that is ready to join and load the checkpoint (which is a unrealistic over-approximation of the allocation model on any spot market); as such, the training cost remains unchanged and its throughput is reduced as the preemption rate increases.

Model	System	Throughput	Cost ($/hr)	Value
ResNet	Demand	24.51	24.48	1.01
	Checkpoint	[12.26, 8.42, 5.03]	[7.34, 7.34, 7.34]	[1.67, 1.15, 0.68]
	Bamboo	[**21.22**, 18.31, 12.31]	[**10.56**, 10.09, 9.18]	[**2.01**, 1.84, 1.34]
VGG	Demand	144.28	24.48	5.89
	Checkpoint	[83.21, 67.21, 45.31]	[7.34, 7.34, 7.34]	[11.33, 9.15, 6.17]
	Bamboo	[**125.59**, 96.51, 73.73]	[**10.56**, 10.09, 9.18]	[**11.89**, 9.56, 8.03]

Table 6: Comparison between pure data-parallel training over on-demand instances, a checkpoint-based approach on spot instances, Bamboo on spot instances. For *Checkpoint* and Bamboo, we trained each model three times, and their results are explicitly listed in the form of [a, b, c] for the 10% (average), 16%, and 33% preemption rates, respectively.

As shown, Bamboo outperforms *Checkpoint* by **1.64**× and **1.22**× in throughput and value. Both *Checkpoint* and Bamboo deliver a higher value than on-demand training (by **2**× and **1.79**×).

We make two observations on these numbers. First, Bamboo incurs a higher cost than *Checkpoint* due to resource over-provisioning. However, as discussed above, *Checkpoint* assumes the availability of standby nodes. In practice, guaranteeing such availability requires over-provisioning as well, but we did not take this into account when calculating costs (because it is hard to know exactly how many nodes we should over-provision). Hence, the cost and value reported for *Checkpoint* are the *lowerbound* and *upperbound* of those that can be achieved by any practical implementation of a checkpoint-based approach.

Second, *Checkpoint* works much better for pure data parallelism than for pipeline parallelism (as discussed in §3). This is because recovering from a checkpoint in pure data-parallel training is much easier than pipeline-parallel training where a pipeline reconfiguration process is needed for each restart.

ONEWAN is better than two: Unifying a split WAN architecture

Umesh Krishnaswamy*, Rachee Singh*†, Paul Mattes*, Paul-Andre C Bissonnette*, Nikolaj Bjørner*, Zahira Nasrin*, Sonal Kothari*, Prabhakar Reddy*, John Abeln*, Srikanth Kandula*, Himanshu Raj*, Luis Irun-Briz*, Jamie Gaudette*, Erica Lan*

*Microsoft, †Cornell University

Abstract

Many large cloud providers operate two wide-area networks (WANs) — a software-defined WAN to carry inter-datacenter traffic and a standards-based WAN for Internet traffic. Our experience with operating two heterogeneous planet-scale WANs has revealed the operational complexity and cost inefficiency of the split-WAN architecture. In this work, we present the unification of Microsoft's split-WAN architecture consisting of SWAN and CORE networks into ONEWAN. ONEWAN serves both Internet and inter-datacenter traffic using software-defined control. ONEWAN grappled with the order of magnitude increase in network and routing table sizes. We developed a new routing and forwarding paradigm called traffic steering to manage the increased network scale using existing network equipment. Increased network and traffic matrix size posed scaling challenges to SDN traffic engineering in ONEWAN. We developed techniques to find paths in the network and chain multiple TE optimization solvers to compute traffic allocations within a few seconds. ONEWAN is the first to apply software-defined techniques in an Internet backbone and scales to a network that is 10× larger than SWAN.

1 Introduction

The large-scale commercialization of cloud computing led cloud providers to provision private wide-area networks (WANs). These initial deployments connected both datacenters and Internet peering edges of the cloud using a unified cloud WAN. For instance, Microsoft's cloud WAN, called the CORE (AS8075) network, interconnected Microsoft's datacenters and Internet peering edges. However, as the cloud workloads evolved, inter-datacenter traffic began to dominate, shrinking the capacity available for carrying Internet traffic to peering edges. In response, Microsoft built a second WAN to offload inter-datacenter traffic. This WAN, called the software-defined WAN or SWAN (AS8074) used software-defined traffic engineering (TE) and bandwidth brokering to achieve higher network utilization [14] than RSVP-TE [2] in CORE. Deployment of two cloud WANs (Figure 1), one for Internet traffic and the other for inter-datacenter traffic is an industry-wide trend with Google [17] and Meta [9] operating similar split-WAN architectures.

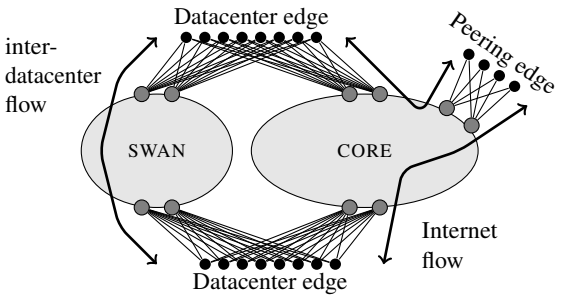


Figure 1: Before ONEWAN, there were two wide area networks, CORE (AS8075) and SWAN (AS8074). Datacenter edge connected to both networks, and peering edge only connected to CORE. Internet traffic was served by CORE and inter-datacenter traffic by SWAN. CORE used RSVP-TE and SWAN used SDN based traffic engineering.

Why is one WAN better than two? Maintaining two heterogeneous WANs specialized for inter-datacenter (SWAN) and Internet (CORE) traffic led to several operational challenges. First, the two-WAN architecture requires that datacenter edges connect to both SWAN and CORE routers (see Figure 1). The dual WAN connectivity from datacenter edges led to wasteful use of expensive network equipment and limited power supply. This problem was made worse by massive build outs of new datacenter regions and edge sites. Second, the split-WAN architecture makes capacity planning hard. At a given time, one WAN can be under-utilized while the other is over-utilized. Moreover, acquiring optical capacity for both WANs in every geographical region and building the required redundancy on each network, became prohibitively expensive. Finally, CORE and SWAN used routers with completely different protocol stacks. As a result, engineers were trained to configure, monitor and manage two distinct networks. Deploying new WAN sites took more time since different routers had to be deployed for the two networks.

In 2020, we observed a steady growth in Internet traffic from peering edges due to increased use of collaboration tools spurred by remote work (Figure 2). While the network capacity between Internet peering edges and the cloud WAN is scare and expensive, it was not in the purview of TE in the split-WAN architecture. Thus, it became important to engineer

the growing traffic on WAN-facing links from peering edges but the key protocol of the CORE network, RSVP-TE, was not up to this task as it had reached scaling limits in our network.

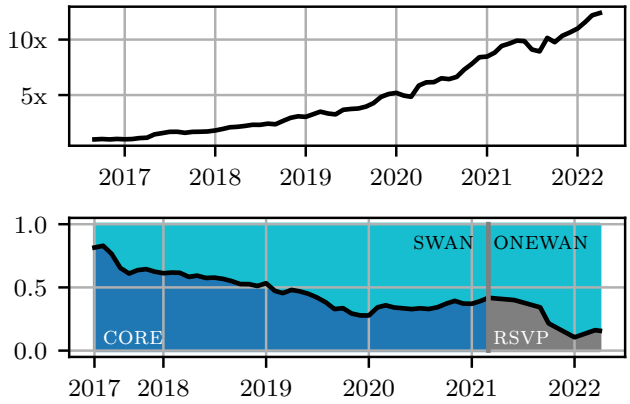


Figure 2: WAN traffic growth, the portions carried by CORE and SWAN when they were separate networks, and the portions of RSVP-TE and ONEWAN-TE traffic in the unified network.

Software-defined control in ONEWAN. Due to these operational challenges, we consolidated the split-WAN architecture into a unified ONEWAN. We decided to implement ONEWAN using SDN principles (like SWAN) instead of a standards-based approach (like CORE) for three main reasons. First, the key protocol of the CORE network, RSVP-TE, was reaching scale limits due the existing size of the CORE network topology. Second, RSVP-TE needed network-wide configuration changes which have a global blast radius. In contrast, we had deployed BLASTSHIELD to control the blast radius of faults in the SWAN network [22]. Finally, despite the earlier intention of using SWAN only for discretionary workloads, SWAN had evolved to carry mission-critical application traffic (*e.g.,* Azure, Bing, Office, and Teams) in addition to discretionary inter-datacenter workloads (*e.g.,* replication, backup). We measure our service level objectives (SLOs) as the daily average over percentage of successfully transmitted bytes in an hour. The SLOs of SWAN exceeded 99.999% for customer traffic, and 99.9% for discretionary traffic.

Challenges in evolving from SWAN to ONEWAN. In this work, we discuss the main technical challenges we overcame to unify the split-WAN architecture into ONEWAN (Table 1):

Increased routing table sizes. SWAN was responsible for routing datacenter prefixes only, which were few enough for all SWAN routers to run BGP and store the datacenter routing tables in the router memory. ONEWAN routers had to contend with Internet routing tables and it would be cost prohibitive to make every ONEWAN router hold the entire Internet routing table. Hence, ONEWAN assigned two roles to routers: (1) aggregation routers that hold full IP routing tables, and (2) backbone routers that act as forwarding only nodes that do not run BGP. We develop a new SDN function in ONEWAN called *traffic steering* on aggregation routers to encapsulate WAN packets with information needed by backbone routers to do TE without IP routing (§ 3).

Fast failure repair. One of RSVP-TE's strengths is *fast reroute*, which enables it to switch from primary to backup paths within milliseconds of a failure. This fast convergence is essential for performance-sensitive services like video streaming and virtual desktop over the WAN. *Local repair* is SWAN's equivalent function of detecting and repairing failed paths using agents that run on the routers. Improving convergence times in ONEWAN required significant enhancements and was an area of new learnings (§ 4).

Scaling TE optimization. ONEWAN has ten times the number devices of SWAN. The traffic engineering optimization techniques used in SWAN had to scale to a network size that is ten times larger. We developed scalable path and linear programming optimizations to deal with the increased scale of traffic engineering in ONEWAN (§ 5).

Estimating Internet traffic matrices. Accurate estimation of traffic matrices is crucial for engineering Internet traffic. Existing mechanisms for traffic matrix estimation fell short for ONEWAN since they did not contend with Internet-facing traffic. We developed a scalable pipeline for accurate traffic matrix measurement for ONEWAN (§ 6).

Hitless transition in the live network. Finally, the transition to ONEWAN was done in the live cloud network as it continued to carry user traffic. We devised techniques to enable both SWAN and CORE networks to undergo a hitless transition to ONEWAN (§ 7).

2 ONEWAN Architecture

The consolidation of SWAN and CORE networks into ONEWAN was a large undertaking that took years of engineering effort in planning, testing, implementing and verifying the new WAN architecture. In this section we motivate the design choices that led to the ONEWAN architecture.

Figure 3 shows a simplified view of ONEWAN with two datacenter regions and two edge sites. Datacenter regions are large campuses with routers at the root of the datacenter network connecting to aggregation routers in regional network gateways (RNGs). RNGs connect multiple datacenters in a geographical region with a maximum fiber distance under 100 kilometers. Backbone routers are present in all RNGs and additional transit gateways that are hubs for long-haul optical links. Peering edge routers at edge sites connect to aggregation routers in the same site. Aggregation routers connect to backbone routers in RNGs and gateways. ONEWAN traffic engineering (ONEWAN-TE) applies to inter-datacenter flows between datacenter regions, and Internet flows between edge sites and datacenter regions. The set of traffic engineered links consists of all backbone–backbone and aggregation–backbone links. The Clos interconnect between edge and aggregation

Challenge	Techniques used
Route scale increase	Only aggregation routers hold full IP routing tables. Controllers add routes for BGP next hops instead of BGP prefixes (§ 3).
Avoid costly new builds	Develop ONEWAN agents for four firmware versions to cover all existing routers. Interconnect CORE and SWAN with aggregation routers (§ 3 and § 4).
TE optimization scale increase	Traffic matrix-aware path computation. Optimize LP solvers (§ 5.1 and § 5.2).
Route convergence time	Fast local repair using diverse backup paths. Tunnel liveness probes that return to sender using controller routes (§ 4 and § 5.3).
Measuring real-time traffic matrix	Use IPFIX sampling with a high throughput pipeline. Anycast source-specific destinations determined from IPFIX flow records (§ 6).
Minimize risk of outages	Divide ONEWAN into geographies managed by separate controllers. Steering routes control what traffic is migrated ([22], § 7).

Table 1: Summary of ONEWAN challenges and our approaches for solving them.

routers is always intra-site or intra-campus, is built to high capacity, and hence is not a part of traffic engineering.

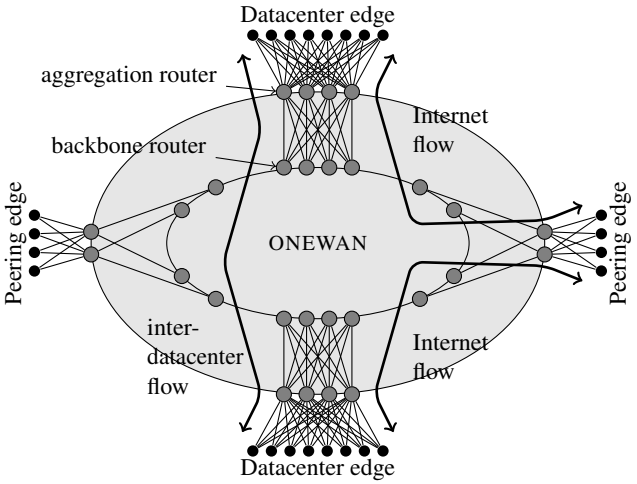
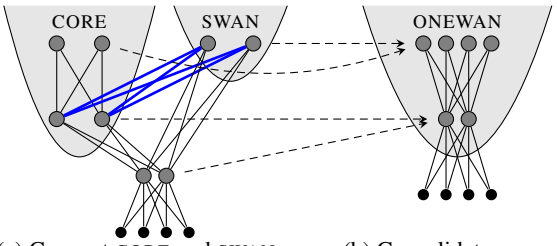


Figure 3: ONEWAN is a unified network that serves Internet and inter-datacenter flows using SDN TE. ONEWAN consists of aggregation routers at its boundary and backbone routers in its interior (gray dots). Datacenter and peering edge routers (black dots) connect to aggregation routers using a Clos interconnect. ONEWAN-TE applies to any inter-site flow between aggregation routers, be they inter-datacenter or between edge site and datacenter.

In-place conversion to ONEWAN. Since both CORE and SWAN networks carry mission-critical customer traffic, the unification of the two networks had to be done *in-place* with no visible impact to client performance. This goal necessitated incremental changes to both physical connectivity and network configuration to unify SWAN and CORE networks. The first step for the unification was to physically connect CORE and SWAN. Figure 4 shows new links that connect CORE aggregation routers to SWAN backbone routers. When CORE and SWAN were separate networks, they each had a set of aggregation routers. In the second step, we eliminated one set of aggregation routers and their links to save power. We merged the routing domains of the interior gateway protocol, which IS-IS [16] in CORE and SWAN, but did not merge the BGP autonomous systems of the two networks to allow each network to carry its original traffic.



(a) Connect CORE and SWAN (b) Consolidate aggregation

Figure 4: (a) Physically connect CORE and SWAN using aggregation routers (shown as thick lines). (b) Consolidate two sets of aggregation routers into one.

Leverage existing network hardware. Our goal was to repurpose the WAN routers in SWAN and CORE networks as opposed to building a clean-slate ONEWAN from the ground-up with an entirely new fleet of routing hardware to reduce the capital expenditure in the consolidation process. SWAN had $O(100)$ routers and $O(10^5)$ datacenter routes while CORE had $O(1000)$ routers and $O(10^6)$ Internet routes. To leverage existing hardware for the increased scale in ONEWAN, only aggregation routers run BGP with the full Internet and datacenter routing tables. This allows the remaining backbone routers to be simpler commodity switches with smaller routing table memory.

Choice of tunneling mechanism. ONEWAN uses Multiprotocol Label Switching (MPLS) [31] to transport IP packets in a tunnel across the backbone. We note that ONEWAN does not use any TE protocols like RSVP-TE or TE extensions in the interior gateway protocols like IS-IS. We use MPLS for its efficient implementation of label stack encapsulation on existing routers in SWAN and CORE. Our approach could be generalized to other tunneling abstractions though it is outside the scope of this work.

Router roles. Aggregation routers implement a key function in ONEWAN, called traffic steering, described in detail in § 3. Aggregation routers are also used as transit routers in ONEWAN-TE tunnels. This enables the large number of legacy

devices in CORE and SWAN to make full use of available bandwidth in either network. Edge routers are sources or sinks for traffic. They are either datacenter routers or peering routers. These routers continue to perform the same function before and after the consolidation of SWAN and CORE.

3 ONEWAN Traffic Steering

Routing in ONEWAN occurs in three parts (see Figure 5). First, a *steering route* on the aggregation router encapsulates IP packets entering ONEWAN. Second, backbone routers forward packets received from aggregation routers along traffic engineered tunnels. In the final step, the egress backbone router uses a segment routed [10] path to route packets to the network destination. ONEWAN-TE controller computes and ONEWAN router agents program traffic steering and engineering routes on routers. IS-IS updates the segment routes.

Steering routes. BGP on aggregation routers receives routes announced by BGP route reflectors or its clients and chooses a set of one or more equal-cost BGP next hops for each prefix. The BGP next hops are typically aggregation routers at network egress sites though they could also be endpoints a few hops beyond the network egress site in legacy portions of ONEWAN. All ONEWAN sites are assigned a static identifier called the *site label*. When BGP looks up the route for the next hop, the highest preference route is the steering route added by the ONEWAN-TE controller. The steering route pushes a stack of two MPLS labels onto packets entering the backbone. The top label in the stack is the egress site label. The bottom label in the stack is the segment routing node segment identifier (node SID) of the BGP next hop learned from IS-IS.

The egress site label refers to the backbone exiting site on the shortest path to the packet's destination. It is the same site as the egress aggregation router, though it can be different in legacy portions of ONEWAN. Ingress backbone routers use the egress site label to determine the set of traffic engineered tunnels to use. IP flows are weighted load balanced to a specific tunnel based on their 5-tuple and traffic class. Once the tunnel is selected, the ingress backbone router swaps the egress site label with the traffic engineered tunnel label.

The bottom label serves two purposes. First, it provides forwarding from the egress backbone router towards the endpoint. Second, it provides a fallback if no traffic engineered tunnel for the egress site is up due to failures. When the ingress backbone router has no operationally up traffic engineered tunnels to a particular egress site, ONEWAN agent automatically adds a route to pop the egress site label and forward the packet using the segment route for the BGP next hop node SID. The advantage of this design is that failures in the network are quickly and transparently handled by the routers without immediate intervention of the controller.

Steering route weights. The steering route sprays packet flows to connected backbone routers using unequal load balancing. Although aggregation routers are directly connected with equal capacity to backbone routers, each backbone router is not an equal choice for ingress. For example, a backbone router may have a longer path to the endpoint or may have less available bandwidth to an egress site. Latency increases in the former and congestion can occur in the latter. The ONEWAN-TE controller excludes backbone routers with the shortest path latency from the ingress aggregation router to the egress site exceeding the best latency by a threshold. It then calculates weights using single commodity maximum flow from the ingress aggregation router to the egress site. Weights are recalculated whenever the topology changes. Figure 6 illustrates the weight calculation for flows from aggregation router a to endpoint f. Both backbone routers b and c are used to spread the load because they have similar shortest path latency to the egress site. The weight to b is 33% because the maximum flow bandwidth of $a-b-T$ is proportionately less.

Why have two stages of traffic splits? ONEWAN calculates traffic steering splits using single commodity max-flow and calculates traffic engineering splits using priority max-min fairness optimization (§ 5.2) for the following reasons. We were concerned that the TE optimization and path computation algorithms may not scale to the full size of ONEWAN and operating on a subgraph of backbone routers would ease the scaling challenges. Moreover, aggregation routers are connected with high capacity links to backbone routers and so do not need traffic engineering. Finally, we wanted steering routes to be updated quickly in case of failures and did not want the updates to be slowed down by an optimization phase.

In hindsight, the two traffic splitting mechanisms in ONEWAN can be unified since our improvements to the TE optimization (§ 5) enable it to handle the full ONEWAN topology of backbone and aggregation routers. Aggregation routers are transits for ONEWAN-TE tunnels between CORE and SWAN devices and so have to be part of TE optimization. ONEWAN agents in aggregation routers react to network topology changes faster than the controller.

Segment routing at the egress backbone router. TE tunnels terminate at the backbone router instead of the aggregation router for a separate reason. Segment routing implementations on vendor routers only allow penultimate hop popping, meaning that the penultimate router must remove the node SID before delivering the packet to the intended node. Routers do not easily support popping a label stack. This necessitated at least one segment routed hop and is why the TE tunnel is between backbone routers. Support for ultimate hop popping would eliminate the last segment routed hop.

4 ONEWAN Agent and Local Repair

ONEWAN agents are responsible for installing routes provided by ONEWAN-TE controllers. ONEWAN agents also perform *local repair*. The local repair mechanism detects

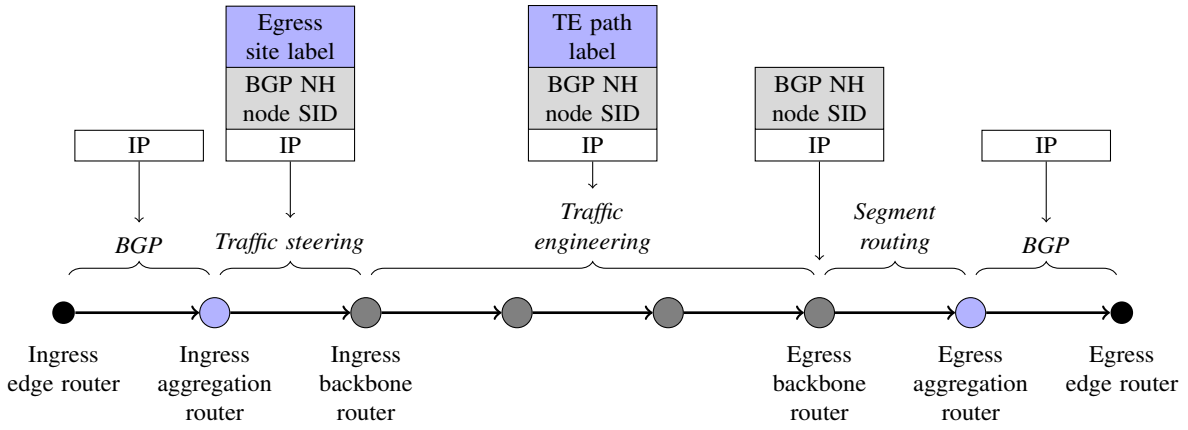
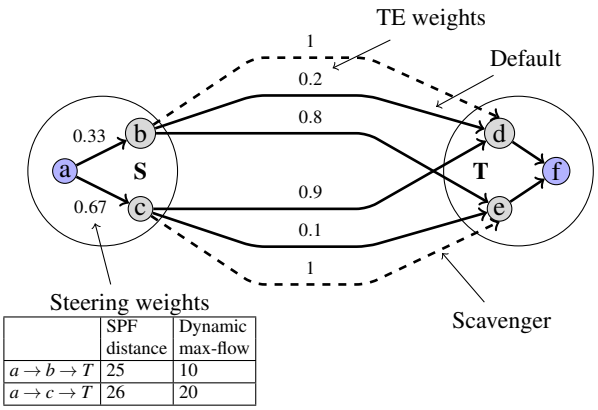


Figure 5: ONEWAN routing occurs in three parts. The first part is the steering route in the aggregation router, the second is the traffic engineered tunnel between backbone routers, and the third is the segment routed path from the egress backbone router to the network egress point. The traffic steering and engineering routes are added by the ONEWAN-TE controller.



	SPF distance	Dynamic max-flow
$a \to b \to T$	25	10
$a \to c \to T$	26	20

Figure 6: ONEWAN-TE example of flows from aggregation router a to endpoint f. Steering route load balances flows to selected backbone routers based on shortest-path distance and maximum flow from the backbone router to the egress site. The backbone routers perform full traffic engineering optimization.

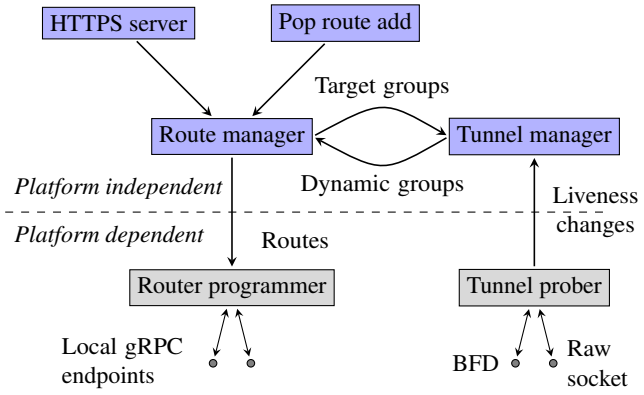


Figure 7: ONEWAN agent has platform dependent and independent components. The agent installs routes provided by ONEWAN-TE controller and switches off tunnels experiencing forwarding faults.

the forwarding state of tunnels and reprograms actions to send traffic on surviving tunnels thereby minimizing transient packet loss due to route blackholes or congestion.

The ONEWAN agent runs as a process on all aggregation and backbone routers, optionally inside a Docker container. It is supported on four different firmware operating systems. To support the heterogeneity of firmware, the agent is structured as separate platform-independent and platform-dependent components, with well-defined APIs between them. The platform-independent portion has the bulk of the complexity in the agent. Figure 7 shows the agent organization.

Route programming. ONEWAN agents communicate with ONEWAN-TE controllers using an HTTPS server; no routing protocol is required. We use OpenFlow [28] match actions and groups to represent routes. Groups represent the set of traffic steering and engineering tunnels originating at ingress routers, tunnel weights for unequal load balancing, traffic class to indicate what type of traffic the tunnel is meant for, whether the tunnel is primary or backup, and attributes for probing tunnel liveness. Transit routes use unary actions without groups.

The *route programmer* implements dynamic route operations through an internal gRPC or equivalent connection provided by the router firmware. It converts groups to weighted cost multipath (WCMP) next hops with 32 members and duplicates each tunnel in proportion to its weight.

The *target group* is the set of tunnels received from the controller and the *dynamic group* is the set of tunnels that are alive, with original weights redistributed among them. When a group has no primary tunnels alive, the backup tunnels are elevated to primary. A tunnel is associated with a traffic class, default matching any traffic class. When a group has no tunnels of default traffic class alive, the tunnels of the next traffic class are modified to be the default traffic class. In Figure 8, the target group has six tunnels A-F. The backup tunnels E and F become primary when A and B fail. If A comes back up, it is the sole primary default tunnel. We explain how ONEWAN-TE calculates diverse backup and class-aware tunnels in § 5.2 and § 5.3.

If no tunnels are alive, the agent replaces the dependent egress site label routes with pop-and-forward (on a backbone router) or removes the dependent BGP next hop steering routes (on an aggregation router). Part of agent initialization is the creation of pop-and-forward routes for the entire allocated egress site label space, to cover any stray traffic received.

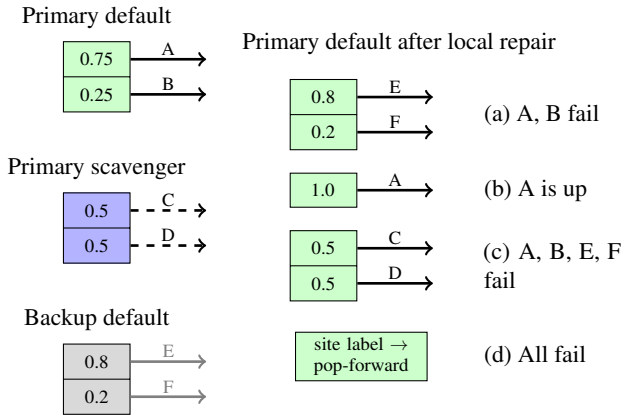


Figure 8: Local repair. The ONEWAN agent automatically adjusts the composition and weights of tunnels based on liveness. (a) If primary default tunnels A and B fail, then backup tunnels E and F become primary. (b) If A comes back up, it is the sole primary default tunnel. (c) If only scavenger tunnels C and D survive, they become primary. (d) If all tunnels fail, the site label route is replaced with a pop-and-forward action.

Tunnel probing. Aggregation routers use single-hop tunnel probing using Bidirectional Forwarding Detection [21] to check the links to backbone routers. Ingress backbone routers perform end-to-end tunnel probing using a labeled self-ping mechanism via a raw socket. Tunnel probes use the same routes as data packets in the forward direction.

Tunnel probes in SWAN relied on IS-IS in the return path. This caused backup tunnels to fail even though they did not use failed links in the forward path because the probe return path was affected by IS-IS route convergence. Without primary and backup tunnels, the agent removed the controller route and traffic reverted to IS-IS. Client traffic experienced IS-IS convergence times and congestion losses as long as the tunnels were still down. This significantly degraded user experience. Therefore, probes in ONEWAN return from the tunnel destination to the tunnel source using controller routes. The return hops are the reverse of the forward hops and thus do not share fate with links unrelated to the data tunnel. In Figure 9, the probe packets return on $d-b-a$, not $d-c-a$ even though the latter is shorter. The ONEWAN-TE controller reuses available data tunnels in the reverse direction when possible, and creates new tunnels otherwise.

Multiple probe packets can be in flight and the probing interval is independent of the path round-trip time. A loss of a configured number of probes marks a tunnel down, and a successful probe marks a tunnel up. We send probes at 100ms

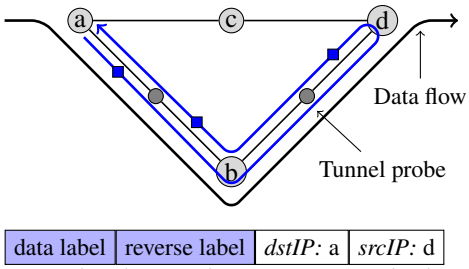


Figure 9: Tunnel probes use the same routes as the data packets in the forward direction and return from the destination using the same links to avoid false failures. The transit routes are programmed by the controller and agents of transit routers.

intervals and mark the tunnel down after loss of 3 probes. Fault detection time is thus 300ms plus the distance to the fault which in the worst case is the tunnel latency.

Local repair. In the split-WAN architecture, Internet traffic was handled by RSVP-TE in the CORE network. Standards-based RSVP-TE in the CORE network implements a fast reroute (FRR) [29] mechanism that allows it to recover from link failures in $O(10)$ ms. Fault detection time is the distance to the fault, which in the worst case is the link latency, plus a small delay for the optical transponder to notify the router. FRR switches to precomputed bypass tunnels at the point of fault and runs in or near the line card network processing unit. Switching times are a few milliseconds.

In contrast with FRR, ONEWAN's local repair happens at the ingress router of the tunnel not the point of fault. Faults in the first hop link are detected by interface down events and faults in subsequent hops are detected by lost probes. Repair is initiated in the route processor and subject to greater inter-process communication and scheduling delays. The route programmer modifies the WCMP in place to decrease the number hardware writes performed at the time of repair. As a result, ONEWAN convergence time is under one second. Although slower than FRR, ONEWAN meets the convergence time requirements of video streaming, video conferencing and other interactive applications currently served by the network. ONEWAN tunnel probes also validate forwarding because they exercise the routes used by data packets. The backup tunnels are chosen based on diversity and residual bandwidth and hence experience less transient packet loss due to congestion (described in § 5.3).

Route programming. ONEWAN is divided into geographical regions and regional ONEWAN-TE controllers program routes on devices in their region [22]. All routers program steering routes in parallel, in a single phase. The pop-and-forward routes on the backbone routers eliminate the need to synchronize programming steering and TE routes.

TE routes are also updated in parallel, using three phases with a barrier between each. A *make-before-break* sequence ensures that no route blackholes or loops form during programming. The set of routes for all routers is logically divided into two sets: the transit and tunnel egress routes T, and the

Metric	ONEWAN-TE
Per-device steering + TE routes	$O(10^3)$
Network level tunnels	$O(10^4)$
Per-device FIB update (p95)	2.0 sec
Network level FIB update (p95)	3.7 sec

Table 2: ONEWAN-TE scale in terms of routes, tunnels, and update times per device and for the entire network.

tunnel ingress routes G. Since the G routes depend on the T routes, make-before-break ensures that traffic cannot enter a tunnel until each subsequent hop has been programmed. An important property of ONEWAN-TE route generation is that the label spaces between successive iterations do not overlap, except where the paths they represent are identical. The label range is large enough to allocate unique labels in the worst case.

The phases of replacing the routes of iteration n, $T_n \cup G_n$, with routes of iteration $n+1$, $T_{n+1} \cup G_{n+1}$, are as follows:

- The initial state in all routers is $T_n \cup G_n$.
- $T_n \cup T_{n+1} \cup G_n$ is sent to agents which perform their route update and report success or error.
- $T_n \cup T_{n+1} \cup G_{n+1}$ is programmed. During this phase there may be a mix of G_n and G_{n+1} routes in the network, but all the T routes they depend on will be present.
- $T_{n+1} \cup G_{n+1}$ is programmed.

Traffic shifts in the second phase. Since a phase completes in under 4 seconds, a moderate scratch capacity avoids transient congestion. We reserve 15% scratch capacity to handle transients from programming and traffic microbursts. If an agent reports an error or connection is lost, the controller rolls back to the initial state in a single phase. Any inconsistencies after the rollback are corrected by the ONEWAN agent local repair.

4.1 Evaluation

Route scale. The number of traffic engineering tunnels was a significant scaling issue with RSVP-TE in CORE. Large number of RSVP-TE tunnels increased network convergence time after link failures and exceeded hardware resources in older aggregation routers. Table 2 shows that ONEWAN-TE uses 10 times fewer tunnels than RSVP-TE. When ONEWAN-TE optimizes, it simply reserves more bandwidth in existing tunnels, or creates new and destroys unused tunnels. On the other hand, RSVP-TE signals new tunnels with incremental bandwidth reservation, which it combines less frequently. Second, RSVP-TE requires a full mesh of label switched paths between nodes, but ONEWAN-TE only creates tunnels for nodes within a geography, and inter-geography flows reuse the intra-geography tunnels.

Steering routes scale with number of endpoints, and traffic engineering routes scale with number of backbone nodes. Both forwarding information bases of routes (FIB) have small sizes even for a large network. Time to update the FIB is affected by the round-trip distance between the controller and router, and the number of routes in the FIB (see Figure 10).

Class based forwarding. An objective of ONEWAN-TE is to use underutilized links on longer paths for replication or backup traffic which are marked as scavenger traffic class but use diverse shortest paths for best-effort and higher traffic classes. ONEWAN agent installs the egress site label route with an intermediate policy lookup that is indexed by traffic class to change from default class WCMP to a class specific WCMP. Figure 11 shows that ONEWAN-TE assigns 55% of scavenger traffic to longer paths. Differentiated paths decrease scavenger drops due to microbursts in higher traffic classes. When best-effort and scavenger queues use weighted round-robin queue scheduling, differentiated paths also decrease best-effort transient drops due to scavenger microbursts. Figure 12 shows the successful transmission rate SLO for best-effort and scavenger traffic classes for a 3-month period in 2022.

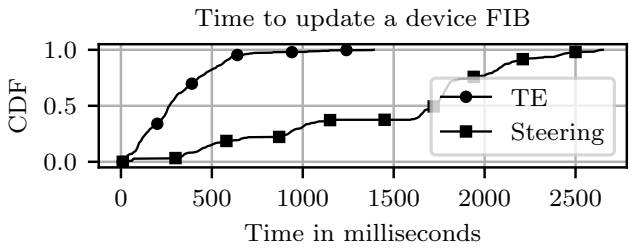


Figure 10: Per-device FIB update times for steering and TE routes.

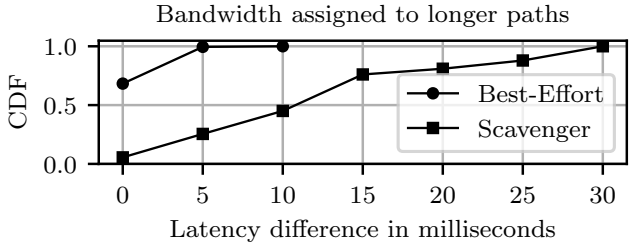


Figure 11: Class based forwarding uses paths for scavenger traffic class that are different from best-effort and higher classes. Longer paths carry 55% of scavenger bandwidth.

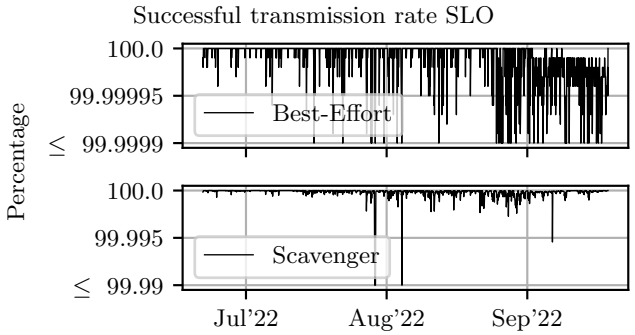


Figure 12: Successful transmission rate SLO for best-effort and scavenger traffic classes for a 3-month period in 2022.

5 Traffic Engineering Optimization

The ONEWAN TE optimizer has two inputs – predicted traffic matrix (TM) (described in § 6) and dynamic topology. A traffic trunk is an aggregate traffic flow from a source backbone router to a destination site for a specific traffic class, and the traffic matrix is a collection of predicted bandwidths for traffic trunks. The dynamic topology consists of sites, nodes and links. Nodes and links have tens of different attributes, including interface addresses, device role, link operational bandwidth, bandwidth reserved for RSVP-TE, link metric, whether a link or node should be avoided due to maintenance activity, and link reliability information. Each node is associated with a site, and all nodes in a site are equivalent destinations for a traffic trunk destined to the site. The TE optimizer operates in two phases: path computation phase and optimization phase.

In the path computation phase, we perform online computation of paths on the dynamic topology for all traffic trunks using efficient path finders (§ 5.1). We compute enough paths to enable the optimization phase to have adequate choices when allocating traffic.

In the optimization phase, the priority fairness optimization solver (§ 5.2) allocates traffic trunks to paths using the path formulation of multi-commodity flow problem. The TM is divided based on the traffic class of trunks and each traffic class is optimized differently. The priority fairness solver allocates demands of high priority trunks (best-effort and higher traffic classes) with the objective of minimizing the cost-bandwidth product. In contrast, it allocates low priority scavenger traffic trunks with the goal of minimizing maximum link utilization. The rationale to minimize maximum link utilization objective for low priority traffic is to decrease congestion drops in the scavenger class from microbursts in higher traffic classes, and to allow bandwidth broker [14] to serve more requested bandwidth using under-utilized links. High priority users expect the best latency the network can offer.

The priority fairness solver chains four solvers, max-min fairness, minimize cost, minimize maximum utilization, and diverse path, in different combinations based on traffic classes. The inputs to all solvers are paths computed in the first phase, TM, and upstream solver constraints.

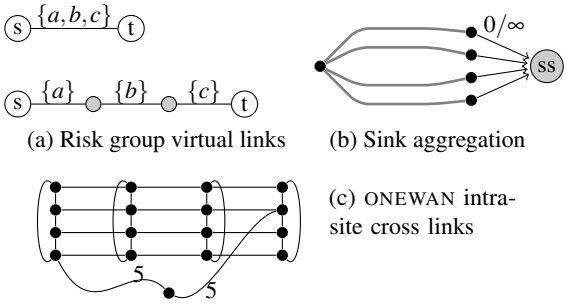
5.1 Path computation

Path finders in ONEWAN implement techniques for exploring paths in the network topology. Over the years, we have developed many path finders. Today, we accumulate paths from two paths finders in ONEWAN— penalizing and maximum flow path finders. The union of paths from the two finders has a mix of diverse shortest and maximum flow paths.

The penalizing path finder returns risk diverse paths with policies to never reuse a *risk group* for paths between the same source–sink pair or reuse only if necessary. Risk group is an identifier for shared optical infrastructure used by two or more links. A link can have zero or more risk groups. The finder uses Dijkstra's algorithm by setting link weights to penalties for risk groups used in previous shortest paths. Penalties are either infinity or a large value like the sum of all edge weights. The first path returned by the solver is the shortest path. Each subsequent path is the shortest path in the graph with modified weights. The finder operates on a graph where the risks groups have been expanded into virtual links (see Figure 13 (a)).

The maximum flow path finder uses maximum flow algorithms [11] and converts the augmenting paths into network paths. Link bandwidth is set to the reservable bandwidth for ONEWAN-TE. Link distance is not used in this finder.

Recall from § 3 that traffic engineered routes are not to individual routers, but to a group of routers in the destination site. Therefore, ONEWAN path finders compute paths from source nodes to destination sites using the technique of *sink aggregation*. In sink aggregation, we add a super sink node (ss) to the graph and connect sink nodes to it using directed edges of zero weight and infinite bandwidth (see Figure 13 (b)). Sink aggregation reduces the number of paths that ONEWAN-TE needs to allocate traffic on. Work is further reduced by computing paths only for source-sink pairs in the traffic matrix, instead of all pairs of nodes. This is called *TM-aware source sinks*. Figure 14 shows that these techniques reduce the path counts by a factor of 30. The resulting speed-ups extend into the optimization phase because fewer paths mean fewer columns in the linear programming constraint matrix. The average time to compute paths in ONEWAN is only 5 seconds.



(a) Risk group virtual links (b) Sink aggregation

(c) ONEWAN intra-site cross links

Figure 13: Path computation: (a) penalizing path finder expands risk groups to virtual links, (b) sink aggregation finds paths to all nodes in a site in the same exploration using a super sink, (c) cross-links within a site complicate *k*-shortest path exploration.

Why does ONEWAN not use k-shortest paths? Path computation using maximum flow algorithms works better than *k*-shortest path algorithms on the ONEWAN topology. This is because ONEWAN sites have a Clos or cross-link structure that requires *k* to increase exponentially with the number of inter-site hops (see Figures 3 and 13 (c)). There is dissimilar link bandwidth and cost for nodes in a site that make optimizations difficult. Switching from *k*-shortest paths to maximum flow algorithms gave a significant boost in path choice and speed.

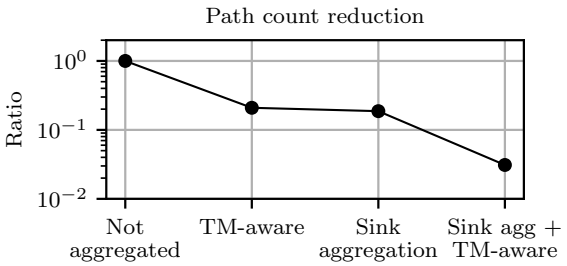


Figure 14: Path count reduction when paths are computed for traffic matrix aware source sinks and aggregated by sink nodes in a site.

5.2 Priority fairness solver

The priority fairness solver assigns priorities to traffic trunks based on their traffic class. Best-effort and higher traffic classes map to priority 0 (highest) and scavenger traffic class trunks are priority 1. Cloud network operators set a *committed data rate* for each priority. Committed data rate is the minimum percentage of link bandwidths guaranteed to a priority. If excess network bandwidth is available for use, traffic trunks receive more than the guaranteed allocation in priority order. By pre-committing link bandwidths based on priority, we ensure that lower priority trunks do not starve in the network.

The priority fairness solver runs in two stages. In the first stage, starting with the highest priority, it sets link bandwidths in proportion to the committed rate of the priority. This excludes the bandwidth committed to lower priorities, guaranteeing that lower priority demands are not starved for bandwidth. The fairness solver then allocates bandwidth for demands in this priority. It repeats this process for each lower priority demand set and accumulates all the downstream solver results into a single priority-aware solver result.

During the second stage, the solver walks through each set of demands in a priority and identifies fully satisfied demands. These demands are marked as frozen. For unsatisfied demands, it adds the credit allocated in the first stage back to the available link bandwidth. Adding the credits back to the links enables these demands to be run again as if none were allocated. In the first stage the demands were limited to available bandwidth after excluding committed rates of lower priorities. In this stage, unused committed rates are available and can be used for unsatisfied or partially allocated demands.

Solver chaining. ONEWAN-TE has multiple objectives. It allocates priority 0 traffic to achieve max-min fairness and minimum cost using a diverse set of network paths. It allocates priority 1 traffic to achieve max-min fairness and minimize the maximum link utilization. The priority fairness solver achieves these TE objectives by chaining multiple solvers to compute traffic allocations. Traffic allocations from upstream solvers in the chain constrain the solution space of subsequent solvers, achieving one TE objective per link of the solver chain. Solver chaining breaks ONEWAN's TE problem into reconfigurable linear programming (LP) steps. The priority fairness solver (see Figure 15) uses four solvers. For brevity,

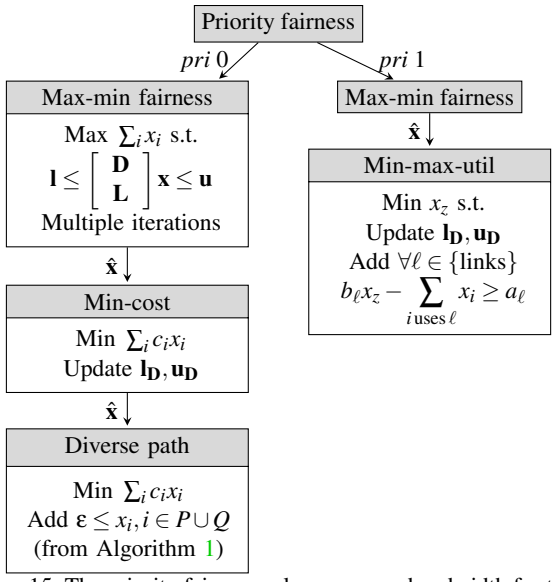


Figure 15: The priority fairness solver reserves bandwidth for traffic classes based on their priorities. It then invokes the chain of max-min fairness, min-cost and diverse path solvers for high priority traffic. High priority traffic does not consume bandwidth reserved for low priority traffic. After allocating high priority traffic, it invokes the max-min and min-max util solvers for lower priority traffic. Unused bandwidth after both high and low priority chains is made available for unmet demands in the second round of priority fairness solver.

we omit a detailed discussion of the optimization problems in the solvers, previously described in [3, 14, 19, 27]. $\mathbf{x}$ has an element for each path and its solution $\hat{\mathbf{x}}$ is the allocation of bandwidths requested by trunks to paths. The solvers share a common core of demand and link constraints ($\mathbf{D}, \mathbf{L}$).

Chain of solvers. The *Max-min fairness solver* optimizes throughput using approximate max-min fairness. It uses multiple iterations of maximizing throughput with adjustments to demand constraint bounds. *Min-cost solver* uses the solution of max-min fairness to minimize the dot product of cost and allocations. For ONEWAN-TE, cost c_i is path metric, which is the sum of link metrics. It uses the solution of the max-min fairness to adjust the bounds in demand constraints. *Diverse path solver* solves the same objective but with diverse path constraints described in § 5.3. Priority 1 trunks are optimized with max-min fairness using the residual link capacity after deducting the allocations of priority 0. *Min-max-utilization solver* uses the upstream solution to adjust the bounds in demand constraints. To minimize maximum utilization with a linear objective, it adds a new variable x_z that represents maximum utilization, and utilization constraints where a_ℓ is previously allocated link bandwidth and b_ℓ is link capacity.

LP solvers. ONEWAN integrates two LP solvers, CLP [5] and GLOP [12]. They provide a 5× speedup over the original solver used in SWAN. We achieve additional speedup by reducing the constraints sent to the solvers: the use of destination sites reduces the number of paths, and therefore columns in the constraint matrix are reduced 7-fold (§ 5.1). The biggest

gain in constraint reduction was achieved by pruning small traffic trunks that are under 500 Mbps. This decreased the constraint matrix rows and columns by factors of 5 and 7 respectively. The small trunks represent only 2.5% of network traffic (described in § 6.2) and get optimized within five minutes of growing large. Practicality of using LP solvers also depends on settings: dual simplex is superior to primal simplex for our LP models. Furthermore, pure cycling is possible as the max-min fairness models have high degeneracy — many columns are the same because the non-zero elements in the constraint matrix and the cost vector coefficients are 1. Cost perturbation is used to jiggle the values out of cycles.

5.3 Diverse path solver

A collection of paths is *risk diverse* if the intersection of risk groups of all paths in the collection is an empty set. ONEWAN applies diverse path protection to best-effort and higher traffic classes. In the ONEWAN-TE solver chain, the diverse path solver computes diverse paths using the primary paths from the min-cost solver. The goal of the diverse path solver is to decrease transient congestion in the interval between local and global repairs. ONEWAN agents perform local repair at the ingress router and ONEWAN controllers perform global repair based on the new topology. Spreading traffic on multiple paths without considering risk diversity can result in transient packet loss due to route blackholes or congestion since local repair is forced to use IS-IS routes.

The diverse path solver uses a greedy weighted set cover to find the minimum weight set of diverse paths that protects the shared risks in primary paths. Algorithm 1 outlines diverse path constraint generation. The WEIGHT function is configurable. Diverse paths can be configured to not exceed the primary path latency by a configurable jitter threshold, or diverse paths with more residual bandwidth can be preferred. We define jitter as a normalized ratio of path latencies, $(max - min)/\sqrt{min}$. Since high priority users expect the best latency in non-failure conditions, diverse paths with excess jitter are excluded. The diverse path solver re-solves the minimum cost objective with diversity constraints (see Figure 15). Diverse paths usually get the smallest possible weight since using them pulls the solution away from the minimum cost solution.

Figure 16 shows the percentage of total traffic on risk diverse paths. Without diverse path solver, any diversity is purely accidental, and was measured at 10%. When the agent only supported primary tunnels, jitter threshold of 5 was used to not adversely impact the latency of flows using the diverse paths. This constrained the choice of diverse paths and risk diverse traffic percentage was 75%. Once the agent implemented primary-backup tunnels, the jitter threshold was not required, and the protected traffic increased to 99%. The remaining 1% is due physical constraints on diversity of optical circuits.

Algorithm 1 Adds diverse path constraints

```
1:  procedure DIVERSITYCONSTRAINTS(P,U)
    P is the set of primary paths.
    U is the set of computed paths.
2:      risks ← SHAREDRISKS(P)
3:      ss ← SOURCESINKS(P)
4:      candidates ← PATHS(U,ss) \ P
5:      for all q in candidates do
6:          protect[q] ← risks \ SHAREDRISKS(q)
7:          residual ← residual bandwidth of q
8:          jitter ← latencies of q and P
9:          w[q] ← WEIGHT(residual, jitter)
10:     end for
11:     Q ← WEIGHTEDSETCOVER(protect, w)
12:     for all i in P∪Q do
13:         ADDCONSTRAINT(ε ≤ x_i)
14:     end for
15: end procedure
```

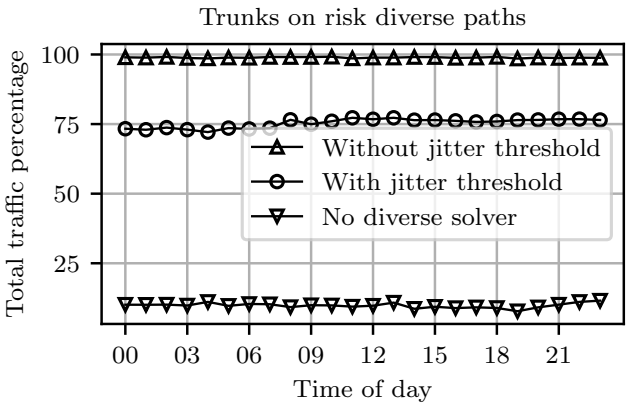


Figure 16: Percentage of total traffic on risk diverse paths in a typical day. 10% of the traffic is protected without diverse path solver, 75% with jitter threshold, and 99% without jitter threshold.

6 Measuring WAN Traffic Matrices

The traffic matrix (TM) is a key input to the traffic engineering optimizer. A traffic trunk is an aggregate traffic flow from a source backbone router to a destination site for a specific traffic class. There are four primary traffic classes in ONEWAN: voice, interactive, best-effort, and scavenger. The WAN TM is a collection of traffic trunks and bandwidths for each trunk. A trunk's bandwidth can be a requested value for discretionary traffic, a measured value for Internet or non-discretionary traffic, or a prediction based on measured values. Bandwidth broker is a service that measures discretionary traffic at sending hosts and aggregates it into trunk-level requested bandwidth. The input to the traffic engineering optimizer is the complete traffic matrix that is a combination of the requested TM and predictions based on the measured TM.

Measured TMs are computed by sampling packets as they enter the WAN. SWAN sampled traffic using sFlow [30] at

backbone routers. On the other hand, ONEWAN sampled traffic using IPFIX [33] at aggregation routers. The sampling point and technique were dictated by which router exported the fields required for identifying traffic trunks. Flow record attributes such as sampling router address, input and output interfaces, BGP next hop, traffic class, and packet 5-tuple are used to identify the traffic trunk. ONEWAN required a high throughput data pipeline to process flow records from a larger number of devices at a faster sampling rate compared to SWAN. Since ONEWAN-TE re-optimizes traffic allocations every 5 minutes, TMs are measured at short, minute-level timescales.

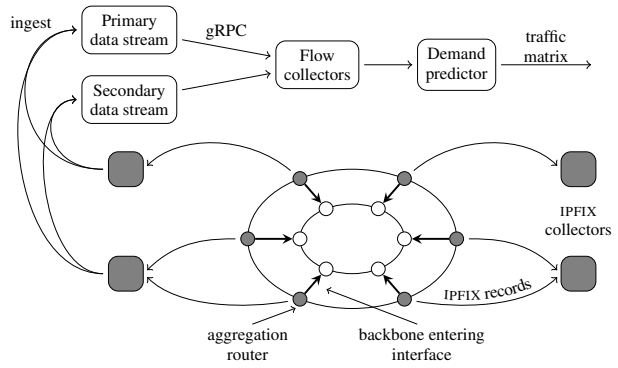


Figure 17: ONEWAN's high throughput data pipeline for TM measurement. Aggregation routers export flow information to IPFIX collectors which process and store them in redundant data streams. Flow collectors query the data stream to compute bandwidths of traffic trunks. The demand predictor predicts the complete TM.

Challenges in measuring anycast traffic volumes. A large fraction of the traffic in CORE is anycast — traffic towards destination prefixes advertised by multiple sites in the WAN. Anycast traffic is routed to the router nearest to the *source* router using standard BGP path selection rules. Correctly identifying the destination of an anycast traffic trunk requires knowledge of the BGP route lookup result, which is different for each source. One way of solving this in software is to acquire copies of the routing tables of all aggregation routers and replay the route lookup. To avoid the overhead of copying tables and replaying routes, ONEWAN leverages IPFIX sampling where routers store the route lookup result in the BGP next hop attribute of IPFIX flow records.

IPFIX data pipeline to measure traffic matrices. In Figure 17, aggregation routers sample one in 4,096 IP packets, and export flow records using anycast to the nearest IPFIX collector cluster. The IPFIX collectors write the statistics to redundant data streams. Flow collectors query the data stream for flow records from aggregation routers to the connected backbone routers and build a traffic matrix. They also identify and tag discretionary traffic in the measured TM to help in combining with the requested TM. The demand predictor aggregates traffic matrices from all collectors and uses linear regression and autoregressive moving average models on measured TMs to make predictions.

6.1 Error correction in measured TMs

We calculate the accuracy of traffic matrices measured using aggregations of IPFIX data against packet counters like interface counters, output queue counters, and RSVP-TE used bandwidth counters. Early versions of the IPFIX data pipeline had issues like invalid or missing attributes in flow records *e.g.,* missing BGP next hop attribute for IPv6 flows, or incorrect egress interface in certain flows. We detected and fixed such issues over time.

Flow record exporters use UDP, even though it is unreliable, because it needs less router resources. Hence, it is important that the data pipeline does not lose flow records due to traffic surges, unequal load distribution, or systemic or fault induced capacity crunch. The first generation of the data pipeline was lossy. The second generation of the pipeline used gRPC streaming from the data stream to the flow collectors (see Figure 17) and was not lossy. While operating with the first generation, we developed an error correction that gave similar results as the second generation. Since there is potential of failures in any pipeline, we outline the error correction technique used to improve reliability of our TM estimation.

We define the *interface error rate* as the ratio of input interface bit rate on the backbone entering interface measured by SNMP and calculated by IPFIX. Values greater than one indicate underestimation, and less than one indicate overestimation by IPFIX. The flow collector continuously calculates interface error rate, and scales the IPFIX measured bandwidths of individual traffic trunks in proportion to the interface error rate. The flow collector scales the bandwidth of each trunk separately based on its input interface error.

6.2 Traffic matrix characteristics

Figure 18 (a) shows typical diurnal and weekly traffic patterns seen in service provider networks. Interestingly, the number of traffic trunks in ONEWAN is 8× the trunks in SWAN due to the doubling of source backbone routers, and increase in destination sites in ONEWAN.

In Figure 18 (b), 82% of trunks are under 100 Mbps and represent 1.3% of total traffic in WAN. The small trunks are uniformly distributed across the network, consist of many flow types, and do not pathologically contend for specific links. We take advantage of this distribution to speed up traffic engineering optimization by letting the smallest trunks go unengineered.

In Figure 18 (c), trunk distance is calculated based on fiber distance of the shortest path from source to destination. The largest traffic trunks have source and destination close to each other. 25th-percentile by distance is 95% of total traffic. We use this to define ONEWAN geographies such that 75% of the WAN traffic is intra-geography.

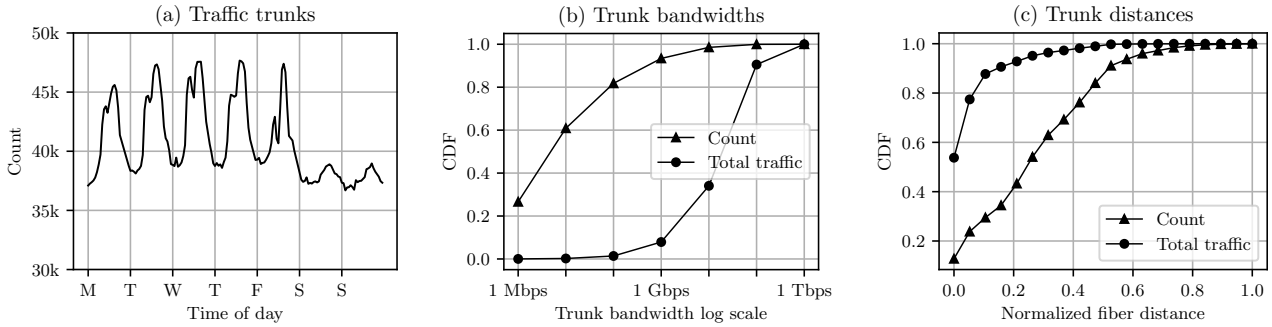


Figure 18: Traffic matrix characteristics for a typical week: (a) count of trunks by time of day, (b) distribution of trunks by bandwidth, (c) distribution of trunks by normalized fiber distance from source to destination.

7 Operational Experience

In this section we share lessons learned from the unification of SWAN and CORE networks into ONEWAN.

Traffic migration. We migrated traffic from RSVP-TE to ONEWAN-TE in the production network. So, it was important to perform this migration in a hitless manner. We used steering routes to migrate traffic from a specific set of source routers to another specific set of endpoints. We began by migrating traffic that was sourced and destined within the same geography because the configuration changes required for this are local to that geography. We used RSVP-TE maximum reservable bandwidth configuration on routers to control the percentage of link bandwidth made available to ONEWAN-TE. Initially, ONEWAN-TE had a small fraction of link bandwidth, sufficient to initiate traffic migration. We gradually increased ONEWAN-TE traffic and reduced the RSVP-TE maximum reservable bandwidth until RSVP-TE configuration could be entirely removed from the router.

Merging IS-IS routing domains. Merging the SWAN and CORE IS-IS domains posed a high risk for SWAN routers, whose IS-IS would observe a 7-fold increase in link state advertisements after the merge. IS-IS uses backoff timers to pace the shortest path first (SPF) execution. The purpose of backoff timers is to react quickly to the first few events but under constant churn, slow down to prevent the router from collapsing. The interactions of TE extensions still using IS-IS with interoperability issues caused IS-IS SPF to be persistently in backoff state slowing convergence. This prompted the design change to make ONEWAN-TE tunnel probing completely independent of IS-IS by using controller routes to return from the tunnel destination to the tunnel source (§ 4).

Cost savings with ONEWAN. ONEWAN brings the benefits of SDN TE to Internet traffic which was previously managed by RSVP-TE in the CORE network. SDN TE is known to make efficient use of network capacity, thereby reducing the need for capacity augmentation in the network. In Figure 19, we compare projected capacity augments needed with ONEWAN vs. SWAN + CORE to meet the organic traffic growth. We expect to reduce capacity augments by 10% of current installed capacity in the next few years, which is of significant value for the size of our network. This does not include savings on inorganic growth like building new regions.

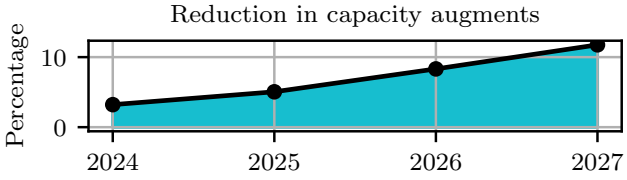


Figure 19: Reduction in capacity augments with ONEWAN compared to SWAN + CORE as a percentage of current installed capacity.

8 Related Work

We are the first to apply SDN techniques to replace RSVP-TE in an Internet backbone. SDN based traffic engineering was first used in inter-datacenter networks [9, 14, 15, 17, 20, 24, 25]. An inter-datacenter network has fewer points of presence, simpler scaling requirements, and easily built from scratch. We show that the challenges of scalability, reliability, feature parity, and migration can be overcome and replace RSVP-TE in an Internet backbone. This leads to unification of the cloud network with a single SDN controlled backbone.

SDN controllers for Internet peering have been discussed in [4, 7, 13, 34, 35, 37]. They tackle an important adjacent problem of Internet traffic engineering at the peering edge. These controllers enable performance-aware egress peer selection and inbound traffic engineering between autonomous systems. Our work focuses on the dynamic path selection and load balancing of this traffic between the peering edge and the end host in a datacenter, as it transits the backbone within the autonomous system. Microsoft peering edge uses a similar SDN controller that is outside the scope of this paper. ONEWAN controllers measure peering traffic and adapt the backbone to the needs of the peering edge traffic.

Traffic matrix estimation through models and link data have been studied in [8, 26, 32, 38]. In our experience, direct measurement of traffic matrices is, unfortunately, necessary for online TE in operational networks. We extend prior work

with an evaluation of our IPFIX sampling based measurement system, and present data on real world traffic matrices.

Prior work [14, 18, 23] states that programming end-to-end paths in WANs takes minutes, and the size of tables to store traffic engineering rules is a constraint. Our work shows that network updates take seconds, even in very large networks. A key reason for this difference is that prior work used TCAM-based policy engines that are flexible but limited resources, and our work uses IP and MPLS lookup tables, which are optimized and large even in commodity switches, and programming in parallel using make-before-break semantics. Path selection is usually done offline [6] and load balancing over selected paths is online. In ONEWAN-TE, both are done online using the dynamic topology and traffic matrix. Traffic engineering is studied as the optimization of one objective like minimizing link utilization [36]. ONEWAN-TE optimizes each traffic class with different objectives, and uses class based forwarding to achieve the intent in the data plane. [1, 25] study the TE problem with faults. ONEWAN-TE formulation is tuned for a larger network. It protects faults at the granularity of optical risk groups and balances the opposing requirements of proactive fault protection without increasing latency in non-failure conditions.

9 Conclusion

Our journey with using SDN for traffic engineering has completed a full circle. SDN-TE technology matured in our network while managing inter-datacenter traffic, and has now replaced legacy TE in the Internet backbone. ONEWAN represents a 1000× increase in traffic volume and a 100× increase in network size compared to SWAN a decade ago. Some key elements of SWAN have withstood the test of time. For example, traffic engineering optimization retains the same structure and formulation. We still use router agents on WAN routers which run on multiple firmware operating systems, but have modified them to deal with greater scale and functionality. ONEWAN brings the benefits of smaller fault-domains from BLASTSHIELD to the Internet backbone, making it more reliable. We hope ONEWAN expands the scope of future work in wide-area TE, standardization of the controller-agent programming abstraction, and analysis of traffic matrix characteristics of cloud WANs. ONEWAN marks the progression of SDN into an Internet backbone. It is driven by scaling our backbone to serve today's users. We continue to expand the roles and functionality of ONEWAN as new opportunities emerge with the growth of network demands.

Acknowledgements. We thank our colleagues in Azure wide area networking team who made significant contributions to make ONEWAN successful, our shepherd Ram Durairajan, and NSDI reviewers for their comments that improved this paper.

References

[1] David Applegate, Lee Breslau, and Edith Cohen. Coping with network failures: Routing strategies for optimal demand oblivious restoration. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 270–281. 2004.

[2] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP tunnels, December 2001. RFC 3209.

[3] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 29–43. August 2019.

[4] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of routing control platform. In *Proceedings of USENIX NSDI*, pages 15–28, May 2005.

[5] COIN-OR linear programming solver. https://github.com/coin-or/Clp.

[6] Anwar Elwalid, Cheng Jin, Steven H. Low, and Indra Widjaja. MATE: MPLS adaptive traffic engineering. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1300–1309, 2001.

[7] Nick Feamster, Jay Borkenhagen, and Jennifer Rexford. Guidelines for interdomain traffic engineering. *ACM SIGCOMM Computer Communication Review*, 33(5):19–30, October 2003.

[8] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred D. True. Deriving traffic demands for operational IP networks: methodology and experience. *IEEE/ACM Transactions on Networking*, 9(3):265–279, 2001.

[9] Mikel Jimenez Fernandez and Henry Kwok. Building Express backbone: Facebook's new long-haul network, May 2017. https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/.

[10] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Brune Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture, July 2018. RFC 8402.

[11] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bernhard Korte, Lásló Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI Layout (Algorithms and Combinatorics)*, volume 9, pages 101–164. Springer-Verlag, 1990.

[12] OR-Tools – Google optimization tools. https://github.com/google/or-tools.

[13] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined internet exchange. In *Proceedings of ACM SIGCOMM*, pages 551–562, 2014.

[14] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of ACM SIGCOMM*, pages 15–26, August 2013.

[15] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *Proceedings of ACM SIGCOMM*, pages 74–87, August 2018.

[16] Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473). ISO/IEC 10589:2002, November 2002. https://www.iso.org/standard/30932.html.

[17] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of ACM SIGCOMM*, pages 3–14, August 2013.

[18] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of ACM SIGCOMM*, pages 539–550, August 2014.

[19] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 253–264. 2005.

[20] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *Proceedings of ACM SIGCOMM*, pages 515–526, August 2014.

[21] Dave Katz and Dave Ward. Bidirectional Forwarding Detection, June 2010. RFC 5880.

[22] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *Proceedings of USENIX NSDI*, pages 325–338, April 2022.

[23] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *Proceedings of USENIX NSDI*, pages 157–170. April 2018.

[24] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. Inter-datacenter bulk transfers with NetStitcher. In *Proceedings of ACM SIGCOMM*, pages 74–85. 2011.

[25] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of ACM SIGCOMM*, pages 527–538, August 2014.

[26] Alberto Medina, Nina Taft, Kavé Salamatian, Supratik Bhattacharyya, and Christophe Diot. Traffic matrix estimation: Existing techniques and new directions. In *Proceedings of ACM SIGCOMM*, pages 161–174. 2002.

[27] Debasis Mitra and K.G. Ramakrishnan. A case study of multiservice, multipriority traffic engineering design for data networks. In *IEEE GLOBECOM*, volume 1B, pages 1077–1083, 1999.

[28] OpenFlow switch specification, March 2015.

[29] Ping Pan, George Swallow, and Alia Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels, May 2005. RFC 4090.

[30] Peter Phaal and Marc Levine. sFlow version 5, July 2004.

[31] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol label switching architecture, January 2001. RFC 3031.

[32] Matthew Roughan, Mikkel Thorup, and Yin Zhang. Traffic engineering with estimated traffic matrices. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pages 248–258. 2003.

[33] Ganesh Sadasivan, Nevil Brownlee, and Benoit Claise. Architecture for IP flow information export, March 2009. RFC 5470.

[34] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with Edge Fabric: Steering oceans of content to the world. In *Proceedings of ACM SIGCOMM*, pages 418–431, 2017.

[35] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with Cascara. In *Proceedings of USENIX NSDI*, pages 201–216, April 2021.

[36] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. Cope: Traffic engineering in dynamic networks. In *Proceedings of ACM SIGCOMM*, pages 99–110. August 2006.

[37] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with Espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of ACM SIGCOMM*, pages 432–445, 2017.

[38] Yin Zhang, Matthew Roughan, Nick Duffield, and Albert Greenberg. Fast accurate computation of large-scale IP traffic matrices from link loads. In *Proceedings of ACM SIGMETRICS*, pages 206–217, 2003.

RHINE: Robust and High-performance Internet Naming with E2E Authenticity

Huayi Duan, Rubén Fischer, Jie Lou, Si Liu, David Basin, and Adrian Perrig
ETH Zürich

Abstract

The variety and severity of recent DNS-based attacks underscore the importance of a secure naming system. Although DNSSEC provides data authenticity in theory, practical deployments unfortunately are fragile, costly, and typically lacks end-to-end (E2E) guarantees. This motivates us to rethink authentication in DNS fundamentally and introduce RHINE, a secure-by-design Internet naming system.

RHINE offloads the authentication of *zone delegation* to an end-entity PKI and tames the operational complexity in an *offline* manner, allowing the efficient E2E authentication of *zone data* during *online* name resolution. With a novel logging mechanism, Delegation Transparency, RHINE achieves a highly robust trust model that can tolerate the compromise of all but one trusted entities and, for the first time, counters threats from superordinate zones. We formally verify RHINE's security properties using the Tamarin prover. We also demonstrate its practicality and performance advantages with a prototype implementation.

1 Introduction

The importance of DNS as an integral part of the Internet cannot be overstated. If DNS is corrupted, so would be all relying Internet services [33]. Yet, this critical system has no built-in protection for data at rest or in transit. The infamous Kaminsky attack [57] raised worldwide awareness of the severity of DNS cache poisoning and thereafter spurred the deployment of several protocol-level defense mechanisms. Recent years have, however, witnessed a flurry of new vulnerabilities [17, 66, 67, 90] that revive the threat of cache poisoning and DNS hijacking in general [50].

The implications of these attacks are profound: they enable the sabotage of a wide spectrum of online systems, ranging from web applications and email to time synchronization and cryptocurrencies [33]. One of most alarming facts is that DNS plays an essential role in bootstrapping the Internet's security. In the modern web PKI, certificate issuance relies on DNS-based channels for domain validation. If such channels are unauthenticated, attackers can manage to acquire fraudulent TLS certificates and impersonate domains [25, 27, 81]. Hence, an end-to-end (E2E) authenticated naming system is necessary for E2E secure communication.

DNS Security Today. Strengthening plain DNS with security guarantees has been a decades-long but still largely ongoing endeavor. DNSSEC [18] is by far the most important security extension to DNS. It allows a zone owner to cryptographically sign DNS records which, at least in theory, averts the threat of DNS hijacking. However, the deployment of DNSSEC is still far from complete (e.g., it is estimated that only 25% of DNS responses worldwide are validated as of mid-2022 [15]), and years' of practical experience indicates that it is highly fragile and fraught with problems.

The complexity of DNSSEC makes its operation an error-prone and expensive process. It requires each zone to synchronize its keying materials with its parent. Any inconsistency in an authentication chain will cause validation and hence resolution failure. This has caused frequent outages at all levels of the DNS hierarchy [54]. Validation failure can incur severe overhead to DNS servers and the name resolution process [53]. Partly because of these factors, and partly by design [88], end hosts rarely validate signed records by themselves but rely on validating recursive resolvers at best [64]. As a result, DNSSEC fails to provide E2E data authentication in practice, despite pervasive DNS interception [65, 71, 77].

The trust model of DNSSEC is also controversial. DNS is not designed for security, and mismanagement of DNSSEC by DNS operators is commonplace [29, 82]. Compromising a zone's secret key implies the control of all its subzones. This raises the concern that DNSSEC consolidates the power of the few Internet governance bodies and state governments over the DNS namespace [83]; in fact, large-scale DNS hijacking campaigns sponsored by state agencies have already been observed in the wild [46]. DNSSEC requires a validating entity to trust all zones on an authentication chain; any one of them can provide correctly signed yet bogus data [2].

These issues have their root in DNSSEC's underlying ar-

chitecture, which mirrors the hierarchical namespace, and therefore they cannot be resolved within DNS. This poses the question: *Is it possible to build a DNS-compatible yet robust naming system that enables efficient E2E authentication?*

Introducing RHINE. We provide an affirmative answer to this question with the design, verification, implementation, and evaluation of a system called RHINE. Our key insight is that the authentication of *zone data* and *zone delegation* in DNS, while treated identically by DNSSEC, should be decoupled. The latter form of authentication, which is more delicate and costly, can be performed by external trusted entities in an *offline* manner. Specifically, we employ certificate authorities (CAs) from the web PKI to certify zone delegation, allowing clients that already rely on these CAs to efficiently validate zone data during *online* name resolution.

Despite its promising opportunities, this architecture also raises unique challenges. Certifying a zone's authority with CAs creates a *circular dependency*, because, as mentioned earlier, secure certificate issuance hinges on a secure naming system in the first place. On a different front, the corruption of a single CA may put the entire DNS namespace at risk. Moreover, malicious DNS and PKI authorities can interact in subtle ways to subvert a zone's authenticity.

What we strive for is a system of *checks and balances* where the parties involved (zone owners, CAs, and loggers) watch over each other so that no single party or partial collusion between them can undermine a zone's authority. RHINE systematically addresses security threats arising from the envisioned architecture, offering a set of protocols for secure zone management and E2E-authenticated name resolution. At its core is Delegation Transparency (DT), a novel public logging mechanism to maintain global zone delegation status.

It is essential to rigorously establish the expected security properties for our design. Using a state-of-the-art security protocol verifier, Tamarin [68], we have formally proved that RHINE guarantees E2E data authenticity for legitimately delegated zones in a highly robust trust model.

Our evaluation with a prototype implementation shows that RHINE can cope with real-world certificate issuance rates (millions per day) and, compared with DNSSEC, achieve lower resolution latency and higher resolver performance.

2 Problem Statement

We start by introducing the basic concepts of DNS. Afterwards, we contextualize the data authentication problem and analyze the intrinsic weaknesses of DNSSEC.

2.1 Name Resolution Basics

The global DNS namespace is organized as a tree structure, where each node is a *zone* that manages *resource records* mapping names to IP addresses and other data. Delegating a portion of a zone creates another node in the tree and hence a (sub)zone. Below the root zone lie top-level domains (TLDs)

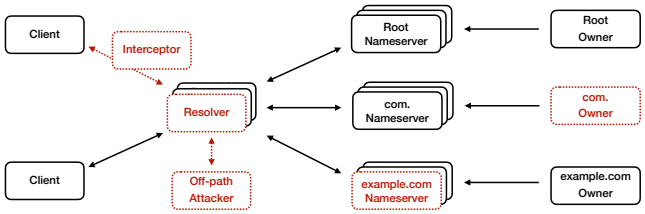


Figure 1: A simplified DNS infrastructure. Components in red and dotted lines indicate various threats to data authenticity.

such as `.com` and `.org`, second-level domains (SLDs) such as `a.com`, and so forth. A zone should be authoritative for all names under it except those under its *delegated* subzones. For example, assuming the zone `b.a.com` exists but `c.a.com` does not, then the zone `a.com` is authoritative for `c.a.com` and `d.c.a.com` but not `b.a.com` or `d.b.a.com`. A zone's apex is the name identifying the zone itself.

DNS runs on a distributed infrastructure. We consider a simplified infrastructure with four types of entities depicted in Figure 1. The *owner* of a zone is a logical entity with legitimate authority over it. When the context is clear, we extend the term "zone" to also indicate its owner. A zone hosts its data on multiple *(authoritative) nameservers*, which in many cases are not under the control of the zone owner [58, 76]. In the name resolution process, a *(recursive) resolver* handles name lookup queries from *clients* (aka stub resolvers), by iteratively asking nameservers for matching record(s) in a top-down manner. Caching at resolvers reduces the overall lookup costs and helps DNS operate at Internet scale.

2.2 Authentication in DNS

Plain DNS offers no authentication of resource records. They can be corrupted anywhere before reaching a client, as highlighted in Figure 1. Any on-path network node can access, modify, and fabricate DNS messages. An off-path adversary can also intervene in the resolution process and inject bogus data, as demonstrated by the Kaminsky attack and its variants [66, 67]. Nameservers and resolvers may deviate from their expected behavior due to domain hijacking [85], malware infection [32], business incentives [87], or regulatory pressure [72]. Less obvious threats are posed by malicious zone owners themselves, who can surreptitiously (and somewhat rightfully) manipulate their subzones [2, 83].

Network attackers can be thwarted by secure channels. DNSCurve [23] was proposed to protect the communication between a resolver and nameservers using an in-band key exchange scheme. More recent and widely deployed protocols include DNS over TLS (DoT) [52] and DNS over HTTPS (DoH) [47], which focus on securing the last-mile communication between a client and a resolver. These solutions fail to mitigate risks arising from nameservers, resolvers, and any intermediary servers on the resolution path.

As the most prominent security addition, DNSSEC enhances DNS with data integrity and origin authentication.

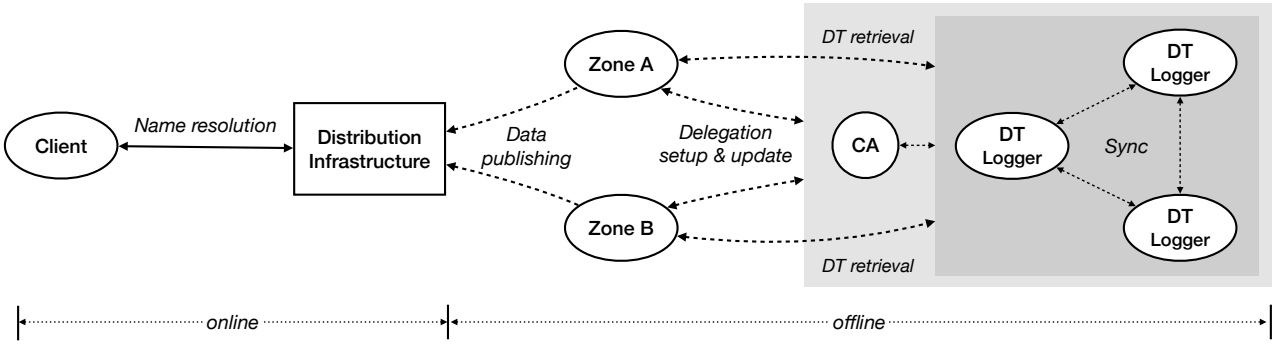


Figure 2: The high-level architecture of RHINE, where operational complexity (e.g., the authentication and management of zone authority) is pushed towards an *offline* phase. The arrows represent interactions between (groups of) entities.

It allows a zone to cryptographically sign its records using a secret key, with the corresponding public key signed by the parent zone. A security-aware resolver can verify a signed record by following an authentication chain all the way up to the root zone (key), without trusting any on-path servers.

2.3 Problems with DNSSEC

Since the signing of the root zone's key in 2010, DNSSEC has seen gradual uptake, but the deployments are not all smooth. It is often cited by practitioners as overly complicated and not worth the costs it exacts [54]. We analyze its drawbacks in practical operation and from a security perspective.

Fragile Operation. DNSSEC requires synchronization between each pair of adjacent nodes in an authentication chain. Any inconsistency (e.g., missing or mismatching keys or security parameters) between a zone and its parent will cause validation and hence resolution failure, blocking not only the failed zone but also all its subzones. It is thus unsurprising that Internet outages caused by DNSSEC happen frequently *at all levels* including the root, TLDs and SLDs, and *across various organizations* including DNS governance bodies themselves (e.g., ICANN and RIPE) as well as large service providers (e.g., Verisign, Dyn, and Google) [54].

While DNSSEC already imposes significant performance overhead with respect to plain DNS resolution, validation failure can further boost its costs. It is estimated that with failure factored in, the authoritative nameservers of a DNSSEC-signed zone should be prepared to handle 10 times the query traffic volume and 100 times the response traffic volume of their unsigned counterparts for an Internet-wide deployment [53]. The potential of abusing DNSSEC for denial of service (DoS) is well-recognized and many real-world attacks have been reported [1].

The operational complexity, high failure rate, and performance overhead all contribute to the fact that end hosts rarely validate DNSSEC-signed records [64]. It is actually by design that end hosts should rely on validating recursive resolvers to verify records [88]. As a result, DNSSEC almost never provides E2E data authentication in practice.

Fragile Security. The security of DNSSEC rests on DNS itself. However, unlike PKIs, DNS is not designed for security; and unlike CAs, zone owners and operators may not be security-savvy. Real-world measurements have revealed widespread mismanagement of DNSSEC with flawed security practices (using weak keys, reusing keys for multiple zones, etc.) [29, 82]. The compromise of a zone's secret key endangers not only the zone itself but also all its subzones.

A common criticism of DNSSEC is that it consolidates the Internet's governance [83]. The root zone is governed by ICANN, the most important TLD `.com` is managed by Verisign under the jurisdiction of US law, and each country-code TLD is ultimately controlled by the corresponding sovereign state. Large-scale DNS hijacking campaigns sponsored by state agencies have been observed in real world [46].

While the governance model of DNS remains a subject of controversy, from a technical point of view, DNSSEC's trust model is fundamentally fragile in that it provides clients with no option but to trust all zones on a delegation chain. A malicious zone can surreptitiously claim and serve authenticated data for names belonging to any subzone. This problem has just begun to gain attention from the Internet community, and there is a proposal to mark the root zone and TLDs as delegation-only so that their ability to serve authoritative data is limited [2]. However, implemented within DNS, this mechanism cannot solve the inherent limitations of DNSSEC.

2.4 Desired Properties

Our analysis of DNSSEC reveals the following properties desired by an ideal authenticated Internet naming system.

- **P1: End-to-end (E2E) data authenticity.** A validating client must be assured that any verified resource record is indeed generated by the genuine authoritative zone.
- **P2: Authentication efficiency.** The computation and communication costs of authenticating resource records, especially in case of failure, are lower than DNSSEC.
- **P3: Operational robustness.** The authentication of a zone's data is unaffected by any superordinate zone's op-

erational faults in managing security, e.g., misconfigured security policies or keying materials.

- **P4: Robust trust model.** If a zone relies on a group of entities (including its parent and any external trusted parties) to establish its authority, then no single entity or partial collusion between them can claim authority over the zone.

3 RHINE Overview

RHINE is a naming system with built-in security, satisfying all the properties listed in Section 2.4. Our starting point is the observation that the authentication of a DNS zone consists of two parts: authenticating resource records during name resolution and, when the zone is created, authenticating the delegation's legitimacy. The latter can be offloaded from clients to external trusted entities: in particular, the CAs in today's web PKI that billions of clients already rely on. Once a zone is delegated and certified, it can serve authenticated data and manage its security independently, without synchronizing with its parent as in the case of DNSSEC. This isolates the failures caused by a zone's security mismanagement from its subzones. The reduction in validation failure and authentication chain length also improves name resolution performance.

This new security architecture simultaneously achieves the desired properties **P1**, **P2**, and **P3**. Yet, it introduces both unprecedented opportunities and challenges to meet **P4**, without which the system can be broken in many ways. This is the main focus of our design (Section 3.3).

RHINE Architecture. We depict our architecture in Figure 2. It consists of two parts. In the *offline* part, zone owners establish new delegations by acquiring publicly logged RHINE certificates (RCert) from a CA and loggers (Section 5.1). An existing zone can update its RCert or delegation status (Section 5.2). It also periodically retrieves delegation status proofs (DSP) (Section 5.3) from a public transparency log (Section 4). A zone signs its resource records using its RCert and publishes them to a distribution infrastructure. During *online* name resolution, a client who already relies on the web PKI can easily verify an answer's authenticity using the associated RCert and DSP (Section 5.4).

This architecture shifts much of DNSSEC's complexity to offline operations, minimizing the risk of failure during the name resolution process. It also clearly separates the distribution and authentication of DNS data. While we intend to reuse the existing DNS infrastructure consisting of authoritative nameservers, recursive resolvers, forwarders, etc., RHINE can be instantiated with other distribution architectures such as a peer-to-peer network [14], or enable client authentication of records received through DoT or DoH.

3.1 Notation and Primitives

We use *uppercase* letters (e.g., X) to identify entities (zone owners, CAs, and loggers) that run RHINE protocols, and *lowercase* letters in the subscript to identify zones (e.g., $\texttt{ZN}_x$) and their associated data (e.g., $\texttt{RCrt}_x$). For brevity, we sometimes refer to the pair of zones related by delegation and their corresponding owners simply as the *parent* and *child*.

Table 1: Summary of Notation.

Notation	Definition
pk_X, sk_X	The key pair of entity X (in *uppercase*)
$\texttt{ZN}_x$	A zone identified by x (in *lowercase*)
$\texttt{RCrt}_x$, zpk_x, zsk_x	The RCert and associated key pair of $\texttt{ZN}_x$
$:=$	Definition/assignment operator
$(a,b,\ldots)$	A tuple of values encoded as a string
$\texttt{H}(\cdot)$	A secure hash function
$\langle m\rangle_X$ or $\langle m\rangle_x$	A message signed with sk_X or zsk_x
$\Sigma.\texttt{Vf}(k,m)$	Verify a signed message m with a key/cert k
$\texttt{Acc.Vf}(ac,p)$	Verify a membership proof p with digest $\texttt{ac}$

We use standard cryptographic primitives including secure hash functions and digital signatures. The public keys of CAs and loggers are known to all entities. To design succinct data structures, we also use cryptographic accumulators [34] that can commit sets of values into small digests and generate compact membership proofs. Classic constructions include the Merkle hash tree (MHT) [69] and its variants. Table 1 summarizes our notation.

3.2 Threat Model

Table 2 summarizes the adversaries we consider in the design of RHINE and the expected security properties.

$\mathcal{A}_1$ is a conventional Dolev-Yao network attacker [37] (who can eavesdrop, modify, and inject messages transmitted over the network) augmented with the ability to control the entire DNS distribution infrastructure.

The next two types of adversaries pertain to today's web PKI ecosystem: $\mathcal{A}_2$ can issue arbitrary certificates by compromising a CA; $\mathcal{A}_3$ can compromise some loggers and provide fake or inconsistent log data to users. Since we repurpose the web PKI to authenticate delegated zone authority, RHINE must also deal with these adversaries.

For the first time, we systematically address an adversary ($\mathcal{A}_4$) that controls a zone and attempts to subvert its subzones by declaring authoritative data for them. This capability is inherent in the hierarchical naming structure of DNS, i.e., a name under a zone is also under its parent zone. For an authenticated naming system where zone data is cryptographically signed, $\mathcal{A}_4$ has access to the private key of the zone it controls. As an example, an $\mathcal{A}_4$ attacker compromising the TLD `xyz` can generate valid records for `abc.example.xyz`, despite that the SLD `example.xyz` has been legitimately delegated.

Overall, we consider attackers that seek to, given the stated capabilities, break the naming system's *data authenticity but not availability*—that is, tricking clients into accepting malicious data rather than preventing clients from receiving any answer. We assume that the attackers cannot break the cryptography primitives used by RHINE, and that privacy aspects of DNS are outside this paper's scope.

Table 2: Summary of adversaries considered in DNS and the web PKI. We also list the corresponding security properties and representative defense mechanisms to achieve them.

Adversary	Capability	Security property (informal)	Defense mechanisms
Dolev-Yao	controls communication networks	channel security	DoT/DoH/DoQ, DNSCurve
$\mathcal{A}_1$	+ DNS distribution infrastructure	data authenticity	DNSSEC, GNS [14]
$\mathcal{A}_2$	controls some CA(s)	certificate misissuance prevention	ARPKI [21], F-PKI [28]
$\mathcal{A}_3$	controls some logger(s)	tolerating all but one compromises	LogPicker [36], CTng [63]
$\mathcal{A}_4$	controls a DNS zone (e.g., a TLD)	authority independence (of subzones)	-
$\mathcal{A}_1$ + $\mathcal{A}_2$ + $\mathcal{A}_3$ + $\mathcal{A}_4$ **(strongest possible adversary)**		**E2E authenticity with robust trust**	**RHINE**

3.3 Design Rationale

RHINE strives to counter the strongest possible adversary that combines all capabilities as shown in Table 2. Before fleshing out RHINE's design, we discuss the main aspects to be considered, analyze why existing approaches fail, and highlight the intuitions behind our solutions.

3.3.1 Validating Zone Ownership ($\mathcal{A}_1$)

Secure delegation in RHINE requires the expected zone owner to request an RCert from a CA. The issuing CA must verify that a requesting entity indeed controls the zone to be certified. Commonly known as domain validation (DV), this process is mandatory for the issuance of TLS certificates. In standard practice [20], the requester proves its ownership of a domain by publishing a challenge token specified by the contacted CA. A network attacker can exploit an insecure channel in this process to obtain a fraudulent certificate. Unfortunately, all practical DV channels hinge on DNS and are therefore exploitable by an $\mathcal{A}_1$ attacker [24, 27, 81]. Applying these standard DV methods to our case will lead to a ***circular dependency***: *the CA depends on an authenticated zone for ownership validation and RCert issuance, but meanwhile, the zone needs an RCert to authenticate its data in the first place.*

RHINE solves this dilemma by engaging the parent to approve the delegation. This is indeed necessary, as the parent still legitimately controls the child before it is established. Specifically, the parent must sign a delegation request using its own RCert. The CA can then verify that the *current* owner of the child zone approves the delegation. In doing so, RHINE creates an *implicit offline* authentication chain of delegated authority, as opposed to what is explicitly constructed by DNSSEC, and shifts the heavy authentication workload away from the client side of DNS.

3.3.2 Preventing Certificate Misissuance ($\mathcal{A}_2$ & $\mathcal{A}_3$)

Security breaches of CAs [49] spurred the deployment of Certificate Transparency (CT) [61], which employs public logs to make misissued certificates detectable. Mainstream browsers have mandated public logging for TLS certificates to be valid [3, 6]. One limitation of CT is that it provides *deterrence* rather than *prevention*. Fraudulent certificates may still be used before being detected and revoked. CT loggers passively accept certificates that meet basic validity criteria (properly formatted and signed, non-expired, etc.) but never validate domain ownership as CAs do. Also, the compromise of loggers has already occurred in practice [79].

RCerts are more critical than TLS certificates in terms of security, because the naming service is one of the weakest links in many Internet systems including the web PKI itself [33]. In addition, the detection of fraudulent RCerts is more involved in that it requires investigating delegation chains rather than individual domains. Therefore, we need preventive measures to foil the misissuance and logging of unauthorized RCerts.

There are proposals to make today's web PKI more resilient to the compromise of CAs and loggers [21, 28, 36, 63]. Yet, they are not applicable to the new security architecture we envision for RHINE. This is because: (1) they are designed for TLS certificates and so they will suffer from the bootstrapping dilemma discussed earlier (Section 3.3.1), and (2) their log data models, either reusing or building upon CT, do not meet our security and performance requirements (Section 3.3.3).

We address the $\mathcal{A}_2$ and $\mathcal{A}_3$ adversaries from several aspects: (1) integrating loggers into the certificate issuance process for proactive verification of the data to be logged, (2) enabling a zone to choose its own trusted loggers rather than relying on whichever loggers are chosen by CAs (as in the case of CT), and (3) enforcing loggers and CAs to crosscheck each other throughout the certificate issuance and logging process. This allows RHINE to defeat attackers that can compromise multiple trusted entities designated by a zone.

3.3.3 Countering Parental Attacks ($\mathcal{A}_4$)

A zone gains *authority independence* if its ancestors cannot claim authoritative data under its authority. The cryptography of DNSSEC makes the situation even worse than in regular DNS. Our new security architecture does not immediately address this challenge. In particular, a malicious parent can still serve authentic records for delegated children, using its own RCert or alternative child RCerts acquired by it. In order to counter such parental attacks, we must enable a dependent entity (client or CA) to verify the status of the delegations in question without trusting zone owners themselves.

Since delegation status can be inferred from logged RCerts, it seems plausible to design a solution atop CT. A closer

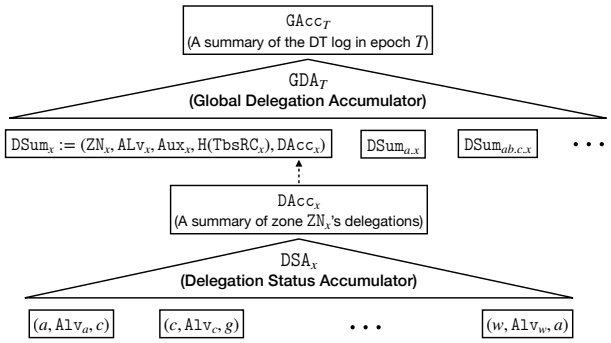


Figure 3: The data structures used by DT.

look reveals several pitfalls. First, a dependent entity needs to ascertain that a zone in question does *not* exist, but CT has no native support for *absence proofs*. Second, such proofs must have global coverage, but CT loggers operate independently and maintain only partial views of all issued certificates. It would be onerous to assemble and synchronize data from all CT logs with correctness and performance guarantees. Third, the data structures used by CT are too heavy to represent and authenticate global delegation status.

These inefficiencies motivate us to create a more efficient transparency mechanism dedicated to keeping track of the entire namespace's delegation structure.

4 Delegation Transparency

At the heart of RHINE is Delegation Transparency (DT), a lightweight verifiable log design. In contrast to CT, which maintains the *history* of all certificates ever issued, DT offers an up-to-date *snapshot* of global zone delegation status. A single DT log is replicated to a consortium of loggers. The loggers receive requests to update delegation status and periodically synchronize with each other to maintain a consistent log. They also provide publicly verifiable delegation information. Below we introduce the basics of DT. Its operation as an integral part of RHINE is described in Section 5.

Log Data. Figure 3 depicts DT's data model. We define the delegation status of a zone $\texttt{ZN}_x$ as a tuple $(\texttt{ALv}_x, \texttt{Aux}_x, \texttt{CSet}_x)$, where the first item is the *authority level* of $\texttt{ZN}_x$ (explained below), the second is auxiliary information for $\texttt{ZN}_x$ (e.g., expiration time or revocation status of the delegation), and the third is a set representing $\texttt{ZN}_x$'s child zones and their authority levels: $\texttt{CSet}_x := \{(c_1, \texttt{ALv}_{c_1}), (c_2, \texttt{ALv}_{c_2}) \ldots, \}$.

We encode the delegation status of $\texttt{ZN}_x$ into a data structure called $\texttt{DSum}_x$ (delegation summary). $\texttt{DSum}_x$ contains a cryptographic digest of the zone's RCert. This ensures that at any time there is only one valid RCert per zone, capturing that the authority over a zone should be unique. Since a zone may have many delegations, $\texttt{DSum}_x$ stores the digest ($\texttt{DAcc}_x$) of an accumulator $\texttt{DSA}_x$ over $\texttt{CSet}_x$ rather than $\texttt{CSet}_x$ itself. This reduces the cost of authenticating a specific child's (non)existence. Each input element of $\texttt{DSA}_x$ contains the label

	TER	¬TER ∧ ¬DOL	DOL	
IND	(1, 0)*	(1, 1)*	(0, 1)*	EOI
		(1, 2)	(0, 2)*	¬EOI
¬IND	(1, 0)*	(1, 1)*	(0, 1)*	

Figure 4: The authority level matrix derived from the interaction of constraint flags. The shaded area indicates the division caused by the EOI flag. In each pair (a, b), $a \in \{0, 1\}$ encodes a zone's ability to serve authoritative data for all its names excluding those of its independent subzones; $b \in \{0, 1, 2\}$ encodes a zone's delegation capability (0: not allowed; 1: non-independent child only; 2: any child). The cases marked with * permit fast data validation (see Section 5.4).

and authority level of one child as well as the label of the next child in a *canonical order* [19]. This allows a single membership proof from the accumulator to prove either the presence or absence of a child zone.

For efficient synchronization and auditing of the DT log, we introduce a global accumulator GDA over all $\texttt{DSum}_x$s. Loggers can commit GDA's digest (GAcc), along with the necessary data to replay logged changes, into an authenticated data structure that supports succinct consistency proofs [62].

Authority Level. While we envision that authority independence is desired by many zones (including all TLDs and SLDs), this may not always be the case, for instance when the parent and child are managed by the same entity. To enable fine-grained control over zone authority, we introduce the concept of *authority level*, which places constraints on what a zone can do to its data and delegation. We define authority levels using *constraint flags*, as depicted in Figure 4.

The flag IND indicates a zone's authority independence. By definition, an independent zone has the sole authority over its names, whereas a non-independent (¬IND) zone's names are also under the authority of its parent. The data served by a zone comes in two types: authoritative and delegation. A *terminating* (TER) zone can serve only authoritative data; all leaf zones are by default terminating. A *delegation-only* (DOL) zone can serve only delegation data (i.e., NS records in DNS); all TLDs are supposed to be delegation-only. Note that these two flags cannot be set simultaneously as this would lead to an empty and useless zone. Since a non-independent zone can never delegate to an independent child, authority independence can end at some non-leaf zone on a delegation path; such a zone is marked as *end-of-independence* (EOI).

Delegation Status Proof (DSP). Clients make use of the DT log in the form of DSP, which consists of a timestamped $\texttt{DSum}_x$ signed by loggers and, if necessary, a membership proof from $\texttt{DSA}_x$ for some child zone $\texttt{ZN}_y$ of $\texttt{ZN}_x$. A DSP enables clients to determine a zone's realm of authority and hence whether to accept an answer signed with the corresponding RCert. A malicious parent may use an outdated

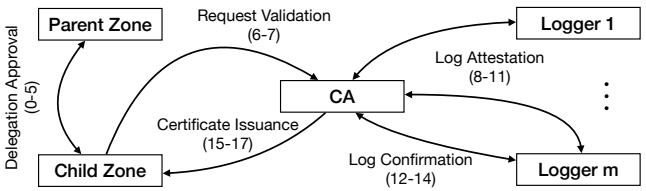


Figure 5: Overview of the delegation setup protocol.

DSP to trick clients into accepting fraudulent data for the names belonging to a delegated child. Prudent clients should accept only DSPs that are recent enough.

5 RHINE Protocols

We specify RHINE's core functions with a set of protocols, including the secure management of zone delegation, the maintenance and usage of DT, and E2E-authenticated name resolution. The entire system operates in *epochs*, which are consecutive time windows of a predetermined length. This is necessary to keep DT loggers in synchrony and to establish the system's security. In each epoch, zone owners can securely set up new delegations or update existing ones until a cut-off time. The resulting changes in these zones' delegation status will be applied to the DT log within the same epoch and take effect from the next epoch. Zone owners can actively monitor the log for unexpected events like attacks or operational faults, and take action accordingly. They also regularly retrieve signed log entries to prove their authority over answers served during name resolution.

5.1 Secure Delegation Setup

In RHINE, delegating a zone $\texttt{ZN}_c$ begins with the intended owner C negotiating the delegation with P, the owner of the parent zone $\texttt{ZN}_p$. This follows standard DNS practices, e.g., domain registration. Afterwards, C should run the secure delegation setup protocol specified in Figure 6 to obtain an RCert. This protocol follows the design intuitions presented in Section 3.3. An overview of its flow is depicted in Figure 5.

In the initial phase (Steps 0-5), C asks for a signed approval (apv) for its delegation request (sdr) from P. The request encodes the trusted entities selected by C, the delegation parameters negotiated with P, and most importantly, the public key to be certified. The corresponding private key is also used to sign the request. There must be a way for P to authenticate the association between C and the key. This is done using an initial secure out-of-band key registration procedure (Step 0), for example, via a secure web portal with account-based client authentication when P is a domain name registrar.

Next, C sends the request to a CA for validation (Steps 6-7). In addition to verifying the parent's approval, the CA checks the delegation's legitimacy using the DT log. If everything is correct, the CA sends a pre-logging request (prl), which includes a to-be-signed certificate, to the designated loggers for crosschecking (Steps 8-11). After assembling the loggers' attestations, the CA randomly picks one of them to store the logging request ($lreq$) as an input to the later aggregation process (Steps 12-14). Finally, C receives an RCert accompanied by attestations and a confirmation that the zone $\texttt{ZN}_c$'s delegation status will be added to the DT log (Steps 15-17).

Our design ensures that the entire delegation setup process is witnessed by multiple parties and any misbehaving party will be held accountable for the messages it signs. Any verification failure will cause the protocol execution to abort, broadcasting a failure message to all the involved parties. It is impossible to obtain a valid logged RCert without faithfully following the protocol. Even in the presence of an omnipotent attacker whose capabilities go beyond our threat model, RHINE still allows a zone owner to detect and counter attack attempts before harm is caused (see Section 6).

Secure Bootstrapping. The delegation setup protocol assumes the parent's RCert already exists. A bootstrapping problem arises at the top of the namespace. Representing a critical Internet authority in itself, the root zone should not depend on another CA. Therefore, we treat the root zone as a root CA that signs its own RCert. Similarly, TLDs resemble intermediate CAs with their RCerts signed by the root RCert. This allows the root zone and TLDs to retain their innate power over the namespace, effectively restricting an external CA's influence over the namespace to SLDs and below.

5.2 Secure Delegation Update

Once delegated, a zone can manage itself mostly independently of its parent. This includes updating its RCert and other delegation parameters. Similarly to delegation setup, processing an update request involves some CA and loggers as witnesses. The parent's involvement is required only for a request to extend the delegation's validity period or to change the child's authority level from non-independent to independent. RHINE has built-in support for certificate revocation. An updated RCert automatically revokes the old one, because by design a zone can only have one valid RCert at any time; a zone can also request for explicit revocation.

The update protocol is similar to the delegation setup protocol for the message flow and verification procedures. The major difference is that a zone should now sign the update request using its own RCert (instead of the parent's) to prove its authority. We provide further details in Appendix A.2.

5.3 DT Aggregation and Retrieval

In each epoch, loggers will receive disjoint sets of requests to update the DT log. To ensure the log's global consistency, they must aggregate all requests by synchronizing with each other. Wanner et al. formalized this problem as *secure log replication*—a special case of state machine replication—and proposed Logres, a formally verified log replication protocol with Byzantine fault tolerance that is optimal in terms of round complexity and the number of tolerable faults [86].

0. C generates a key pair (zpk_c, zsk_c) and register zpk_c to P via a secure out-of-band channel.
1. C : select A (a CA) and $\mathcal{L}_c$ (a set of loggers)
   // t_0 is a timestamp within the current epoch $\mathtt{T}$, al is the requested authority level, aux is auxiliary information.
   : $rid \coloneqq \mathtt{H}(\mathtt{ZN}_c, zpk_c, A, \mathcal{L}_c, t_0, al, aux)$
   // rid is implicitly included in all subsequent messages
2. $C \to P$: $sdr \coloneqq \langle rid, \mathtt{SDReq}(\mathtt{ZN}_c, zpk_c, A, \mathcal{L}_c, al, aux)\rangle_c$
3. P : Verify $\Sigma.\mathtt{Vf}(zpk_c, sdr)$ and whether al, aux
   : match what are agreed upon with C.
4. $P \to C$: $\mathtt{RCrt}_p, apv \coloneqq \langle \mathtt{SDApprvl}(\mathtt{H}(sdr))\rangle_p$
5. C : Verify $\Sigma.\mathtt{Vf}(\mathtt{RCrt}_p, apv) \wedge \mathtt{Match}(apv, sdr)$
6. $C \to A$: sdr, apv, $\mathtt{RCrt}_p$
7. A : Verify $\Sigma.\mathtt{Vf}(\mathtt{RCrt}_p, apv) \wedge \Sigma.\mathtt{Vf}(zpk_c, sdr)$
   : $\wedge\ \mathtt{Match}(apv, sdr)$
   : Retrieve $dsp \coloneqq (\langle \mathtt{DSum}_p, \mathtt{T}'\rangle_{\mathcal{L}_c}, mem)$
   : from local cache or the loggers $\mathcal{L}_c$.
   // Check if DSP is valid and the delegation is legit
   : Verify $\Sigma.\mathtt{Vf}(pk_{\mathcal{L}_c}, \langle \mathtt{DSum}_p, \mathtt{T}'\rangle_{\mathcal{L}_c})$
   : $\wedge\ \mathtt{Match}(\mathtt{RCrt}_p, \mathtt{DSum}_p) \wedge \mathtt{T}' = \mathtt{T} \wedge$
   : $\mathtt{Acc}.\mathtt{Vf}(\mathtt{DAcc}_p, mem) \wedge \mathtt{Legal}(\mathtt{ALv}_p, al)$
   : $tbsrc \coloneqq \mathtt{TBSCert}(\mathtt{ZN}_c, zpk_c, A)$
   // Pre-logging requests to all designated loggers
8. $A \to \mathcal{L}_c$: $prl \coloneqq \langle \mathtt{PreLog}(sdr, apv, tbsrc)\rangle_A, \mathtt{RCrt}_p$
9. L_i : Verify $\Sigma.\mathtt{Vf}(pk_A, prl) \wedge L_i \in \mathcal{L}_c$
   : $\wedge\ \Sigma.\mathtt{Vf}(\mathtt{RCrt}_p, apv) \wedge \Sigma.\mathtt{Vf}(zpk_c, sdr)$
   // Check if the to-be-signed cert matches the requested
   : $\wedge\ \mathtt{Match}(apv, sdr) \wedge \mathtt{Match}(sdr, tbsrc)$
   // Check the delegation's legitimacy using local DT log
   : $\wedge\ \mathtt{ZN}_c$ not delegated $\wedge\ \mathtt{Legal}(\mathtt{ALv}_p, al)$
   : $nds \coloneqq (\mathtt{T}, A, \mathcal{L}_c, \mathtt{ZN}_c, al, aux, \mathtt{H}(tbsrc))$
10. $L_i \to A$: $att_i \coloneqq \langle \mathtt{LogAttest}(L_i, \mathtt{H}(nds))\rangle_{L_i}$
11. A : Verify $\Sigma.\mathtt{Vf}(pk_{\mathcal{L}_c}, \{att_i\}) \wedge \mathtt{Match}(prl, \{att_i\})$
    // L is randomly selected from $\mathcal{L}_c$ by A
12. $A \to L$: $lreq \coloneqq \langle \mathtt{LogReq}(L, nds, \{att_i\}_{L_i \in \mathcal{L}_c})\rangle_A$
13. L : Verify $\Sigma.\mathtt{Vf}(pk_A, lreq) \wedge \mathtt{Match}(nds, \{att_i\})$
    : $\wedge\ \Sigma.\mathtt{Vf}(pk_{\mathcal{L}_c}, \{att_i\}) \wedge L \in \mathcal{L}_c$
    : Add $lreq$ to a pending pool for aggregation
14. $L \to A$: $lc \coloneqq \langle \mathtt{LogCfm}(L, \mathtt{H}(nds))\rangle_L$
15. A : Verify $\Sigma.\mathtt{Vf}(pk_L, lc) \wedge \mathtt{Match}(lc, lreq)$
    : $\mathtt{RCrt}_c \coloneqq \langle \mathtt{FinalRCert}(\mathtt{TbsRC}_c, \mathcal{L}_c)\rangle_A$
16. $A \to C$: $\mathtt{RCrt}_c$, $\{att_i\}_{L_i \in \mathcal{L}_c}$, lc
17. C : Verify $\Sigma.\mathtt{Vf}(pk_A, \mathtt{RCrt}_c) \wedge \mathtt{Match}(sdr, \mathtt{RCrt}_c)$
    : $\wedge\ \Sigma.\mathtt{Vf}(pk_{\mathcal{L}_c}, \{att_i\}) \wedge \mathtt{Match}(sdr, \{att_i\})$
    : $\wedge\ L \in \mathcal{L}_c \wedge \Sigma.\mathtt{Vf}(pk_L, lc) \wedge \mathtt{Match}(sdr, lc)$

Figure 6: The secure delegation setup protocol. A party stores the messages it sends and receives whenever necessary. The function `Match()` checks the consistency between two data objects and `Legal()` checks a delegation's legitimacy. Other functions, such as `SDReq()` and `TBSCert()`, construct proper data objects from the input parameters.

This protocol however cannot be directly applied to our case, because it is agnostic to the validity of inputs: a malicious logger that participates in the consensus process faithfully can still inject arbitrary bogus data into the log.

To this end, we enhanced Logres with input validation (among other technicalities), requiring each logging request to be attested by the specified trusted entities (Steps 10-13, Figure 6). Using the modified version as a core consensus routine, we designed a secure aggregation protocol (Appendix A.3) that allows a majority of honest DT loggers to efficiently maintain a consistent log even in case of Byzantine faults.

Within an epoch, DT loggers can run the aggregation protocol multiple times according to some system-wide policies, e.g., at regular intervals or whenever their pools of pending request become filled up. Pipelining log aggregation with delegation setup and update improves overall system efficiency. Loggers should stop accepting new requests when the number of pending requests is estimated to exceed what they can aggregate after the cut-off time. This ensures that all requests confirmed in an epoch (Step 15, Figure 6) can be applied to the DT log by the end of the epoch.

After a successful execution of the delegation setup or update protocol in epoch $\mathtt{T}$, the owner of a zone $\mathtt{ZN}_x$ should actively monitor the DT log. Once the change to its delegation status has been admitted, it can retrieve from its designated loggers $\mathcal{L}_x$ a signed log entry $\langle \mathtt{DSum}_x, \mathtt{T}+1\rangle_{\mathcal{L}_x}$ (and $\mathtt{DSA}_x$ as well if it is updated), which will be used to generate DSPs in epoch $\mathtt{T}+1$. Each zone should retrieve its (re-)signed log entry once per epoch, even if its delegation status is not changed. Note that using the epoch counter, instead of higher-precision time units, to timestamp log requests and DSPs effectively guarantees RHINE's synchrony while reducing the system's reliance on secure global time synchronization (e.g., [40]).

Parameter Selection. Epoch length is an important system-wide parameter and its selection comes with trade-offs. A large value means long waiting time for zones' delegation status changes to take effect. A small value leads to frequent retrieval of the DT log and thereby performance issues. On balance, we suggest a practical epoch length of 48 hours and a cut-off time 24 hours before the end of an epoch. This is based on our evaluation results as well as CT's Maximum Merge Delay of 24 hours [62]—the longest time period within which CT loggers must add promised certificates to their logs. We consider doubling the waiting time in DT acceptable because the administration of zone delegation happens less frequently than the management of TLS certificates for domains in already established zones. With the suggested parameters, it takes 24–48 hours to set up an operational zone.

5.4 Authenticated Name Resolution

With only minor changes, RHINE can augment the plain name resolution of unprotected DNS (or any other distribution infrastructure) with E2E data authentication. A zone owner

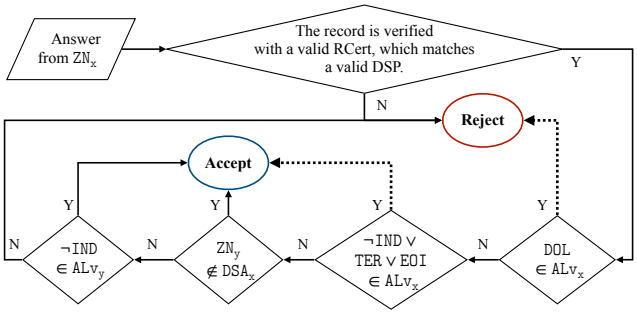


Figure 7: The flow of validating an answer received from zone $\mathtt{ZN}_x$, which has a potential child $\mathtt{ZN}_y$ that encloses the queried name. Dashed arrows indicate shortcuts for verification.

needs to sign its resource records using the zone's RCert before publishing them to nameservers. Whenever an authoritative answer is to be provided, a nameserver will also return the corresponding RCert and DSP to the querying client.

Figure 7 depicts the data validation flow. It starts with the verification of the signed records using RCert, similarly to DNSSEC. The client then additionally verifies whether the RCert matches the cryptographic digest contained in the DSP. Afterwards, the client decides whether the queried name falls within $\mathtt{ZN}_x$'s realm of authority by checking its authority level $\mathtt{ALv}_x$. In most cases (Figure 4), a shortcut can be taken to make a quick decision: an answer for a non-apex name from a delegation-only zone is always rejected by definition; an answer from a non-independent, end-of-independence, or terminating zone is always accepted because there exists no further independent subzone. If none of these applies, the client will examine $\mathtt{ZN}_x$'s potential child zone $\mathtt{ZN}_y$ that encloses the queried name, which involves verifying a membership proof from $\mathtt{DSA}_x$, and accepts the answer only if $\mathtt{ZN}_y$ does not exist or is non-independent.

6 Formal Security Analysis

The overall security goal of RHINE is to preserve a zone's data authenticity against powerful adversaries. This can be broken down into two concrete objectives: (1) preventing attackers from obtaining a valid RCert to take over a zone that is *not yet delegated*, and (2) preventing the forgery of authoritative data from an *already delegated* zone. In the first case, a victim zone is still under the legitimate control of its parent, and therefore the parent must be assumed trusted for a meaningful notion of security. In the second case, the additional protection of a zone from its malicious parent leads to the notion of authority independence.

We define these two objectives with the following theorems (presented informally). Table 3 summarizes the main security parameters in RHINE. The constraint $f < n/2$ is required by Logres [86]. We additionally require that $m \leq f+1$ holds for any zone, as otherwise an attacker with the $\mathcal{A}_3$ capability can inject arbitrary data into the DT log.

Table 3: Main Security Parameters in RHINE

	Definition	Constraint
n	Number of loggers in a global DT setup	-
f	Number of tolerable faulty loggers	$f < n/2$
m	Number of loggers chosen by a zone	$m \leq f+1$

Theorem 1 *If a zone $\mathtt{ZN}_x$ is delegated with $\mathtt{RCrt}_x$ issued and logged in epoch $\mathtt{T}$, then a corresponding secure delegation request must have been approved by the parent earlier in epoch $\mathtt{T}$, even if an $\mathcal{A}_1+\mathcal{A}_2+\mathcal{A}_3$ attacker formed by the entities specified in $\mathtt{RCrt}_x$ is present throughout epoch $\mathtt{T}$.*

The security guarantee provided by Theorem 1 resembles that of the ACME protocol, which also concerns unauthorized certificate issuance [24], but RHINE deals with much stronger attackers. In fact, RHINE allows even better security than is promised by this theorem. We discuss the following situations where the threat assumptions are violated.

It may happen that an adversarial parent, despite having approved a legitimate delegation for a child, front-runs the delegation setup protocol for the child zone with different parameters (in particular the key to be certified) in the current epoch. Yet, the expected owner of the child zone can detect from the DT log the misissued RCert, which will remain unusable until the associated DSP becomes valid in the next epoch (see Section 5.3). The owner being impersonated can then request to revoke the illegitimate delegation before it takes effect, by presenting the parent's approval as evidence to the relevant CA and loggers.

An even worse, though unlikely, case is that an attacker manages to control a CA and m loggers. This enables it to issue an RCert for a target zone and log the corresponding entry to DT. Still, the zone's real owner can actively monitor the log and take action to defeat such attack attempts.

Theorem 2 *For a DT-logged zone $\mathtt{ZN}_x$ with $\mathtt{RCrt}_x$ in epoch $\mathtt{T}$ and its delegation status not updated between $\mathtt{T}$ and $\mathtt{T}+k$ ($k>0$), if a client accepts an answer for a name under $\mathtt{ZN}_x$ using $\mathtt{RCrt}_y$ in epoch $\mathtt{T}+k$, then it must be that $\mathtt{RCrt}_x$=$\mathtt{RCrt}_y$, even if an $\mathcal{A}_1+\mathcal{A}_2+\mathcal{A}_3+\mathcal{A}_4$ attacker formed by the entities specified in $\mathtt{RCrt}_y$ is present between epoch $\mathtt{T}$ and $\mathtt{T}+k$.*

There are several technicalities in the definition above. First, between epoch $\mathtt{T}$ and $\mathtt{T}+k$ ($k>0$), $\mathtt{RCrt}_x$ is zone $\mathtt{ZN}_x$'s only valid certificate whose secure digest is logged in DT; $\mathtt{ZN}_x$ may have been delegated and updated before $\mathtt{T}$. Second, because of the hierarchical naming structure, $\mathtt{ZN}_y$ is either $\mathtt{ZN}_x$ or its ancestor, but not an arbitrary zone. Third, the attacker is defined with respect to the $\mathtt{RCrt}_y$ received by the client instead of $\mathtt{RCrt}_x$. This captures the reality that a client has no prior knowledge of an RCert's validity.

Theorem 2 formalizes data authenticity for established zones. It covers various scenarios where an attacker may acquire and use invalid, fraudulent or outdated RCerts to trick clients into accepting bogus data. RHINE maintains security

Table 4: Summary of our implementation in Go.

Component	LoC	Supporting System	Used by
librhine	2.2K	gRPC, CBOR [26]	Common
rmanager	0.6K	BadgerDB [4]	Zone owner
rhine-ca	0.68K	BadgerDB	CA
dt-log	0.76K	BadgerDB, SMT [10]	Log operator
rserver	0.35K	CoreDNS [7]	DNS operator
rresolv	0.7K	SDNS [16]	DNS operator
rdig	1K	miekg/dns [41]	End user

as long as one of the involved loggers stays non-compromised. This assumption is much weaker than that of DNSSEC, which rests on every node on the chain being honest.

Formal Verification. An informal argument or even pen-and-paper proof of these theorems can hardly lead to high assurance of RHINE's security guarantees. We have formally proved them using the Tamarin prover [68], an advanced tool for the verification of security protocols [22, 31, 42]. This approach helped us identify many subtle flaws in our early designs. We have modeled all the core protocols of RHINE, covering the secure setup and update of a zone delegation, the aggregation and retrieval of the DT log, as well as the authenticated distribution and resolution of the zone's data. This amounts to around 1500 lines of formal specification. We refer the reader to Appendix B for further details.

7 Implementation

We developed a prototype of RHINE. Table 4 summarizes our implementation efforts and the system's dependencies. The lines of code (LoC) reported do not count the supporting systems. Our prototype provides two software suites.

Offline Management. This suite includes four components. The library `librhine` defines common data structures and utilities. `rmanager` is intended to be an all-in-one toolbox for zone owners: key registration and delegation approval (in parent mode), request generation and validation (in child mode), log data retrieval, etc. `rhine-ca` offers all functions needed by a CA. `dt-agg` realizes a DT logger. It implements a self-balancing MHT for `DSA` (the per-zone accumulator) and a sparse MHT for `GDA` (the global accumulator).

These components operate in synchronous mode. For every protocol instance, they each create a `goroutine` that blocks itself after sending out a request and resumes upon receiving a response. All components can handle concurrent requests up to the available computing, memory, and bandwidth resources.

Name Resolution. We implement our nameserver (`rserver`) and recursive resolver (`rresolv`) with existing DNS frameworks. RHINE introduces new data types, RCert (encoded in the `X.509` format) and DSP. We store them using `TXT` records encoded as base64 strings and call them RoA (realm-of-authority) records. For DSP we store `DSum` and the membership proofs from `DSA` separately, as the latter are not required in most cases. Each zone has just one `DSum`, whereas the number of membership proofs equals the number of delegated child zones. Below are example records for zone `eg.com`.

```
_rcert.eg.com 60 IN TXT "Ed25519 MIIBITCB..."
_dsum.eg.com  60 IN TXT "rZXAwGzEZMBcGA1U..."
eg.com        60 IN DNSKEY 257 3 15 8fcCpq...
eg.com        60 IN RRSIG DNSKEY 15 2 60 2...
abc.eg.com    60 IN TXT "DSAPf u+EuVu6xX+..."
xyz.eg.com    60 IN TXT "DSAPf t9GsbAeavK..."
```

We do not use the private key of an RCert to directly sign regular records but instead treat this key as an equivalent of DNSSEC's key signing key (KSK). A KSK authenticates a zone signing key (ZSK), which in turn signs regular records. The record `eg.com` of type `DNSKEY` in the example above is a ZSK, followed by a `RRSIG` record authenticating it using the RCert. This layer of indirection in key usage allows a zone owner to securely store an RCert's private key offline and fetch it on demand, reducing the risk of security breaches.

We modified two built-in plugins of `CoreDNS` for `rserver` to serve RHINE-related data: the `file` plugin parses RoA records loaded from a zone file and, when processing a query, places them in the additional section of a response message; the `sign` plugin provides signing functions using RCerts.

`rresolv` augments `SDNS` with the functions to query, cache, validate and serve RoA records, reusing most of its codebase for the resolution of regular DNS records. `rresolv` always validates authoritative answers received from nameservers using RoA records and caches only verified data. It also always attaches the corresponding RoA records in the response message to a client, enabling E2E data authentication by default.

On the client side, we developed `rdig`, a `dig`-like tool for DNS-style name lookup with mandatory data validation.

8 Performance Evaluation

We evaluated our RHINE prototype in a private cloud network with 2Gbps bandwidth, using cloud servers with dedicated 8-core CPU (2.6GHz) and 16GB RAM running Ubuntu 22. Unless otherwise specified, the round-trip time (RTT) as reported by `ping` between any pair of servers is expanded to 100 ms using the `tc` utility. For the cryptographic algorithms in both RHINE and DNSSEC, we use Ed25519 for digital signatures and SHA256 for secure hash functions. In line with RHINE's architecture, the evaluation consists of two independent parts for offline and online operations.

8.1 Offline Management Performance

The first part of our evaluation aims to answer two questions. (1) *Can RHINE's offline protocols cope with real-world certificate issuance rates?* (2) *Is DT practical and scalable in terms of computation, communication, and storage cost?*

RCert Issuance. We measured the throughout of the secure delegation setup protocol, using two servers to run `rmanager`

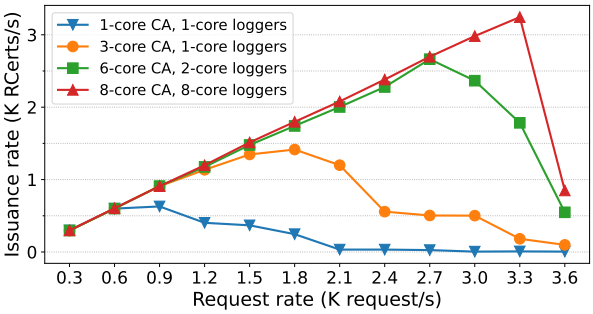


Figure 8: RCert issuance throughput.

(one for the child and the other for the parent), one server to run `rhine-ca`, and two servers to run `dt-log`. The child server generates delegation setup requests for predefined child zones whose keys have been registered at the parent server. The log maintained at the logger servers is initialized with an entry for the parent zone without any child. We limit the number of CPU cores used by the most critical CA and logger servers to understand their scaling behavior.

Figure 8 reports the results. With one core, the CA is the slowest server capping the issuance rate at around 600 RCerts per second. Using three cores, it can catch up with the loggers for a throughput of 1.4K RCerts per second. Doubling this configuration also doubles the achievable throughput. The decay in throughput with increasing request rates is due to the child server overwhelming itself with too many pending requests. Overall, our setup with these 8-core servers can issue a maximum of 3.3K RCerts per second.

To put this number in context, we consider the performance requirement of Let's Encrypt, the largest ACME-backed CA that accounts for around 80% (5M) of all daily logged CT entries [11]. Our test servers with moderate resources can already issue nearly 12M RCerts per hour. This indicates that our design can easily cope with real-life certificate issuance workloads. Such a performance is explained by RHINE's streamlined issuance process, which unlike ACME does not involve a time-consuming challenge-response procedure.

DT Consensus. We evaluate the performance-critical DT consensus process with different numbers of loggers (each on a separate server). We consider only delegation setup requests as the main consensus routine is agnostic to the type of requests. We limit the bandwidth between each pair of loggers to 1Gbps to simulate a common network setup. Each server pre-loads 50K requests into its memory before the protocol starts. With $n = 5$ and $f = 2$, it takes merely 54 seconds for the honest loggers to achieve consensus. The time increases slightly to 71 seconds with two more honest loggers. When considering one more faulty node ($n = 7, f = 3$), the consensus process finishes after 208 seconds. This trend in performance is expected as more faulty nodes mean more rounds of message exchanges in the consensus routine. Assuming the loggers run instances of the protocol consecutively with input batches of size 50K, it will take roughly 2.5 hours to process

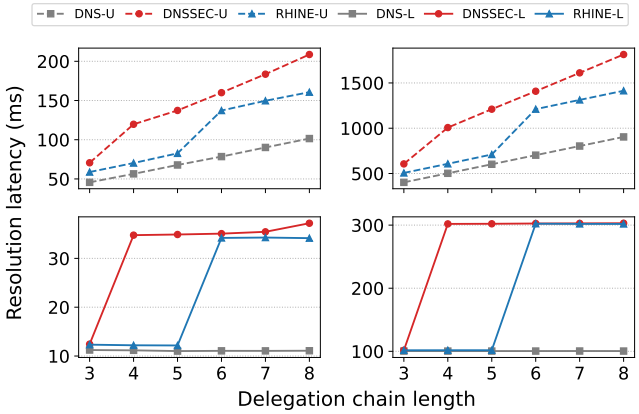


Figure 9: Upper and lower bounds of resolution latency under two network settings (left: RTT=10 ms; right: RTT=100 ms). The similar trends on the left and right plots suggest that network delay dominates the overall resolution latency.

2M requests. This meets the above-mentioned requirement for daily certificate issuance.

We observe that bandwidth is the determining factor for the overall performance, as the protocol runs n consensus routines in parallel. Yet, the bottleneck in our experiment setup is the memory size of individual servers, each of which needs to cache the input from all others. With more memory, the servers can exchange larger batches of requests. We can thus expect higher performance in a production environment with more powerful hardware.

Log Size. We estimate the DT log's overall size by taking into account the `DSums` and `DSAs` of all zones as well as the `GDA`. The size is determined by the distribution of zone delegations. The number of children of each TLD is publicly known from domain registries (e.g., 159M for `.com`) [13], but zone enumeration is required to learn the exact number of children for SLDs and further subzones. Developing an accurate view of the global DNS delegation tree is a challenging task in itself, and we leave it for future work.

Our estimation uses the statistics collected by enumerating sample zones from the Tranco list [74] and assumes an exponential decay in delegation: on average, x% of all SLDs have 4 children (and the rest of them have no child), x% of all third-level domains have 3 children, and so forth. The DT log's size is estimated to be 48GB with $x = 1$, 75GB with $x = 10$, and 779GB with $x = 50$ (which is likely an overestimation). This is only a fraction of the space requirement of CT logs, each of which can consume TBs of storage [5].

8.2 Name Resolution Performance

The second part of our evaluation investigates how E2E authentication affects name resolution performance from the perspectives of end users and naming service operators.

We compare RHINE with plain DNS and DNSSEC. For the

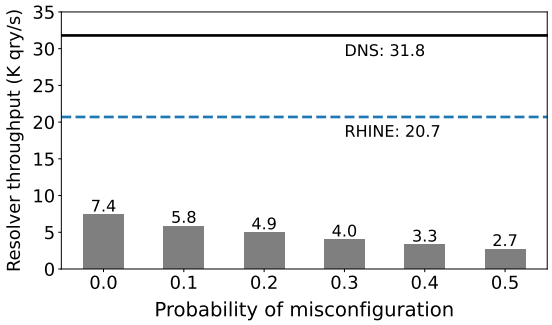


Figure 10: The resolver's query processing capacity in different systems. For DNSSEC (data shown as bars), we vary the probability of inconsistency between a zone and its children.

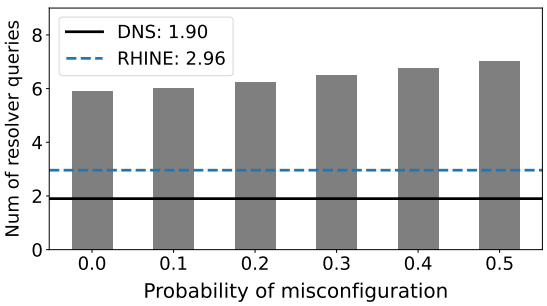


Figure 11: The average number of resolver queries per client request in case of cache miss. DNSSEC data is shown in bars.

latter two, we use the unmodified `CoreDNS` as the nameserver and `SDNS` as the resolver. For DNSSEC, we implement a validating client and modify `SDNS` to always return a complete authentication chain. For RHINE, we consider fast validation without membership proof as this covers the vast majority of cases (Section 5.4). The resolver is preloaded with a root zone key for DNSSEC and a CA certificate that is used to sign RCerts. The experiments used synthetic zone files populated with random resource records. Each record's name contains one more label than its residing zone. All labels have a fixed length of 6, the average calculated from the Tranco list [74].

Resolution Latency. End users are sensitive to the latency of name lookup queries. We inspect the bounds of resolution latency as determined by the resolver's cache. The upper bound is obtained when a resolver iteratively queries all the relevant nameservers for a non-cached record. The lower bound is obtained when a resolver returns an answer directly from its cache. We set up eight nameservers, which host a delegation chain of zones, one client, and one resolver. We run the experiments with two network settings: one with the RTT between the cloud servers expanded to 10 ms, and the other to 100 ms. The data reported for each experiment is averaged over 100 trials. Figure 9 depicts the results.

As can be seen, RHINE constantly outperforms DNSSEC, and its performance edge comes mainly from the savings in network communication. If a UDP message carrying a DNS response exceeds the size limit (512 bytes by default [35]), the client will retry the query using TCP. Since RHINE has fewer data authenticating records than DNSSEC, retransmission is triggered less frequently. This advantage is most pronounced in case of cache hits. RHINE can sustain its negligible cost over plain DNS for longer delegation chains than DNSSEC. The performance gap will further increase should more expensive cryptographic algorithms be used. In fact, most DNSSEC-signed zones still use RSA signatures [70], which are much larger than the Ed25519 signatures used in our evaluation.

Resolver Overhead. Since augmenting regular name resolution with E2E data authentication requires the most changes to a resolver's behavior, we focus on analyzing the performance overhead of a validating resolver. Our experiments use separate cloud servers for one client (running `dnsperf` [8] as the load generator), four nameservers each hosting a level of zones from the root to third-level zones (aka subdomains, which are common in modern cloud-based web services [45]), and one resolver under test. More specifically, we generate 15 TLDs each with 8K SLDs; each SLD further delegates to one third-level zone with one `A` record for a terminal name. The client's query trace contains 480K names randomly sampled from the 120K terminal names. The resolver's cache size is set as 100K[1] and the cache is warmed up by querying all terminal names once before each experiment. With this setup, we manage to maintain a typical cache hit ratio of around 80% for client queries [30, 56]. For DNSSEC, we simulate common security mismanagement that causes inconsistencies in authentication chains and hence validation failure [54], by programming the nameservers to return incorrect `DS` records with a given probability.

Figure 10 reports the resolver's throughput in terms of the number of client queries processed per second. The results indicate that RHINE has a moderate impact on the resolver's processing capacity, with a reduction of 34.9% in the overall throughput. Suffering from higher costs to retrieve and validate authentication chains, DNSSEC already reduces the throughput by 76.7% with successful validation and by even wider margins as the failure rate rises.

Figure 11 reports the average number of queries that the resolver sends to authoritative nameservers in case a client query cannot be directly answered from the cache. RHINE introduces roughly one additional query per client request (35.8% increase); this is attributed to the retrieval of RoA records (Section 7). DNSSEC increases the resolver's query load by 2× even in the absence of validation failure. Such overhead is higher than expected and can be explained by the

[1] `SDNS` has several cache instances for different purposes. What matter in our experiments are the primary `PCache` for `A` and `RRSIG` records as well as the `NSCache` for `NS` and `DS` records; for RHINE we have another `RoACache`. All their sizes are set as 100K. The only exception is that for the evaluation of DNSSEC we set the size of `PCache` as 190K, in order to maintain the 80% client query cache hit ratio for fair comparison. This is because `SDNS` also stores DNSSEC's `DNSKEY` records in `PCache`. With this setup, our measurements show that for DNSSEC, half of all the attempts to look up `DNSKEY` records in `PCache` fail and thereby trigger extra queries.

contention between DNSKEY and A records in the cache.

Our measurement results suggest that a fine-grained cache design, which separates security-related records from regular records, is crucial to a validating resolver's performance.

9 Related Work

Authenticated Naming Services. Building a decentralized naming service over a peer-to-peer (P2P) network was first attempted by CoDoNS [75]. It adopts DNSSEC for data authentication and thus suffers from the same drawbacks. The GNU Name System (GNS) [14] is a modern incarnation of this idea. It allows one to define a zone using a unique key and create its own namespace rooted at the zone. However, when used for a global consistent namespace, GNS will establish a chain of trust similarly to DNSSEC with the same fragility problems. Deploying these radical systems is well recognized as a practical challenge [43]. In contrast, RHINE can be deployed on the existing DNS infrastructure.

Several projects use blockchain to design tamper-proof naming systems [9, 12, 55], but whether they can achieve the same level of performance and scalability as DNS remains open. Donovan and Feamster [38] propose to reduce the overhead of DNSSEC by letting resolvers trust each other for signed records, but this does not provide E2E authentication.

In an early position paper [39], Fetzer et al. re-purpose SSL certificates to sign DNS records, each with a separate certificate. Yet, the authors only sketched a preliminary scheme, without thoroughly exploring the challenges and large design space of authenticating DNS with the web PKI as we do.

DNS and PKI. The interplay between DNS and PKIs has a long history. DANE allows a DNS domain to certify its own certificates using DNSSEC [48]. To reduce the risk of users accepting misissued certificates, a domain can also specify the CA authorized by it using the CAA record [44]. A fundamental problem with these designs is that they move users' trust from CAs to DNS authorities. It is unlikely that the latter are more trustworthy, since they are not even in the security business as CAs are. RHINE does not simply reverse the flow of trust, i.e., relying on CAs for the authentication of name data, but rather creates a robust system where all authorities counterbalance each other's power over the namespace.

Transparency Logs. With the widespread deployment of CT, transparency logs have proven to be an integral part of modern PKIs. Many enhancements to their functionality, security, and performance have been proposed. CIRT [80] extends CT to store certificate revocation information. CTng [63] and LogPicker [36] aim to relax the trust assumptions in CT by having multiple entities (loggers or monitors) attest log entries. DT's design also follows this generic approach. F-PKI [28] introduces a map server to provide a global view of all certificates and associated policies; this role is akin to a DT logger. New authenticated data structures are proposed to improve the efficiency of transparency logs [51, 84]. They can be potentially incorporated into DT for performance improvement.

Formal Analysis of PKI. Bhargavan et al. formally model an early draft of the ACME protocol and discovered several attacks [24]. Our formal approach has also helped us identify many subtle flaws in RHINE's early designs. ARPKI [21] and DTKI [89] are PKI designs formally verified with Tamarin. Compared with them, RHINE involves more subtly interacting entities and hence is more challenging to model and verify.

10 Conclusion and Discussion

After much recent activity in the DNS space (e.g., discovery of new attacks, frequent service outages, and growing concerns about privacy), a window of opportunity is opening up for a fundamental re-design of DNS to achieve high levels of security. We revisit the existing security architecture of DNS through a modern lens, pinpointing the intrinsic limitations therein and proposing RHINE as our solution to these long-standing problems. It offloads the heavy and error-prone authentication task away from the client-facing side of DNS, enabling efficient authenticated name resolution all the way to end hosts. The deployment of RHINE can bootstrap the security of today's web PKI and Internet at large.

Deployment. We briefly discuss the incentives and costs for different entities to deploy RHINE. There is no doubt that the global Internet community shares a common interest for E2E-authenticated name resolution.

From the perspective of DNS zone owners and operators, RHINE can offer better security, lower failure rate and operational costs, and more robust naming services than DNSSEC; RHINE indeed requires fewer changes to their existing hardware and software, because it obviates the need to frequently maintain, distribute, and validate DNSSEC-style authentication chains in regular operation. The extra investments in operating the offline part of RHINE, which can be fully automated with our well-defined protocols similarly to ACME, are comparable to what is already required by the web PKI.

CAs are likely among the most enthusiastic proponents of RHINE, because the issuance of RCerts will significantly expand and consolidate their security services. The operators of transparency logs have the same motivations; DT loggers will be a select subset of CT loggers.

For end users, RHINE is easier to adopt than DNSSEC because they already rely heavily on the web PKI. For instance, the RCert-based data validation function can be easily integrated into web browsers, which have built-in support for name lookup (e.g., DoT and DoH) and certificate validation.

Acknowledgment

We would like to thank the reviewers and our shepherd, Matthew Caesar, for their valuable comments that help us substantially improve the paper. We gratefully acknowledge support from ETH Zurich, and from the Zurich Information Security and Privacy Center (ZISC).

References

[1] DNSSEC Targeted in DNS Reflection, Amplification DDoS Attacks. https://community.akamai.com/, 2016.

[2] The DELEGATION_ONLY DNSKEY flag. Internet-Draft draft-ietf-dnsop-delegation-only-02, Internet Engineering Task Force, 2021.

[3] Apple's Certificate Transparency Policy. https://support.apple.com/en-us/HT205280, 2022.

[4] BadgerDB: Fast key-value DB in Go. https://github.com/dgraph-io/badger, 2022.

[5] Cert Spotter Stats. https://sslmate.com/resources/certspotter_stats, 2022.

[6] Chrome Certificate Transparency Policy. https://googlechrome.github.io/CertificateTransparency/ct_policy.html, 2022.

[7] CoreDNS: DNS and Service Discovery. https://coredns.io, 2022.

[8] DNS-OARC/dnsperf: DNS Performance Testing Tools. https://github.com/DNS-OARC/dnsperf, 2022.

[9] Ethereum Name Service. https://ens.domains, 2022.

[10] FPKI/SMT Implementation. https://github.com/netsec-ethz/fpki/tree/smt, 2022.

[11] Merkle Town. https://ct.cloudflare.com, 2022.

[12] Namecoin. https://www.namecoin.org, 2022.

[13] Registrar Stats. https://www.domainstate.com/registrar-tld-breakup.html, 2022.

[14] The GNU Name System. https://www.gnunet.org/en/gns.html, 2022.

[15] Use of DNSSEC Validation for World. https://stats.labs.apnic.net/dnssec/, 2022.

[16] Yasar Alev. SDNS: Privacy important, fast, recursive dns resolver server with dnssec support. https://github.com/semihalev/sdns, 2022.

[17] Eihal Alowaisheq, Siyuan Tang, Zhihao Wang, Fatemah Alharbi, Xiaojing Liao, and XiaoFeng Wang. Zombie Awakening: Stealthy Hijacking of Active Domains through DNS Hosting Referral. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[18] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), March 2005. Updated by RFCs 6014, 6840.

[19] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard), March 2005. Updated by RFCs 4470, 6014, 6840, 6944, 9077.

[20] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. Automatic Certificate Management Environment (ACME). RFC 8555 (Proposed Standard), March 2019.

[21] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. Design, Analysis, and Implementation of ARPKI: an Attack-Resilient Public-Key Infrastructure. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2017.

[22] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV Standard: Break, Fix, Verify. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[23] Daniel J. Bernstein. DNSCurve: Usable Security for DNS. https://dnscurve.org.

[24] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Nadim Kobeissi. Formal Modeling and Verification for Domain Validation and ACME. In *Processings of the International Conference on Financial Cryptography and Data Security (FC)*, 2017.

[25] Kevin Borgolte, Tobias Fiebig, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2018.

[26] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949 (Internet Standard), December 2020.

[27] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. Domain Validation++ For MitM-Resilient PKI. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[28] Laurent Chuat, Cyrill Krähenbühl, Prateek Mittal, and Adrian Perrig. F-PKI: Enabling Innovation and Trust Flexibility in the HTTPS Public-Key Infrastructure. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2022.

[29] Taejoong Chung, Roland van Rijswijk-Deij, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. A Longitudinal, End-to-End View of the DNSSEC Ecosystem. In *Proceedings of the USENIX Security Symposium*, 2017.

[30] Secure64 Software Corporation. Lies, Damn Lies and DNS Performance Statistics. White paper, 2017.

[31] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[32] David Dagon, Chris Lee, Wenke Lee, and Niels Provos. Corrupted DNS Resolution Paths: The Rise of a Malicious Resolution Authority. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2008.

[33] Tianxiang Dai, Philipp Jeitner, Haya Shulman, and Michael Waidner. From IP to Transport and beyond: Cross-Layer Attacks against Applications. In *Proceedings of the ACM SIGCOMM Conference*, 2021.

[34] David Derler, Christian Hanser, and Daniel Slamanig. Revisiting Cryptographic Accumulators, Additional Properties and Relations to Other Primitives. In *Topics in Cryptology — CT-RSA 2015*, 2015.

[35] J. Dickinson, S. Dickinson, R. Bellis, A. Mankin, and D. Wessels. DNS Transport over TCP - Implementation Requirements. RFC 7766 (Proposed Standard), March 2016. Updated by RFCs 8490, 9103.

[36] Alexandra Dirksen, David Klein, Robert Michael, Tilman Stehr, Konrad Rieck, and Martin Johns. LogPicker: Strengthening Certificate Transparency Against Covert Adversaries. *Proceedings on Privacy Enhancing Technologies*, 2021(4):184–202, 2021.

[37] D. Dolev and A. Yao. On The Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[38] Sean Donovan and Nick Feamster. Alternative Trust Sources: Reducing DNSSEC Signature Verification Operations with TLS. *ACM SIGCOMM Computer Communication Review*, 45(4):353–354, 2015.

[39] Christof Fetzer, Gert Pfeifer, and Trevor Jim. Enhancing DNS security using the SSL trust infrastructure. In *Proceedings of the IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2005.

[40] Marc Frei, Jonghoon Kwon, Seyedali Tabaeiaghdaei, Marc Wyss, Christoph Lenzen, and Adrian Perrig. G-sinc: Global synchronization infrastructure for network clocks. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, 2022.

[41] Miek Gieben. DNS library in Go. https://github.com/miekg/dns, 2022.

[42] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David A. Basin. A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols. In Srdjan Capkun and Franziska Roesner, editors, *Proceedings of the USENIX Security Symposium*, 2020.

[43] Christian Grothoff, Matthias Wachs, Monika Ermert, and Jacob Appelbaum. Toward Secure Name Resolution on The Internet. *Computers & Security*, 77:694–708, 2018.

[44] P. Hallam-Baker and R. Stradling. DNS Certification Authority Authorization (CAA) Resource Record. RFC 6844 (Proposed Standard), January 2013. Obsoleted by RFC 8659.

[45] Shuai Hao, Haining Wang, Angelos Stavrou, and Evgenia Smirni. On the DNS Deployment of Modern Web Services. In *Proceedings of the IEEE Conference on Network Protocols (ICNP)*, 2015.

[46] Muks Hirani, Sarah Jones, and Ben Read. Global DNS Hijacking Campaign: DNS Record Manipulation at Scale. https://www.fireeye.com/blog/threat-research/, 2019.

[47] P. Hoffman and P. McManus. DNS Queries over HTTPS (DoH). RFC 8484 (Proposed Standard), October 2018.

[48] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), August 2012. Updated by RFCs 7218, 7671, 8749.

[49] Hans Hoogstraaten, Ronald Prins, Daniël Niggebrugge, Danny Heppener, Frank Groenewegen, Janna Wettink, Kevin Strooy, Pascal Arends, Paul Pols, Robbert Kouprie, Steffen Moorrees, Xander van Pelt, and Yun Zheng Hu. Black Tulip: Report of the Investigation into the DigiNotar Certificate Authority Breach. Technical report, August 2012.

[50] Rebekah Houser, Shuai Hao, Zhou Li, Daiping Liu, Chase Cotton, and Haining Wang. A Comprehensive Measurement-based Investigation of DNS Hijacking. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, 2021.

[51] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A Low-Latency Transparency Log System. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[52] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858 (Proposed Standard), May 2016. Updated by RFC 8310.

[53] Geoff Huston. Measuring DNSSEC Performance. https://labs.apnic.net, 2013.

[54] IANIX. Major DNSSEC Outages and Validation Failures. https://ianix.com/pub/dnssec-outages.html, 2021.

[55] Lin Jin, Shuai Hao, Yan Huang, Haining Wang, and Chase Cotton. DNSonChain: Delegating Privacy-Preserved DNS Resolution to Blockchain. In *Proceedings of the IEEE Conference on Network Protocols (ICNP)*, 2021.

[56] Jaeyeon Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and The Effectiveness of Caching. *IEEE/ACM Transactions on Networking*, 10(5):589–603, 2002.

[57] Dan Kaminsky. It's the end of the cache as we know it. Presented at Black Hat USA, 2008.

[58] Aqsa Kashaf, Vyas Sekar, and Yuvraj Agarwal. Analyzing Third Party Service Dependencies in Modern Web Services: Have We Learned from the Mirai-Dyn Incident? In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[59] Cyrill Krähenbühl, Seyedali Tabaeiaghdaei, Christelle Gloor, Jonghoon Kwon, Adrian Perrig, David Hausheer, and Dominik Roos. Deployment and scalability of an inter-domain multi-path routing infrastructure. In *Proceedings of the International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2021.

[60] Jonghoon Kwon, Juan A. García-Pardo, Markus Legner, François Wirz, Matthias Frei, David Hausheer, and Adrian Perrig. SCIONLab: A next-generation Internet testbed. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*.

[61] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013. Obsoleted by RFC 9162.

[62] B. Laurie, E. Messeri, and R. Stradling. Certificate Transparency Version 2.0. RFC 9162 (Experimental), December 2021.

[63] Hemi Leibowitz, Haitham Ghalwash, Ewa Syta, and Amir Herzberg. CTng: Secure Certificate and Revocation Transparency. Cryptology ePrint Archive, Paper 2021/818, 2021.

[64] Wilson Lian, Eric Rescorla, Hovav Shacham, and Stefan Savage. Measuring the Practical Impact of DNSSEC Deployment. In *Proceedings of the USENIX Security Symposium*, 2013.

[65] Baojun Liu, Chaoyi Lu, Haixin Duan, Ying Liu, Zhou Li, Shuang Hao, and Min Yang. Who Is Answering My Queries: Understanding and Characterizing Interception of the DNS Resolution Path. In *Proceedings of the USENIX Security Symposium*, 2018.

[66] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[67] Keyu Man, Xin'an Zhou, and Zhiyun Qian. DNS Cache Poisoning Attack: Resurrections with Side Channels. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[68] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2013.

[69] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Proceedings of Advances in Cryptology (CRYPTO)*, 1988.

[70] Moritz Müller, Willem Toorop, Taejoong Chung, Jelte Jansen, and Roland van Rijswijk-Deij. The Reality of Algorithm Agility: Studying the DNSSEC Algorithm Life-Cycle. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[71] Jeman Park, Aminollah Khormali, Manar Mohaisen, and Aziz Mohaisen. Where Are You Taking Me? Behavioral Analysis of Open DNS Resolvers. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[72] Paul Pearce, Ben Jones, Frank Li, Roya Ensafi, Nick Feamster, Nick Weaver, and Vern Paxson. Global Measurement of DNS Manipulation. In *Proceedings of the USENIX Security Symposium*, 2017.

[73] Adrian Perrig, Peter Müller, Samuel Hitz, David Hausheer, David Basin, Markus Legner, and Laurent Chuat. *The Complete Guide to SCION*. Springer, 2022.

[74] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2019.

[75] Venugopalan Ramasubramanian and Emin Gün Sirer. The Design and Implementation of a next Generation Name Service for the Internet. In *Proceedings of the ACM SIGCOMM Conference*, 2004.

[76] Venugopalan Ramasubramanian and Emin Gün Sirer. Perils of transitive trust in the domain name system. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2005.

[77] Audrey Randall, Enze Liu, Gautam Akiwate, Geoffrey M Voelker, Stefan Savage, and Aaron Schulman. Home is Where the Hijacking is: Understanding DNS Interception by Residential Routers. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2021.

[78] rhine-team. https://github.com/rhine-team/RHINE-Prototype, 2022.

[79] Jeremy Rowley. CT2 Log Compromised via Salt Vulnerability. https://groups.google.com/a/chromium.org/g/ct-policy/c/aKNbZuJzwfM, 2020.

[80] Mark Dermot Ryan. Enhanced Certificate Transparency and End-to-End Encrypted Mail. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2014.

[81] Lorenz Schwittmann, Matthäus Wander, and Torben Weis. Domain Impersonation is Feasible: A Study of CA Domain Validation Vulnerabilities. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[82] Haya Shulman and Michael Waidner. One Key to Sign Them All Considered Vulnerable: Evaluation of DNSSEC in the Internet. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[83] Thomas and Erin. Against DNSSEC. https://sockpuppet.org/blog/2015/01/15/against-dnssec/, 2015.

[84] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency Logs via Append-Only Authenticated Dictionaries. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[85] Thomas Vissers, Timothy Barron, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The Wolf of Name Street: Hijacking Domains Through Their Nameservers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[86] Joel Wanner, Laurent Chuat, and Adrian Perrig. A Formally Verified Protocol for Log Replication with Byzantine Fault Tolerance. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, 2020.

[87] Nicholas Weaver, Christian Kreibich, and Vern Paxson. Redirecting DNS for Ads and Profit. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2011.

[88] B. Wellington and O. Gudmundsson. Redefinition of DNS Authenticated Data (AD) bit. RFC 3655 (Proposed Standard), November 2003. Obsoleted by RFCs 4033, 4034, 4035.

[89] Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: A New Formalized PKI with Verifiable Trusted Parties. *The Computer Journal*, 59(11):1695–1713, 2016.

[90] Xiaofeng Zheng, Chaoyi Lu, Jian Peng, Qiushi Yang, Dongjie Zhou, Baojun Liu, Keyu Man, Shuang Hao, Haixin Duan, and Zhiyun Qian. Poison Over Troubled Forwarders: A Cache Poisoning Attack Targeting DNS Forwarding Devices. In *Proceedings of the USENIX Security Symposium*, 2020.

0. X (the owner of $\texttt{ZN}_x$) publishes $\texttt{RCrt}_x$, $\texttt{DSum}_x$, $\texttt{DSA}_x$ to the infrastructure D to be queried by client U.
1. $U \rightarrow D$: $\texttt{QUERY}(\texttt{qname}, \texttt{qtype})$
2. D : $rec \coloneqq \langle \texttt{RRset}(\texttt{qname}, \texttt{qtype}) \rangle_x$
   : $dsp \coloneqq \langle \texttt{DSum}_x, T \rangle_{\mathcal{L}_x}$
   // ALv is encoded with four bits: DOL, IND, TER, EOI
   // The only case requiring a membership proof
   : If $\texttt{ALv}_x$ = (IND, ¬DOL, ¬TER, ¬EOI):
   // $\texttt{ZN}_y$ is a (potential) child zone enclosing qname
   : $\texttt{ZN}_y \coloneqq \texttt{GetChild}(\texttt{ZN}_x, \texttt{qname})$
   // Get the membership proof for $\texttt{ZN}_y$'s (non-)existence
   : $mem \coloneqq \texttt{Acc.GenPf}(\texttt{DAcc}_x, \texttt{ZN}_y)$
   : $dsp \coloneqq dsp \cup \{mem\}$
3. $D \rightarrow U$: $\texttt{ANSWER}(rec, \texttt{RCrt}_x, dsp)$
   // $\texttt{ZN}_x$ is extracted from $\texttt{RCrt}_x$
4. U : Verify $\texttt{ZN}_x$ encloses qname
   : $\wedge$ $\Sigma.\texttt{Vf}(\texttt{RCrt}_x, rec) \wedge \Sigma.\texttt{Vf}(pk_{\mathcal{L}_x}, dsp)$
   : $\wedge$ T is the current epoch
   : $\wedge$ $\texttt{ZN}_x$ is not revoked (from $\texttt{Aux}_x$)
   : $\wedge$ $\texttt{Match}(\texttt{DSum}_x, \texttt{RCrt}_x)$
   : // Short-cut cases:
   : If DOL in $\texttt{ALv}_x$ and qname $\neq \texttt{Apex}(\texttt{ZN}_x)$:
   : Reject the answer
   : Else if ¬IND or TER or EOI in $\texttt{ALv}_x$:
   : Accept the answer
   : Else : // Otherwise, check $\texttt{ZN}_y$'s status
   : If mem is for absence :
   : Accept if $\texttt{Acc.Vf}(\texttt{DAcc}_x, mem)$
   : Else : // $\texttt{ZN}_y$ already delegated
   : Verify $\texttt{Acc.Vf}(\texttt{DAcc}_x, mem)$
   : Accept the answer if ¬IND in $\texttt{ALv}_y$

Figure 12: Authenticated name resolution protocol.

0. Each $L_i \in \mathcal{G}$ has a set $\mathcal{X}_i$ of requests ($lreq$) as input.
1. Run n rounds of $O_i \coloneqq \texttt{LogresConsensus+}(L_i, \mathcal{X}_i)$ with each L_i as the leader proposing input data in a round. After that, all loggers obtain the same set $O \coloneqq \bigcup_{i=1}^{n} O_i$ as output.
2. Each L_i filters the requests in O by keeping only the earliest one in case of conflicts, applies the resulting operations to its local $\texttt{GDA}_T$, and computes a new digest $\texttt{GAcc}_{T+1}$. Broadcast $\langle \texttt{GAcc}_{T+1} \rangle_{L_i}$ to $\mathcal{L}$
3. Each L_i accepts and finalizes the aggregation result if it receives f valid signatures over $\texttt{GAcc}_{T+1}$.

Figure 13: DT aggregation protocol.

A Protocol Specifications

A.1 Authenticated Name Resolution

RHINE's name resolution protocol is presented in Figure 12. It is agnostic to the underlying distribution infrastructure D, which can be instantiated, for example, with the existing DNS infrastructure (consisting of authoritative nameservers, recursive resolvers, forwarders, etc.) or a P2P network such as GNUnet [14]. RHINE mandates that each answer to a client query contains, in addition to the authoritative resource records, the RCert and DSP of the zone that claims the authority, and that a client always validates answers by itself, thereby enforcing E2E authentication.

A.2 Secure Delegation Update

Figure 14 describes RHINE's secure delegation update protocol. An established zone uses its own RCert and the associated DSP to prove its realm of authority. The protocol's overall flow resembles the delegation setup process, except that the parent's involvement is needed only in a few cases.

A zone can freely update its RCert and DT entry within the validity period of the delegation. The parameter to update must not include a delegation expiration time beyond what is currently specified in the zone's DSum. To extend the validity period before the delegation expires, a zone must negotiate with its parent (e.g., renewing the business contract) and get the latter's approval. Another type of update that requires the parent's consent is changing a non-independent zone to an independent one, as this affects the parent's realm of authority. The update of authority level is subject to additional restrictions. For example, a zone cannot change itself to terminating unless all its existing delegations become invalid (expired or revoked); similarly, a zone cannot change itself to end-of-independence if it still has any independent child. A zone can only update its delegation status (except its DAcc, which is affected by the changes to its subzones' delegation status) once per epoch. For ease of presentation, we abstract away these checks of an update request's legitimacy in the protocol specification.

It is possible for a zone to update the CA and loggers it relies on, which is important after security breaches of these trusted entities. In this case, the zone will run the update protocol with a set of new trusted entities.

The revocation of a secure delegation comes in two forms. Since RHINE mandates one RCert per zone at any time, the issuance of a new RCert for a zone implicitly revokes the old one. A zone can also make an explicit revocation request through the update protocol. This will fail the validation of the zone's data signed with its current RCert. The operation is irreversible, meaning that a revoked zone can only be re-established through the secure delegation setup protocol.

One caveat in enforcing one RCert per zone is that the loss of a zone's private key may lock up the zone until the existing delegation expires. This conundrum can be addressed by having a zone pre-generate a signed revocation request, preferably immediately after the delegation setup. The zone

0. C prepares a data object Upd that encodes the parameters to be updated.
   // A and $\mathcal{L}_c$ may be different from those in $\texttt{RCrt}_c$
1. C : Select A and $\mathcal{L}_c$
   // *rid* is implicitly included in all subsequent messages
   : $rid \coloneqq \texttt{H}(t_0, \texttt{ZN}_c, \texttt{Upd}, A, \mathcal{L}_c)$
   : $sur \coloneqq \langle rid, \texttt{SUReq}(\texttt{ZN}_c, \texttt{Upd}, A, \mathcal{L}_c)\rangle_c$
   : [Get $apv \coloneqq \langle \texttt{SUApprvl}(\texttt{H}(sur))\rangle_p$
   : and $\texttt{RCrt}_p$ from the parent P].
2. $C \to A$: sur, $\texttt{RCrt}_c$, [apv, $\texttt{RCrt}_p$]
3. A : Retrieve dsp_c [, dsp_p] from $\mathcal{L}_c$
   : Verify sur with $\texttt{RCrt}_c$ and dsp_c
   : [, apv with $\texttt{RCrt}_p$ and dsp_p]
   : Verify the validity of Upd
   : [, create a new *tbsrc* according to Upd]
   : $prl \coloneqq \langle \texttt{PreLog}(sur, [apv, tbsrc])\rangle_A$
4. $A \to \mathcal{L}_c$: prl, $\texttt{RCrt}_c$, [apv, $\texttt{RCrt}_p$]
5. L_i : Verify prl, sur, [and apv,] using the
   : corresponding certificates and the local log
   : Verify the validity of Upd
   : $uds \coloneqq (\texttt{T}, A, \mathcal{L}_c, \texttt{ZN}_c, \texttt{Upd}, [\texttt{H}(tbsrc)])$
6. $L_i \to A$: $att_i \coloneqq \langle \texttt{LogAttest}(L_i, \texttt{H}(uds))\rangle_{L_i}$
7. A : Verify $\{att_i\}$ against prl
   // L is randomly selected from $\mathcal{L}_c$ by A
8. $A \to L$: $lreq \coloneqq \langle \texttt{LogReq}(L, uds, \{att_i\}_{L_i \in \mathcal{L}_c})\rangle_A$
9. L : Verify $lreq$ and $\{att_i\}$ and their consistency
   : Add $lreq$ to the pending pool for aggregation
10. $L \to A$: $lc \coloneqq \langle \texttt{LogCfm}(L, \texttt{H}(uds))\rangle_L$
11. A : Verify lc against $lreq$
   : [, $\texttt{RCrt}'_c \coloneqq \langle \texttt{FinalRCert}(\texttt{TbsRC}_c, \mathcal{L}_c)\rangle_A$]
12. $A \to C$: prl, $\{att_i\}_{L_i \in \mathcal{L}_c}$, lc, [$\texttt{RCrt}'_c$]
13. C : Verify all received data against sur

Figure 14: Secure delegation update protocol (simplified). Messages and operations in square brackets [m] are optional.

can then revoke the existing delegation in case of key loss. Clearly, the pre-generated revocation request itself should be stored separately from the private key in a secure place.

A.3 DT Aggregation with Modified Logres

Figure 13 presents the DT aggregation protocol, with the core modified Logres consensus routine depicted in Figure 15. Logres obviates the need for leader selection by having each participating node lead and run an instance of the consensus routine in parallel with all others. Each consensus instance contains up to $f+1$ rounds of message exchanges among the nodes, where f is the number of Byzantine faulty nodes tolerable by the system. In normal situations where the leader is honest and correctly operates, the consensus routine will terminate in just two rounds.

Our main modification of Logres is in lines 13–17 of Figure 15, which describes the additional data validation required by RHINE. Only valid input values will be added to the output set. Another important change is that we refine the algorithm to allow taking sets of values as input, as the original version abstracts the input data as a single value. This entails several technicalities including whether to have nodes' witness on valid values that may only constitute a subset of the input. For simplicity, we always treat the witnesses on the original input set even it may contain invalid data. This results in redundant data validation in each round. Optimizing performance in this regard is an interesting future work.

The final output set O from the consensus process may contain duplicate or conflicting operations as a result of attacks (when RHINE's threat assumptions are violated; see Section 6) or operational faults. For example, two requests may contain different parameters to create a new log entry for a just delegated zone. This is possible because there is a delay for the DT log to be synchronized across loggers. In such situations, the loggers will keep the earliest operation and discarding other conflicting operations for the zone in question; any potential attacks and faults will become detectable once the current execution of the aggregation protocol ends.

B Formal Verification of RHINE

This section introduces important aspects of our formal specification of RHINE and the security properties we verify. We refer the reader to the project repository [78] for full details.

Abstractions. The formalization of non-trivial protocols using Tamarin can run into the state explosion problem, which makes the analysis intractable. To this end, we use several abstractions to reduce the complexity of our model while still faithfully capturing the essence of RHINE protocols.

First, rather than modeling the entire namespace, we focus on a few zones represented symbolically that suffice to describe generic zone delegation. This includes a parent zone that can be malicious, a child zone in question (i.e., $\texttt{ZN}_x$ in Theorem 1 and 2), and another child zone (of the same parent) serving to validate our model's correctness. We consider all of them to be independent zones for meaningful a security analysis. This also obviates the need to model the processing of authority level.

For the time dimension, we model three epochs. In T0, the pre-established parent zone can publish data for name resolution and approve delegation requests. Only one child zone can be delegated in T0 and the other in T1; the first child zone can also be updated in T1. Zone delegation or update is not permitted in the last epoch T2. These symbolic epochs are intended to enforce the sequence of events and they are not necessarily consecutive. This arrangement allows the model to capture security threats throughout a zone's life cycle.

For the DT aggregation process, we model only RHINE-specific input validation without specifying the Logres con-

0. This is one of the $|\mathcal{L}|$ parallel runs of the consensus process with L_i being the leader. The code is for L_j.
1. If $j = i$:
2. broadcast $\langle \mathcal{X}_i, L_i \rangle_{L_i}$ to all other loggers
3. return $\mathcal{O}_i := \mathcal{X}_i$
4. Else:
5. $\mathcal{W} := \emptyset$ // witnessed values
6. $\mathcal{O}_i := \emptyset$ // agreed-upon output

// Start $f+1$ rounds of message exchanging

7. For round $r = 1, \ldots, f+1$:

// A timeout is needed to stop waiting for the messages from faulty nodes.

8. $\mathcal{M} :=$ received messages
9. $\mathcal{V} := \emptyset$ // valid input values
10. For $\langle \mathcal{X}, L_i \rangle_{\mathcal{L}_W} \in \mathcal{M}$

// Logres-specific checking

11. If $|\mathcal{L}_W| \geq r \wedge \Sigma.\mathtt{Vf}(pk_{\mathcal{L}_w}, \langle \mathcal{X}, L_i \rangle_{\mathcal{L}_W})$
12. $\mathcal{Y} := \emptyset$

// RHINE-specific data validation. A, $\mathcal{L}_c$ are from op

13. For $lreq \in \mathcal{X}$:
14. $lreq := \langle \mathtt{LogReq}(L, op, attset) \rangle_A$
15. If $\Sigma.\mathtt{Vf}(pk_A, lreq) \wedge L = L_i$ $\wedge\ \Sigma.\mathtt{Vf}(pk_{\mathcal{L}_c}, attset)$ $\wedge\ \mathtt{Match}(op, attset)$
16. $\mathcal{Y} := \mathcal{Y} \cup \{lreq\}$
17. $\mathcal{V} := \mathcal{V} \cup \{\mathcal{Y}\}$

// Continue Logres processing on valid RHINE input

18. If $\mathcal{V} \setminus \mathcal{W} \neq \emptyset$:
19. If $|\mathcal{V} \cup \mathcal{W}| = 1$:
20. $\mathcal{O}_i := \mathcal{V}$ // Only one element in $\mathcal{V}$
21. Else :
22. $\mathcal{O}_i := \emptyset$ // No agreed-upon value yet
23. $\mathcal{N} := \emptyset$ // Messages for next round

// Add witness to the original input

24. For $\mathcal{Y} \in \mathcal{V} \setminus \mathcal{W}$:
25. Find $\langle \mathcal{X}, L_i \rangle_{\mathcal{L}_W} \in \mathcal{M}$ s.t. $\mathcal{Y} \subset \mathcal{X}$
26. $\mathcal{N} := \mathcal{N} \cup \{\langle \mathcal{X}, L_i \rangle_{\mathcal{L}_W \cup \{L_j\}}\}$
27. multicast $\mathcal{N}$ to all non-leader loggers

// Update witnessed value

28. $\mathcal{W} := \mathcal{V}$

// In the end of this protocol run, all honest loggers will have the same $\mathcal{O}_i$, which contain all valid log operations as a subset of $\mathcal{X}_i$.

29. return $\mathcal{O}_i$

Figure 15: The $\mathtt{LogresConsensus+}(L_i, \mathcal{X}_i)$ protocol.

sensus routine, as its security has been formally verified by its authors [86] and is inherited by our enhanced version. For the update protocol, we consider updating the certified key as well as the designated CA and loggers without the need for parental approval. This reflects a zone owner's ability to independently manage the zone's security.

We refrain from modeling explicit servers and the name resolution algorithm, as this would result in a overcomplicated model. Instead, we create an abstract distribution infrastructure by taking advantage of Tamarin's underlying pattern matching mechanism. Moreover, we represent all data structures (RCerts, DT log, resource records, etc.) as sets of values and omit non-essential data such as `Aux`.

B.1 Protocol Specification

Tamarin models a security protocol as a labeled transition system (LTS) where a state of the LTS consists of the local states of the protocol participants, the adversary's knowledge, and messages on the network. States are modeled as a finite multiset of *facts*. The system's dynamics are specified by labeled multiset rewriting rules that transform the facts.

Protocol Roles. Our model introduces five roles: `P` is the owner of an established parent zone, `C` is an entity wishing to securely establish a child zone, `CA` is an RCert issuer, `L` is a DT logger, and `U` is an end user trying to resolve a name under a child zone. The state space of a protocol in a symbolic Tamarin model generally grows exponentially with the number of involved actors, which are instances of roles in interleaved protocol sessions. We model `P` as a singleton that is instantiated only once. No limitation is imposed on other roles. RHINE allows a zone to choose the number m of relying loggers. We set $m = 2$ for all zones in the model. This keeps the complexity of verification manageable without weakening the security properties we verify. The other parameters f and n are irrelevant in the model because of the simplified aggregation process.

Adversary. Tamarin provides a built-in network model with a Dolev-Yao adversary: any outbound message is added to the adversary's permanent knowledge; any inbound message is constructed by the adversary from its knowledge. We leverage this feature to create an adversary-controlled distribution infrastructure without any explicit servers: publishing a zone simply means sending its signed records to the network, and name resolution is realized by sending a query to and receiving the matching record (and associated RCert and DSP) from the network. To model the compromise of an entity, we reveal its private key to the network, which enables the adversary to impersonate the entity by forging its signatures. We also allow malicious child zone owners so that the adversary's capability is not limited by the model itself.

Protocol Rules. We use several example rules to explain our modeling style and choices. Figure 16 lists two rules modeling the CA's processing of an initial request in the secure delegation protocol. A rule is defined in the form of

```
[state facts] --[event facts]-> [state facts].
  (premise)                     (conclusion)
```

The lines in a `let ... in` block defines macros that are expanded in the respective rule.

```
rule CA_Preissuance_1:
  let
    sdr_data = <'SDReq', epoch, zone, $C,
                zpkC, $P, $CA, $L1, $L2>
    sdr = <sdr_data, sig>
    apv_data = <'SDApproval', h(sdr)>
  in
  [
    In(<$C, $CA, sdr, apv, rcP>)
  , !CA_St_0($CA, ~skCA)
  , !ZPk_P($P, zpkP)
  , Fr(~dsrid)
  ]
  --[
      NotEq($CA, $L1)
    , NotEq($CA, $L2)
    , NotEq($L1, $L2)
    , Eq(verify(apv, apv_data, zpkP), true)
    , Eq(verify(sig, sdr_data, zpkC), true)
    ]->
  [
    CA_St_1($CA, ~skCA, sdr, apv, rcP,
            ~dsrid, epoch)
  , DSPReq(~dsrid, epoch, $CA,
           zone('Parent'), $L1, $L2) ]
```

```
rule CA_Preissuance_2:
  let
    sdr_data = <'SDReq', epoch, zone, $C,
                zpkC, $P, $CA, $L1, $L2>
    sdr = <sdr_data, sdr_sig>
    rcP = <'RCert', <tbsP, $L1_P, $L2_P>, rcP_sig>
    dsum_P = <'DSum', zone('Parent'), htbsP,
              <'Delegations', dlgt1, dlgt2>>
    dsp_P = <'DSP', epoch, dsum_P, dsp_sig1, dsp_sig2>
    tbsrc = <'TBSCert', zone, $C, zpkC, $CA>
    prl_data = <'PreLog', sdr, apv, rcP, tbsrc>
    prl = <prl_data, sign(prl_data, ~skCA)>
  in
  [ DSPResp(~dsrid, $L1, $L2, $CA, dsp_P)
  , CA_St_1($CA, ~skCA, sdr, apv, rcP, ~dsrid, epoch)
  , !Pk($L1, pkL1), !Pk($L2, pkL2)
  ]
  --[ Eq(verify(dsp_sig1, <dsum_P, epoch>, pkL1), true)
    , Eq(verify(dsp_sig2, <dsum_P, epoch>, pkL2), true)
    , Eq(htbsP, h(tbsP))
    , NotEq(dlgt1, zone), NotEq(dlgt2, zone)
    , CAPreissued(epoch, $P, $C, zpkC, $CA, $L1, $L2)
    ]->
  [ CA_St_2($CA, ~skCA, sdr, tbsrc)
  , Out(<$CA, $L1, $L2, prl>) ]
```

Figure 16: Two rules from our model describing the CA's actions in Step 7, Figure 6

A fact `F(t1, t2, ...)` involves symbolic terms `t1, t2, ...` that contain variables, constants, functions, network messages, etc. The execution of a rule consumes facts in the LTS's current state that match the rule's premise, and produces new facts that are added to the state. Persistent facts of the form `!F(t1, t2, ...)` are never removed from the state, once added. A public variable `$t` (often used to identify an actor) or a constant `'t'` is always known to the adversary. A fresh variable `~t` is typically used to model random numbers such as keys. We use several Tamarin's built-in functions, including pair (`<t1, t2>`), hashing (`h(t)`), and signing (`sign(t1, t2)` and `verify(t1, t2, t3)`). We also defined our own functions including `zone(t)`, `name(t)`, and `epoch(t)`. Although we use them to simply record constants in our current model, it is possible to introduce equational theories for them to capture a hierarchical naming structure and unlimited epoch transition.

The rule `CA_Preissuance_1` models `$CA` receiving a secure delegation request from `$C` over the insecure network using Tamarin's built-in `In()` fact. Facts `!CA_St_0()`, `CA_St_1()` record `$CA`'s local state. `!ZPk_P()` models the access to the parent zone's public key. The event facts `Eq()` and `NotEq()` specify equality and inequality checks using Tamarin's *restriction* mechanism. We apply them to model the bulk of an actor's local processing of a message, including signature verification and consistency checking. According to the protocol specification (Figure 6), the CA needs to retrieve the parent zone's DSP from the designated loggers over an out-of-band secure channel. The facts `DSPReq()` and `DSPResp()` model such a channel. At the end of the rule `CA_Preissuance_1`, the CA makes a retrieval request with a random id generated using the built-in fact `Fr()`.

In the rule `CA_Preissuance_2`, the CA continues to verify the received DSP and send out a pre-logging message over the insecure network using the built-in `Out()` fact. Event facts such as `CAPreissued()` there facilitate the definition of properties in a model-independent way.

One of the most important event facts we consider is `ZoneDelegated()`, which signifies the successful establishment of a child zone. It should not be placed at the last step of the secure delegation protocol, but where the zone owner has verified the updated DT log (within the distribution window of an epoch). Our model precisely captures this consideration in the rule `Child_Accept_T0` shown in Figure 17.

A zone delegated in an epoch can publish its data in the subsequent epochs. To model this, we introduce a linear fact `ZonePublishable()` that allows a zone to publish at most once in an epoch. The rule `Child_Accept_T0` states that a zone delegated in `T0` can publish once in `T1` and once in `T2`. The parent zone is initialized in `T0` and so it can publish in all three epochs.

The reason why we model three epochs instead of two is to cover the scenario where an attacker attempts to acquire an RCert in `T1` for a zone delegated in `T0`. Such an attack sce-

```
rule Child_Accept_T0:
    let // The following are macros used to improve the specification's readability
      sdr_data = <'SDReq', epoch('T0'), zone, $C, zpkC, $P, $CA, $L1, $L2>
      sdr = <sdr_data, sdr_sig>
      tbsrc = <'TBSCert', zone, $C, zpkC, $CA>
      rcert_data = <tbsrc, $L1, $L2>
      rcert = <'RCert', rcert_data, rcert_sig>
      lcfm_data = <'LogCfm', $L1, hnds>
      lcfm = <lcfm_data, lcfm_sig>
      nds = <epoch('T0'), $CA, $L1, $L2, zone, h(tbsrc)>
    in
    [ C_St_2($C, ~zskC, sdr)
    , In(<$CA, $C, rcert, att1, att2, lcfm>)
    , !Pk($CA, pkCA), !Pk($L1, pkL1), !Pk($L2, pkL2)
    , DTMonitor(epoch('T0'), 'Setup', logged_htbs) // Monitor the updated DT log
    ]
  --[ Eq(verify(rcert_sig, rcert_data, pkCA), true) // The cert is issued by the designated CA
    , Eq(verify(att1, <'LogAttest', h($L1, nds)>, pkL1), true) // and attested by the loggers
    , Eq(verify(att2, <'LogAttest', h($L2, nds)>, pkL2), true)
    , Eq(verify(lcfm_sig, lcfm_data, pkL1), true) // The logging operation is confirmed
    , Eq(h(nds), hnds)                            // and matches the previous logging request
    , Eq(h(tbsrc), logged_htbs) // The monitored log entry is correct
    , ZoneDelegated(epoch('T0'), zone, $P, $C, ~zskC, $CA, $L1, $L2) // Successful delegation event
    ]->
    [ ZonePublishable(epoch('T1'), zone, $C, ~zskC, rcert)
    , ZonePublishable(epoch('T2'), zone, $C, ~zskC, rcert) ]
```

Figure 17: A rule modeling the child zone owner's acceptance of an RCert in epoch `T0`.

nario is different from acquiring an RCert for a non-existent child zone of an existing zone. The former case is captured by Theorem 2 and the latter case by Theorem 1.

As mentioned, our model uses constants to encode zones and names. There must be a way to specify the relations between them. We employ a few hard-coded rules and restrictions to model and enforce a hierarchical name structure.

```
rule Zone_Record_Generator_PX:
  [ GenRecord(zone('Parent')) ] --[]->
  [ Record(zone('Parent'), name('NameX')) ]

rule Zone_Record_Generator_CX:
  [ GenRecord(zone('ChildX')) ] --[]->
  [ Record(zone('ChildX'), name('NameX')) ]

restriction Naming_Structure:
  "All z n #i.
    NameInZone(z, n)@i ==>
    (z = zone('Parent') & n = name('NameX')) |
    (z = zone('ChildX') & n = name('NameX')) "
```

These two rules state that the name `'NameX'` is under both zone `'Parent'` and zone `'ChildX'`, and both of them can publish records for the name. This allows the model to capture attack scenarios where a malicious parent zone serves bogus records for an existing child zone.

B.2 Property Specification

In Tamarin, the execution of a protocol generates a *trace*—a sequence of event facts, associated with timepoints, from rules triggered during the execution. A trace property is a set of traces defined using guarded first-order logic formulae over event facts and timepoints (denoted as terms of the form `#t`). We specify the security theorems introduced in Section 6 as trace property (defined using keyword `lemma`) shown in Figure 18. The formal specification is self-explanatory with the event facts serving as predicates that encode the informally presented theorems. We discuss a few technicalities.

Using the `Compromised()` fact, we can flexibly configure the adversary's capabilities. The adversary by default has the $\mathcal{A}_1$ capability and can compromise any actor except an entity requesting the delegation for a child zone. Not allowing the compromise of the parent zone owner and at least one of the designated loggers leads to an $\mathcal{A}_1$+$\mathcal{A}_2$+$\mathcal{A}_3$ attacker. Imposing only the latter constraint gives the adversary the $\mathcal{A}_1$+$\mathcal{A}_2$+$\mathcal{A}_3$+$\mathcal{A}_4$ capabilities.

In the lemma `E2E_Authenticity`, we do not specify the order of the event `ZoneDelegated` and `UserAccept`, as the order is implied by the epochs they occur, i.e., the latter hap-

```
lemma Delegation_Security:
  "All epoch zone P C zskC
       CA L1 L2 #i1 #i2.
    ( RCertRequested(epoch, zone, P, C,
                     zskC, CA, L1, L2)@i1
    & ZoneDelegated(epoch, zone, P, C,
                    zskC, CA, L1, L2)@i2
    & not (Ex #j. Compromised(P)@j & j<i2)
    & ( not (Ex #j. Compromised(L1)@j & j<i2)
      | not (Ex #j. Compromised(L2)@j & j<i2)
      )
    ==>
    ( Ex #k. SDApproved(epoch, zone, P,
                C, pk(zskC), CA, L1, L2)@k
      & k < i2 )"
```

```
lemma E2E_Authenticity:
  "All P czone C_0 epoch zone U qid qname #i1 #i2
         zskC_0 CA_0 L1_0 L2_0 zskC CA L1 L2.
      ( ZoneDelegated(epoch('T0'), czone, P, C_0,
                      zskC_0, CA_0, L1_0, L2_0)@i1
      & UserAccept(epoch, zone, U, qid, qname,
                   pk(zskC), CA, L1, L2)@i2
      & (epoch = epoch('T1') | epoch = epoch('T2'))
      & not (Ex #j. UpdateLogged(epoch('T1'),
                                 czone)@j)
      & ( not (Ex #j. Compromised(L1)@j & i1<j) |
          not (Ex #j. Compromised(L2)@j & i1<j) ) )
      ==>
      ( zone = czone & zskC_0 = zskC
      & CA_0 = CA & L1_0 = L1 & L2_0 = L2 )"
```

Figure 18: The specification of Theorem 1 (left) and Theorem 2 (right) we proved for RHINE.

pens only in T1 and T2. For the adversary, we do not limit its capabilities in epoch 0, which ends at i1 when the concerned zone czone is delegated. This does not affect security analysis, because our model allows only one zone to be delegated per epoch and disallows the adversary to obtain a child zone owner's private key.

To capture the update protocol's security, we also formalize the following property. It states that once a zone is delegated, its RCert (in particular, the certified key zskC_1 and trusted entities C1_1, L1_1, L2_1) can be updated only by the zone owner generating a signed request using the genuine key (zskC = zskC_0), even if an $\mathcal{A}_1$+$\mathcal{A}_2$+$\mathcal{A}_3$+$\mathcal{A}_4$ is present after the initial delegation setup.

```
lemma Update_Security:
  "All P C_0 zskC_0 CA_0 L1_0 L2_0 #i1
        C_1 zskC_1 CA_1 L1_1 L2_1 #i2
       zone zskC.
    ( ZoneDelegated(epoch('T0'), zone, P,
            C_0, zskC_0, CA_0, L1_0, L2_0)@i1
    & ZoneUpdated(epoch('T1'), zone,
        C_1, zskC, zskC_1, CA_1, L1_1, L2_1)@i2
   & (not (Ex #j. Compromised(L1_1)@j & i1<j) |
      not (Ex #j. Compromised(L2_1)@j & i1<j)))
    ==>
    ( Ex #i3. UpdateRequested(epoch('T1'), zone,
        C_1, zskC, zskC_1, CA_1, L1_1, L2_1)@i3
      & zskC = zskC_0
      & i3 < i2 ) "
```

All these properties are defined over *all* traces. Tamarin also supports proving lemmas that hold when there *exists* a fulfilling trace. This is commonly used for sanity checks of the specification. We have defined multiple such lemmas to test whether our model implements the expected semantics. The following example checks whether the parent zone can legitimately serve records in T0 when no child is delegated.

```
lemma Normal_Resolution_Parent_T0:
  exists-trace
   "Ex P zpk CA L1 L2 U
         qid qname #i1 #i2 #i3.
      ParentInit(zone('Parent'), P, zpk,
                 CA, L1, L2)@i1
    & UserSentQuery(U, qid, qname)@i2
    & UserAccept(epoch('T0'), zone('Parent'),
           U, qid, qname, zpk, CA, L1, L2)@i3
    // no compromise of any actor
    & not (Ex A #k. Compromised(A)@k)"
```

C Achieving High Availability

DNS is a frequent target of (distributed) DoS attacks [58]. A massive DNS outage can make a wide swath of online services unavailable, for example the historic Facebook outage in October 2021. By decoupling the authentication and distribution of a naming system's data (see Section 3), RHINE also separates the concerns of data authenticity and service availability, allowing them to be addressed independently.

One promising direction to ensure the naming service's availability, even amid large-scale DDoS attacks, is to protect the distribution infrastructure with SCION [73], a next-generation secure Internet architecture. With an array of orchestrated mechanisms, including high-speed packet (source) authentication, traffic monitoring and filtering, as well as lightweight bandwidth reservation, SCION can defend against all types of network-level DoS attacks that target network links and nodes and end hosts, offering guaranteed control-plane operation and data delivery. SCION has seen real-world deployments with proven scalability and performance [59].

We plan to deploy RHINE in SCIONLab [60], a full-fledged global Internet testbed, and thoroughly evaluate its practicality, usability, and availability against DDoS attacks.

Enabling Users to Control their Internet

Ammar Tahir, Radhika Mittal
University of Illinois at Urbana-Champaign

Abstract

Access link from the ISP tends to be the bottleneck for many users. However, users today have no control over how the access bandwidth (which is under the ISP's control) is divided across their incoming flows. In this paper, we present a system, CRAB, that runs at the receiver's devices – home routers and endpoints – and enforces user-specified weights across the incoming flows, without any explicit support from the ISP or the senders. It involves a novel control loop that continuously estimates available downlink capacity and flow demands by observing the incoming traffic, computes the max-min weighted fair share rates for the flows using these estimates, and throttles the flows to the computed rates. The key challenge that CRAB must tackle is that the demand and capacity estimated by observing the incoming traffic at the receiver (after the bottleneck) is inherently ambiguous – CRAB's control loop is designed to effectively avoid and correct these ambiguities. We implement CRAB on a Linux machine and Linksys WRT3200ACM home router. Our evaluation, involving real-world flows, shows how CRAB can enforce user preferences to achieve 2× lower web page load times and 3× higher video quality than the status quo.

1 Introduction

This paper tackles a common and seemingly simple problem: how can users control how their Internet access link gets shared across their incoming flows? For instance, how can a user ensure that their Youtube video streaming is not impacted when the Dropbox app on their device starts downloading large files at the same time, and the two flows compete at the user's Internet access link?

At a glance, a plausible solution is to exploit the traffic shaping features provided in many home routers (e.g. mechanisms for weighted fair queuing or prioritization) [1, 15, 18]. However, these mechanisms are effective only when the bottleneck is at (and queues build up at) the home router – this happens when the bottleneck is the uplink from the router to the Internet Service Provider (ISP) for outgoing flows or the downlink from the router to the end-devices for the incoming flows [1]. Our work targets a different problem as illustrated in Figure 1, where the bottleneck for the incoming flows is the downlink from the ISP to the home router, and queues build up in the ISP. This is a common scenario [36, 50], with the Internet access bandwidth being governed by contractual agreements between an end-user and their ISP, and the median broadband download speed being less than 35Mbps in more than half the countries worldwide [3].

Existing literature provides us with two options for managing flow shares at the Internet access link, both of which are beyond the receiving user's control. The first option is to directly schedule or shape traffic at the bottleneck (e.g. via prioritization or weighted fairness) [17, 25, 43, 49]. However, the access bottleneck is controlled by the ISP. ISPs are unaware of user preferences and do not deploy any mechanisms that enable end users to configure how their traffic is scheduled and shaped at the access bottleneck. [1] The second option is for the senders to appropriately control the rate at which flows arrive at the bottleneck. For example, low-priority senders can use "scavenger" protocols that yield bandwidth more readily to higher priority flows [34, 38, 41, 48]. This is again outside the receiving user's control – it is up to the sender to use and configure such protocols.

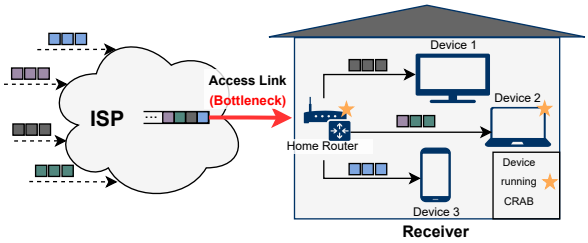


Figure 1: CRAB's Target Scenario. The user may own multiple devices, each downloading multiple flows over the Internet. These flows arrive at the user's home router via the access link from the ISP, from where they get routed to individual devices. The Internet access link is often the bottleneck for the incoming flows [3, 36].

We design a system, CRAB, [2] that enables users (receivers) to control how their Internet access bandwidth gets shared across their incoming flows without any explicit support from external entities (i.e. the ISPs and the senders). Here we use the term receiver to collectively refer to devices in the receiving domain that an end user can directly access and configure – these include end devices (phones and computers) as well as home routers (attached to the access link). More generally, CRAB provides a mechanism to control flow shares exclusively from a vantage point that is topologically placed *after* the bottleneck – where queues don't build up naturally, and where one has seemingly zero control.

CRAB allows a user (i) to configure the home router with weights across each end-device, and (ii) to optionally configure end-devices with weights across their incoming flows

[1] Although a few research proposals of this form exist [19, 24, 27, 35, 54], they have not been realized in practice, given the inherent difficulty of coordinating across multiple domains.

[2] For Customizable Receiver-driven Allocation of Bandwidth.

(defined based on application, web domain, etc). It then strives to throttle the incoming flows at the home router (grouped by destination device) and individually at the end devices (if enabled) to their respective *max-min weighted fair share rates*. CRAB's key challenge lies in correctly computing these rates after the bottleneck (as discussed below). While this after-the-bottleneck throttling cannot *directly* control how the access bandwidth is divided across the incoming flows, it signals the senders (which typically run some form of congestion controller [22, 26, 30, 32, 39, 46, 52]) to lower their sending rates to the throttled values, thus enabling the flows to eventually converge to their desired shares.

So what makes it difficult to compute the weighted fair share rates after the bottleneck? Given flow weights, computing the correct (max-min) weighted fair rates for each flow requires knowing the bottleneck link capacity and the flow demands. Once the absolute weighted share of a flow has been computed from the link capacity and flow weight, the max-min weighted fair rates can be computed by re-allocating any excess capacity, that is unused by flows with demands smaller than their absolute share, to other flows in the proportion of their weights. While the capacity and the flow demands are naturally available at the bottleneck, CRAB (placed after the bottleneck) must estimate them by observing the incoming traffic at the receiver. This introduces multiple challenges:
(1) It is not possible to distinguish whether the total traffic observed at the receiver is limited by the access link capacity or by the flows' cumulative demands – the latter would result in underestimating capacity.
(2) The arrival rate of a flow at the receiver depends on the rate at which it was served at the bottleneck. A flow that got a small share of bandwidth at the bottleneck (less than its weighted share or demand) could be wrongly perceived as having low demand.
(3) If the receiver incorrectly throttles flows to rates lower than their weighted shares (due to spuriously low capacity or demand estimates), the flows' sending rates (and their observed arrival rates at the receiver) would end up matching the throttled rates. As a result, the link capacity and demand estimates would stay unchanged and the system will not self-correct. Similar reasoning makes it difficult to adapt to an increase in link capacity and flow demands.

The centerpiece of CRAB is a control loop that is designed to tackle the above challenges (§3). It continuously loops between (i) measuring flow arrival rates (over timescales of hundreds of milliseconds) to estimate link capacity and flow demands, and (ii) re-computing and enforcing weighted fair-share rates based on these estimates. By waiting for rates measured over long enough timescales before reacting, CRAB avoids fast reaction to spuriously low demand estimates – it allows for the impact of any flow throttling to kick in, and for the sending (and observed) rates for the remaining flows to grow to their true demands (or weighted shares), before reallocating any unused capacity. When re-allocating capacity from a flow with low demand, CRAB leaves some headroom to detect growing demand, at which point it immediately reclaims all of the flow's re-allocated bandwidth, again allowing the flow to grow to its true demand or its weighted share. To self-correct capacity underestimation, it periodically probes for more bandwidth by explicitly increasing the total rate assigned to flows and checking for any consequent increase in observed rates.

CRAB runs the same logic at the home router and at the end-points, without requiring any explicit coordination among them CRAB at the home router enforces per-device shares based on estimated access link capacity and per-device demands. CRAB at the end-point independently adapts its capacity estimate to the per-device rate enforced by the home router and controls per-flow shares. When directly attached to the ISP's link (without a home router) or when the router and end-device are owned by different entities (e.g. in airports, cafes, etc), an end-device with CRAB enabled can self-sufficiently enforce its desired shares across its incoming flows.

CRAB's cautious re-allocation of unused capacity can leave the bottleneck link under-utilized at times. As we illustrate in §2, some amount of link under-utilization is inevitable when shaping traffic *after* the bottleneck (maximally utilizing the link would imply no flow gets throttled at the receiver, and consequently no impact on how the bottleneck bandwidth is shared). The link under-utilization with CRAB typically manifests as transient dips in the throughput for lower priority (throttled) flows, below their max-min weighted shares. This is a reasonable price to pay for better performance for higher priority traffic that is achieved by enforcing user-specified flow shares with CRAB.

We implement CRAB (§4) on a Linux endhost and a Linksys WRT3200ACM router. Our end-host implementation also includes hooks for classifying flows that are broadly defined by users based on applications and web domains, and involve cross-origin requests (e.g. to CDNs and ad networks). Our experiments involving real-world flows with different sender-side congestion controllers (YouTube videos, web browsing, and bulk downloads), show how CRAB comes close to achieving the desired weighted fair share rates. In particular, CRAB achieves 2.5-3× higher video quality and 2× lower web page load times in presence of lower-priority bulk flows than the status-quo (that cannot enforce desired preferences), with 10-20% decrease in overall link utilization.

2 Overview

CRAB enables the user to manage how their Internet access link (that is often the bottleneck for downloads [3, 36]) gets shared across the incoming flows. Figure 1 illustrates a typical target scenario. CRAB at the home router controls how the access bandwidth is shared across traffic destined to each end-device. CRAB at the end-device (if enabled) controls how its router-enforced bandwidth share is divided across its incoming flows. Note that all end devices need not run CRAB

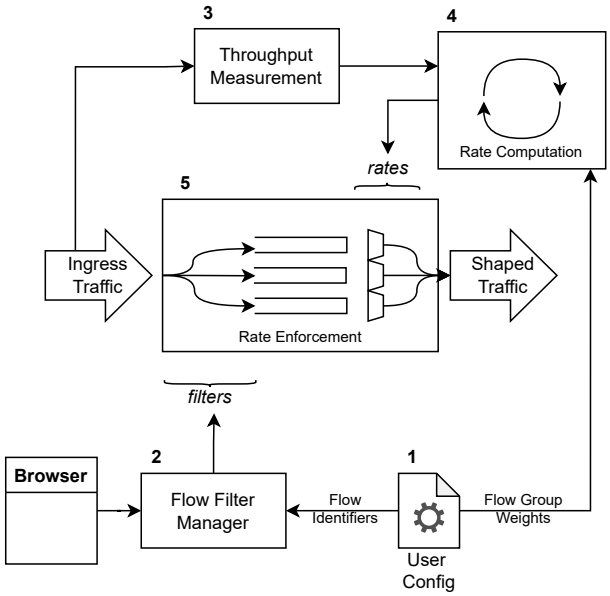


Figure 2: CRAB's high-level workflow (detailed in §2.1).

– only an end device that wishes to control the bandwidth sharing across its own flows needs to enable CRAB.

2.1 CRAB Framework

We provide an overview of the CRAB framework at the end-device (noting the small differences in CRAB's router design towards the end). CRAB sits in front of the ingress interface, where it intercepts and shapes the incoming traffic before forwarding it to the kernel's TCP/IP stack. Figure 2 shows the key elements in CRAB's architecture.

1. User Interface. Our current prototype allows users to define flows based on three criteria: (i) all traffic destined to a specified application running on the end-device, (ii) all traffic associated with a specified web-domain, and (iii) all traffic originating from a specified source address. [3] The user can group multiple flows into a flow-group, and specify a weight for each flow-group. Users can also specify a weight for a default flow group, where traffic not classified in any other flow group is mapped. Henceforth, we use the terms flow and flow-group interchangeably.

2. Flow Filter Manager. It maps high-level flow identifiers (as specified by the user) into TCP/IP header fields that the system can use to classify packets as and when they arrive at the interface. Mapping a web-domain into header fields requires some inputs from the browser (we detail this in §4).

3. Throughput Measurement. CRAB sniffs the incoming traffic to measure (i) throughput of each flow group to estimate demands, and (ii) the cumulative throughput over all flows to estimate link capacity. The link capacity estimated by CRAB at the end-device corresponds to the device's bandwidth share as enforced by the home router.

4. Rate Computation. CRAB computes weighted fair share rates for each flow, based on user-specified weights and the capacity and demand estimates obtained from flow throughput measurements (as detailed in §3).

5. Rate Enforcement. CRAB uses the mappings from the flow filter manager to classify incoming traffic into user-defined flow groups. It puts the traffic for each group into separate queues, and throttles each queue to its computed weighted fair share rate. Since CRAB throttles traffic *after* the bottleneck, it cannot *directly* control how flows are scheduled at the bottleneck. However, it induces packet losses and queuing delay at the receiver, which signals the senders to adjust their rates to the throttled value, thus eventually achieving the desired bandwidth shares at the bottleneck.

We considered a few other alternatives for signalling sending rates from the receiver, e.g. by adjusting TCP receive windows. We decided to use throttling for rate enforcement because of its generality – all senders that run some form of congestion controller (either over TCP [22, 26, 30, 32, 52] or UDP [39, 46]) would naturally react to queue buildups and/or packet drops induced by throttling.

CRAB framework at the home router is same as that at the end-device. The only difference is that since users configure the home router with weights across each end-device (directly identified by the destination IP address), an explicit flow filter manager is not required.

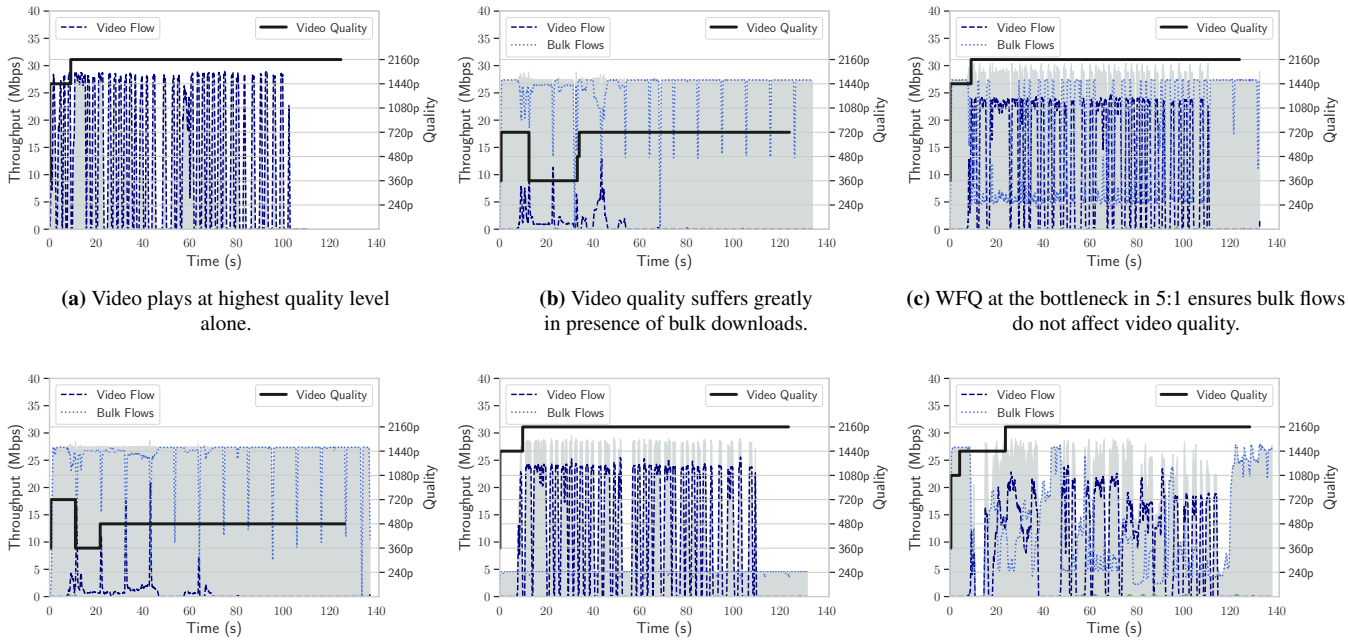
Note that, in principle, one could have managed flow-group shares directly at the home router, instead of the end-device. But then classifying the incoming traffic based on flow-groups at the router would have required explicit coordination with the applications running at the end-device in real-time (as explained in §4), which would have complicated system deployment. Our current division of functionality between the home-router and end-points requires no explicit coordination among them, which greatly simplifies CRAB's deployment and use, and extends CRAB's utility to other contexts beyond home users (as discussed in §7). It also reduces computational complexity at the router, with the router managing only per-device queues, and each device then managing the rates for their own flows.

2.2 Goals and Challenges

We use a series of experiments to illustrate some of the challenges that CRAB must tackle. We consider a scenario where a Linux end-host is directly attached to the access link from the ISP (without a home router). For repeatable experiments, we emulate an ISP-controlled access link by routing all traffic for the end-host via a router that mimics the ISP and throttles the traffic to 30Mbps. When we only stream a 4K YouTube video on the end-host, we find that the video quality stays at the maximum level (Figure 3a). We then stream the same video in presence of two long-lasting bulk downloads [4] over

[3] Future extensions of our system can support more criteria.

[4] Bulk downloads of games and movies are fairly common among users with limited bandwidth because of inaccessibility of high quality video streaming or cloud gaming [29, 31].



(a) Video plays at highest quality level alone.

(b) Video quality suffers greatly in presence of bulk downloads.

(c) WFQ at the bottleneck in 5:1 ensures bulk flows do not affect video quality.

(d) Instant reaction fully utilizes the link, but cannot enforce desired weights.

(e) Indefinitely throttling bulk flows results severely under-utilizes the link.

(f) CRAB ensures high video quality at cost of slight link underutilization

Figure 3: Video quality suffers in presence of bulk download flows, we look at different possible ways to ensure this does not happen. In all experiments, link bandwidth is set to 30 Mbps.

the Internet, and evaluate the results under different settings.

Status Quo. Today, there is no way for a user to enforce how their access bandwidth gets divided across their flows. We find that with the bulk downloads consuming a large share of the access bandwidth, the status-quo achieves very low video quality (Figure 3b).

This degradation in video quality is clearly undesirable and can be mitigated if the user can specify and enforce a higher weight (say 5×) for the video flow.

Impractical Ideal: WFQ at the bottleneck. We next model the ideal, but impractical scenario where the ISP enforces user preferences at the bottleneck via WFQ. For this, we configure the router in our setup (that mimics the ISP) to use weighted deficit round-robin (DRR) [49], with the video flow to bulk flows ratio set to 5:1. As shown in Figure 3c, the video flow is able to use its absolute share of 25Mbps, and achieve the highest video quality, while bulk downloads get at least 5Mbps. WFQ is *work-conserving* – whenever the video flow consumes less than its share of 25Mbps (e.g. when its playback buffer is full), the remaining capacity is used by the bulk downloads, keeping the link maximally utilized.

Impossible to mimic WFQ after the bottleneck. CRAB strives to mimic the ideal WFQ-enforced rates. However, while WFQ at the bottleneck can achieve both desired bandwidth shares and maximal link utilization, there is an inherent trade-off between the two when shaping traffic *after* the bottleneck. We highlight this trade-off by illustrating two extreme strawmen for rate computation at the receiver.

(i) Work-conserving re-allocation cannot enforce weighted fairness after the bottleneck. In order to maximally utilize the link, whenever the video flow's demand becomes less than its absolute share, any unused capacity should be explicitly reallocated to the bulk flows. It is natural to use the arrival rate of the video flow at the receiver as an estimate of its demand. However, if the video flow gets a small share of bandwidth at the bottleneck due to competing flows, it will have a low arrival rate at the receiver and will be incorrectly perceived as having low demand. We now evaluate the effects of instantaneously re-allocating unused capacity based on such spurious demand estimates.

For this, we configure the end-device to use Linux HTB (Hierarchical Token Bucket) [5] to throttle the incoming flows to their absolute shares (25Mbps for video and 5Mbps for bulk), and enable HTB's "bandwidth borrowing" feature which immediately re-allocates any capacity that is unused by a flow with a smaller arrival rate. Since the total arrival rate at the receiver is already capped by bottleneck link capacity, such instantaneous re-allocation induces no throttling – while this achieves maximal link utilization, it cannot enforce desired bandwidth shares and produces the same outcome as the status-quo (Figure 3d). For the setup in Figure 1, enabling WFQ for the incoming flows at the home router (after the bottleneck), will produce the same effect.

More generally, maintaining maximal link utilization is fundamentally at odds with CRAB's mechanism of enforcing desired rates. In order to signal any change in the sending

rates, CRAB must throttle some flows at the receiver – the link under-utilization thus induced is what gives room to the remaining flows to grow to their true demands or absolute weighted shares.

(ii) No re-allocation leads to severe under-utilization. At the other extreme, indefinitely throttling bulk flows to their absolute weighted fair rates, oblivious of video flow demand estimates, allows the video flow to grow to its true demand and ensure high video quality, but decreases the link utilization by 46% compared to the status quo (Figure 3e).

CRAB achieves desired shares with high link utilization. CRAB navigates the above trade-off by re-allocating unused capacity more cautiously (at timescales of a few hundred milliseconds) – this allows the impact of any throttling to kick in, and for the flows to grow to their true demands or absolute shares, before the unused capacity is re-allocated. For flows consuming less than their absolute shares, CRAB provisions for detecting a growth in demand, upon which it immediately reclaims all lent out capacity. This cautious reallocation and aggressive reclamation may under-utilize the link at times, which is inevitable (as discussed above) and primarily affects the bulk flows. On the whole, as shown in Figure 3f, we find that CRAB can effectively enforce the desired shares while lowering the link utilization by only 19% with respect to the status quo. CRAB has 2× higher link utilization than the extreme alternative of no re-allocation or of the user explicitly pausing the bulk flow while the video lasts.

Other Challenges. CRAB was able to correctly estimate link capacity in the above experiments. However, link capacity estimation can be more challenging in other scenarios. In particular, if the total incoming traffic at the receiver is limited by flows' cumulative demand (as opposed to the access link capacity), it would result in underestimation of link capacity – CRAB should be able to self-correct its capacity estimate to ensure correct bandwidth shares if flows' demands increase. Link capacity could also vary over time – CRAB should be able to detect any changes and accordingly recompute weighted share rates. What makes such self-correction and adaptation particularly difficult is that once flows have been throttled to a spuriously low rate, their sending rates (and, consequently, their arrival rates and CRAB's capacity estimates) could stay stuck at the throttled values. CRAB handles this by explicitly probing for more bandwidth.

We detail CRAB's control loop, i.e. its re-allocation, reclamation, and bandwidth estimation logic in §3.

The specific control loop we describe in this paper is one way of using CRAB's framework for controlling flow shares at the receiver. There can be alternative ways of using our framework while complying with our observation that some amount of link under-utilization is inherently needed to control flow shares from the receiver. For instance, we can directly throttle the *cumulative* rate of all incoming traffic at the receiver to a value lower than the overall link capacity that CRAB estimates (via its throughput observation and bandwidth probing logic). This creates an artificial bottleneck at the receiver where we can enforce the desired scheduling policy (prioritization, weighted fair queuing, etc) across the flow classes maintained by CRAB. While effective at controlling flow shares, such an approach would be more sensitive to precise bandwidth estimation and may lead to unnecessary wastage of bandwidth due to consistent link under-utilization. We evaluate this alternative in §5.4.

2.3 CRAB's Scope

CRAB's scope is limited in the following key ways:

1. CRAB relies on the fact that flows have a sender-side rate control mechanism that is responsive to throttling. This holds for most of the traffic on the Internet that either uses TCP's congestion control mechanism or runs adaptive rate control over UDP [39]. CRAB is not effective in scenarios where a flow is not responsive to throttling.

2. CRAB cannot directly control the fine-grained queuing behavior at the bottleneck. It can only influence the rates achieved by different flows over long-enough timescales (a few hundred milliseconds), as it requires the senders to react to the throttled rates or the increased room for growth in rates. This allows CRAB to effectively control bandwidth shares across long-lived flow groups, e.g. video streaming, conferencing, web browsing sessions, bulk downloads, etc. However, CRAB cannot effectively control the queuing delay experienced at the bottleneck by short intermittent downloads that terminate before CRAB gets a chance to react (e.g. a flow group comprising of only interactive chats). Though CRAB cannot actively help such flows, it will not hurt them either.

3. CRAB can only actively control how a user's incoming flows share their common bottleneck (at the ISP's access link or at the access router). If a flow is bottlenecked elsewhere (e.g. at the sender's uplink), it is simply perceived by CRAB as having a lower demand at the shared access bottleneck, and CRAB accordingly reallocates the access link capacity unused by this flow across the remaining flows.

4. Since CRAB must react at slow timescales (of hundreds of ms) to build correct capacity and demand estimates, it is not a good fit for volatile cellular networks where link capacity changes by large magnitudes at much smaller timescales due to high mobility, hand-overs, etc [23, 37, 51, 57]. In comparison, we found broadband connectivity and home WiFi networks to be significantly more stable (see Appendix B).

3 CRAB Control Loop

We now describe CRAB's control loop that continuously iterates between (i) measuring flow throughput to estimate capacity and demand, and (ii) computing and applying new per-flow rates. Table 1 lists different attributes that CRAB maintains for each flow and uses for rate computations.

3.1 Throughput Measurement

Two parameters govern the granularity of our throughput measurement: the observation period (t) and the number of

Flow Attribute	Description
weight	The weight assigned by the user
observed_tpt	The measured throughput of the flow (its arrival rate at the ingress)
true_bw	The absolute weighted fair share of the flow computed from estimated link capacity and flow weight
lent_bw	The unused bandwidth the flow lends out to other flows
borrowed_bw	The amount of bandwidth the flow borrowed from other flows
assigned_bw	The rate assigned to a flow (set to $true_bw + borrowed_bw$)
saturating (bool)	set to true if the flow's demand is potentially higher than its assigned bandwidth
non_saturating (bool)	set to true if the flow's demand is smaller than its assigned bandwidth
growing (bool)	set to true if the flow needs to reclaim its lent bandwidth

Table 1: Attributes of a flow in CRAB

observations (n). In each observation, CRAB measures the throughput (or arrival rate) for each flow over time t (i.e. number of bytes received in t time divided by t). It makes n such observations and picks the maximum value as the observed throughput of a flow. We use the max filter instead of mean or median to capture bursts which are common for applications like web browsing and video streaming.

The values of t and n govern how long we wait before making any changes to flow rates. A very small value of t would result in inaccurate throughput measurements, whereas a very high value can mask spikes in demand. [5] A very small n will not give flows enough time to adjust to new rates skewing demand estimates, and a very large n would induce much slower reactions to changes in demands and capacity. In practice, $n \times t$ should be as high as a few RTTs to allow the senders enough time to react. We find that setting t to 200ms and n to 5 works well across different scenarios. We evaluate the impact of these parameter settings in §5.6.

After every throughput measurement (over $n \times t$ s), CRAB sets following flags of each flow f:

Growing: A flow is determined to be growing if f had previously lent out bandwidth but its observed throughput indicates an increase in its demand (i.e. it is using more than what it was using earlier). $f.growing = (f.lent_bw > 0)$ **and** $(f.observed_tpt \geq f.assigned_bw - f.lent_bw)$.

Saturating: If the flow f is either growing or it is utilizing almost all of its assigned bandwidth, i.e., $f.saturating = f.growing$ **or** $(f.observed_tpt + \delta \geq f.assigned_bw)$. Here, δ masks noise in throughput observations, and is set to $max(0.1 \times f.observed_tpt, 0.25Mbps)$. Note that we consider all growing flows to be saturating, but a saturating flow (that is simply utilizing all of its assigned bandwidth) may not necessarily be growing.

Non-saturating: If f is under-utilizing its assigned bandwidth (after subtracting its lent out bandwidth): $f.non_saturating = (f, observed_tpt + \delta) < (f.assigned_bw - f.lent_bw)$.

[5] The value of t should be at least as high as the inter-arrival time between multiple consecutive 64KB chunks to correctly compute throughput (with TSO/LRO enabled, packets arrive in bursts of 64KB).

3.2 Rate Computation Overview

Figure 4 shows a simplified state diagram for CRAB's control loop. Followed by a throughput measurement, we take one of the four actions in the exact priority order.
(1) If there exists any growing flow, we do reclamation for it (i.e. reclaim any bandwidth it has lent out to other flows).
(2) Otherwise, if there is at least one non-saturating and one saturating flow, we reallocate (or lend out) bandwidth unused by non-saturating flows to saturating flows.
(3) If observed throughput has dropped and there does not exist any saturating flow, we decrease the bandwidth estimate and divide it between flows according to their weights.
(4) In all other cases, we try to probe for more bandwidth.
We return to the throughput measurement after each action. The following sub-sections describe these actions and their triggers in more detail.

3.3 Reallocation

Reallocation takes place if there is at least one non-saturating flow (which can lend out bandwidth) and at least one saturating flow (which can potentially utilize this lent bandwidth).

CRAB first computes the bandwidth each flow f can lend:

$$f.lent_bw = f.assigned_bw - f.observed_tpt - headroom$$

We keep a small headroom (set to 0.25Mbps) to enable detecting when the flow needs to grow back (§5.6 evaluates the impact of this parameter).

If $f.lent_bw > f.borrowed_bw$, this means that the flow can no longer make use of the bandwidth it has previously borrowed and instead has extra unused bandwidth to lend. In this case, CRAB subtracts $f.borrowed_bw$ from $f.lent_bw$ and resets $f.borrowed_bw$ to zero. CRAB computes the global excess (unused) bandwidth by summing up the lent bandwidth across all such flows. It then resets the assigned bandwidth for each flow f to $f.true_bw$, after which it reallocates the excess bandwidth across all flows in proportion to their weights until their demands are satisfied, accordingly updating $f.borrowed_bw$ and $f.assigned_bw$ (set to $f.true_bw + f.borrowed_bw$) for each flow. The algorithm for this redivision is given in AppendixA.

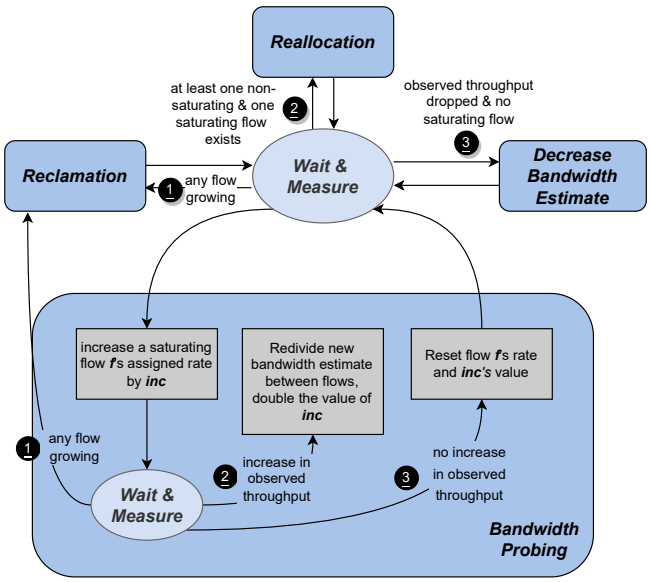


Figure 4: State diagram of CRAB's control loop.

3.4 Reclamation

Notice that we do not decrease the assigned bandwidth of the non-saturating flow during reallocation. This, combined with the lent bandwidth headroom, ensures that CRAB can detect growth in a flow's demand, and classify such a flow as *growing* (as noted in §3.1). To enable faster reclamation, CRAB terminates the throughput measurement sooner than $n \times t$ s, once it detects that any flow f is growing. It reclaims all bandwidth lent out by the flow f by setting $f.lent_bw = 0$. It reduces the global excess if $f.lent_bw$ (before updating to 0) was greater than $f.borrowed_bw$, and accordingly recomputes how the updated excess bandwidth is redivided across flows (using the same logic from §3.3). Note that when redividing excess bandwidth, we consider a growing flow to be a saturating flow because it can potentially utilize more bandwidth as its demand is still unknown.

3.5 Bandwidth Estimation

CRAB keeps track of estimated bandwidth (*estimated_bw*) based on the overall observed throughput across all flows ($total_observed_tpt$). We now discuss CRAB's mechanism for detecting a change (increase or decrease) in capacity. This is required for (i) correcting spuriously low capacity estimates caused by limited demand, and (ii) adapting to potential capacity variations (e.g. due to change in an end-device's share of bandwidth triggered by changes in another device's demands).

Decrease in Bandwidth. A drop in $total_observed_tpt$ can happen due to two reasons – either the total bandwidth has dropped, or a flow's demand has decreased. Bandwidth estimate should not be decreased in the latter case – reallocating the bandwidth now unused by the flow can increase observed throughput. Since there is no way to tell these two scenarios apart, we first let reallocation try to fix things before we reduce *estimated_bw*. More specifically, as long as there is a saturating flow (that can potentially use more bandwidth), CRAB keeps trying to reallocate excess capacity. If no flow can be classified as saturating and $total_observed_tpt$ remains lower than *estimated_bw*, [6] it can assume that the bandwidth has dropped and reduces *estimated_bw* to the $total_observed_tpt$. This assumption can still be incorrect because it is possible that it is not the bandwidth, but the demand for all the flows that have actually decreased. However, in such a case, decreasing *estimated_bw* does not hurt the flows, and later when flows grow back, we can re-estimate bandwidth through bandwidth probing as discussed ahead.

After updating *estimated_bw*, CRAB resets global excess bandwidth to zero and assigns each flow its absolute weighted share of the new bandwidth estimate. This eradicates the effect of erroneous reallocations that happen before decreasing the bandwidth estimate. Correct reallocation (if needed) can then take place in subsequent iterations of the control loop.

Increase in Bandwidth. Detecting an increase in bandwidth is particularly tricky because CRAB itself limits the arrival rates of flows by throttling them. Thus, it needs to explicitly probe for more bandwidth. To do so, CRAB increases the assigned rate of a saturating flow and then waits to take a throughput measurement. If it detects any significant increase in $total_observed_tpt$, [7] it updates *estimated_bw* to $total_observed_tpt$. It accordingly computes the absolute weighted share for each flow (setting it as the flow's true and assigned bandwidth). It then accordingly increases the global excess bandwidth and redivides the bandwidth across all flows in proportion to their weights until their demands are satisfied (as in §3.3).

The above requires careful consideration of two aspects – which saturating flow should we select for bandwidth probing and what should the increment value be. We select a new saturating flow in a round-robin fashion in each bandwidth probing round, such that any one flow does not get an advantage over the other. We calculate the increment in proportion of *estimated_bw* i.e. $increment = inc \times estimated_bw$. To prevent large disruptions in flow shares, we start off with a small value of *inc* (0.125), but every time bandwidth probing results in an increase in estimated bandwidth, we double the value of *inc*. When probing does not result in an increase, we reset the increment to its starting value. We continuously keep probing for bandwidth until this happens. The only time we terminate bandwidth probing prematurely is if we detect a flow to be growing, in which case we skip to reclamation.

Bootstrapping. CRAB bootstraps by calculating weighted fair share rates of flows based on an arbitrary initial estimate of bandwidth. By keeping this estimate large, we can avoid doing

[6] To be robust against minor throughput changes, the precise condition we check for is $total_observed_tpt < 0.9 \times estimated_bw$.

[7] If the increase in $total_observed_tpt$ is greater than $\max(0.1 \times total_observed_tpt, 1\text{Mbps})$

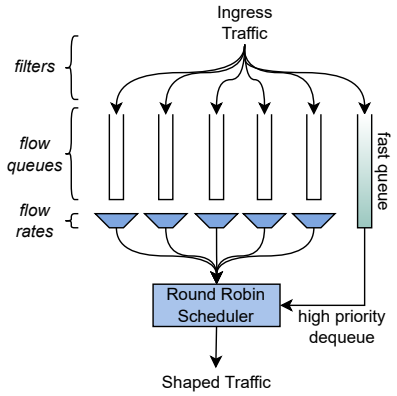


Figure 5: Linux's HTB scheduler.

bandwidth probing at startup time. Once the first measurement interval is over, CRAB is able to fix this estimate. CRAB can maintain a historical average estimate of bandwidth in the persistent state to feed as an initial value for more efficient bootstrapping.

3.6 CRAB's Router Control Loop

CRAB runs the same control loop at the home router, except for one change: each throughput measurement takes $3n$ observations, so the length of throughput measurement is $3n \times t$ s. This ensures that CRAB in end-devices is able to adjust their flow rates to per-device rate changes made by the home router, before the router's next measurement.

4 System Implementation

We implement CRAB's end-host logic on a 2.4GHz 8-core Ubuntu 20.04 machine with Linux 5.11 kernel, and its home-router logic on a 1.8GHz dual-core Linksys WRT3200ACM router running Linux-based OpenWRT firmware. We start with discussing the end-host implementation (§4.1-§4.5), and then discuss router implementation (§4.6).

4.1 CRAB's Placement

The inbound traffic arrives at an ethernet (eth) or wireless (wlan) interface. Since Linux does not have rich options to shape ingress traffic, we redirect the inbound traffic to an intermediate function block (ifb) [6] interface, where CRAB can shape traffic using Linux TC [10] (§4.3). The shaped traffic then gets picked up by the receiver's TCP/IP stack for further (normal) processing.

4.2 Throughput Measurement

We measure the flow throughputs (or arrival rates) by using scapy [16] to sniff and record the incoming traffic at the original ingress interface (eth or wlan).

4.3 Rate Enforcement

We enforce the computed weighted fair share rates for each flow group at the ifb interface using Linux's HTB (Hierarchical Token Bucket) scheduler [5].

```
Flow Groups : {
    Video Streaming: {
        flow identifiers: ["app>netflix", "web>youtube.com", "web>hulu.com"],
        weight: 5
    },
    Work: {
        flow identifiers: ["app>dropbox", "ip>1.2.3.4"]
        weight: 5
    },
    Default:{
        weight: 1
    }
}
```

Figure 6: An Example of a CRAB Config File.

Primer on HTB. Figure 5 shows the basic working of HTB. HTB allows a user to classify traffic into different classes (based on filters defined by packet header fields such as IP address, protocol type, TCP/UDP ports, etc) and specify different rates for throttling each class. If an incoming packet cannot be classified into a class defined by the filter rules, it is put into a special queue called the *fast queue*. Each class consists of a FIFO queue (to buffer packets) and a token bucket filter. Tokens are added to the bucket at the specified rate. If the amount of tokens in the bucket is greater than or equal to the size of the head packet in the queue, then the packet is dequeued. Otherwise, the queue blocks. If the queue is full, new incoming packets for that class are dropped.

A round-robin scheduler moves between classes to dequeue packets. If a class cannot dequeue a packet because it does not have enough tokens, the scheduler moves to the next class without blocking. The scheduler prioritizes dequeuing from the fast queue before any of the HTB classes.

HTB supports work-conserving traffic shaping by allowing unused tokens to be borrowed by other classes – we do not enable this feature in CRAB for reasons discussed in §2.2, and use the re-allocation logic described in §3 instead.

HTB in CRAB. We maintain a class for each flow group. The flow filter manager (detailed in §4.5) installs the filter rules for classifying incoming packets into their respective classes. We set the throttling rate for each class to the weighted fair share rate of its respective flow-group (as computed by CRAB's control loop), and accordingly adjust the queue size. [8]

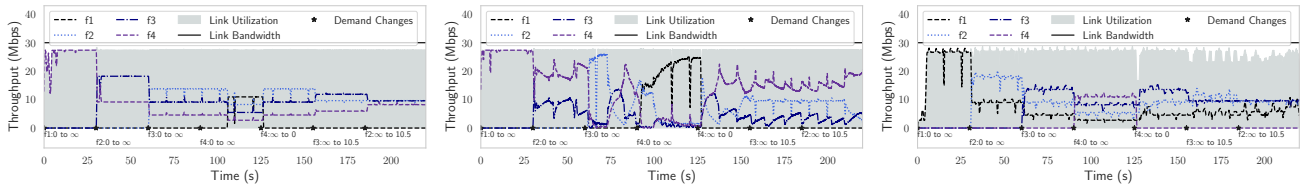
4.4 User Interface

The user specifies their preferences in a config file. Figure 6 shows a sample config file. It contains different flow groups, where each flow group consists of a list of flow identifiers and a weight associated with the flow group. Our current implementation allows a user to give three kinds of flow identifiers – *ip* (for traffic coming from the specified IP address), *app* (for traffic destined to the specified application), and *web* (for traffic destined for webpages from the specified web domain).

4.5 Flow Filter Management

Flow filter manager (FFM) maps user-specified high-level flow identifiers into packet header fields that can be used for

[8] Queue size is adjusted to 2BDP = 2 × rate × RTT, where we assume RTT to be 50ms.



(a) WFQ@bottleneck **(b)** Status Quo **(c)** CRAB@end-host

Figure 7: Weighted sharing of 30 Mbps bottleneck between 4 flows in a ratio of 4:3:2:1, where f1 has a weight of 1 and f4 has a weight of 4. The bottom part of each graph shows flow demand changes in Mbps, where ∞ means unknown demand.

filtering packets at HTB. When we encounter a packet from an unknown *source* IP address (that does not correspond to an installed filter rule), we copy the packet header to FFM. FFM determines the mapping for the packet (as detailed below) and installs a new filter rule for it. Note that FFM installs filter rules asynchronously, and we do not block traffic while this happens. Instead, the unclassified packet is put (and served) in HTB's fast queue. Once FFM installs the corresponding rule, it is used to correctly classify future packets from that flow. This ensures that if the unclassified packets are of a potentially important flow, their service is not degraded. Typically, FFM is able to install new filter rules fast enough, such that only the first couple of packets for an unclassified flow land up in the fast queue. FFM installs filter rules as follows:

(i) Source IP Mapping: If the source IP address of a packet arriving at the ingress matches with an *ip* field in the config file, we install the corresponding filter simply based on that.

(ii) Application: If the source IP is not found in the config file, we use psutil [7] to reverse map the destination port on the packet header to find which application has opened that port. If the application is specified in any flow group in the config file, we install a filter that maps traffic from the packet's source IP address to this flow group. If the application does not match any identifier in the config file, we map it to the default flow group (unless the app is the web browser, which we handle as a special case as discussed below)

(iii) Web Domain: Mapping packets to the web pages (identified by web domains) is tricky for multiple reasons:

(a) Web pages from the same web domain may be hosted on several different machines.

(b) Web pages make many cross-origin requests e.g. to CDNs and ad servers. Since these requests are often dynamic (e.g. due to load balancing in CDNs or real-time bidding in ad networks), it is not possible to pre-populate a list of IP addresses a webpage from a certain web domain would access.

(c) Packet headers do not carry any information about which web domain the packet belongs to. The packet payload of HTTPS traffic, which does carry some information, is encrypted at the ingress (where CRAB sniffs) and is decrypted only at the browser.

Thus, if the application using the destination port is a web browser, FFM needs more context from the browser to correctly classify the packet. We built a Google Chrome plugin that provides this context. As soon as the browser starts receiving an HTTPS response, the plugin prepares and sends a message to FFM that contains the source IP address of the response, and the URL of the webpage that initiated the corresponding HTTPS request. Using this mapping, the FFM can extract the web domain from the URL and find its match in the config file. If there is a match, we install a filter for the source IP address, mapping it to the corresponding flow group. Otherwise, we map it to the default flow group.

FFM may not be able to correctly classify packets if the relevant packet header fields are encrypted (as in the case of VPNs). In such cases, application integration similar to the plugin we built for Google Chrome can help remove FFM's dependency on encrypted packet headers and enable the classification of non-encrypted fields. Note that DNS encryption does not affect FFM, as it does not rely on DNS packets.

4.6 Home Router Implementation

Similar to our end-host implementation, CRAB at the router sniffs incoming traffic (using tcpdump [12]) at the ingress (eth) interface, and redirects the traffic to the ifb interface. It classifies the traffic based on the destination IP address at the ifb interface and enforces the per-destination rates computed by CRAB's control loop using Linux HTB.

5 Evaluation

We now evaluate the following:

- CRAB's ability to adapt to changes in flow demands and link capacity in synthetic scenarios involving real-world bulk flows (§5.1).
- Performance (QoE) improvement enabled by CRAB for real-world video streaming (§5.2) and web browsing (§5.3), when competing with bulk downloads.
- An alternative way of using CRAB's framework to enforce user preferences, and the trade-offs involved (§5.4).
- The need for CRAB's home router logic with multiple active devices in the user's domain (§5.5).
- The impact of changing CRAB's key parameters, i.e. throughput observation length and lending headroom (§5.6).
- CRAB's robustness to diverse traffic characteristics and its overheads (summarized in §5.7, and detailed in the appendix).

We use the same setup as in §2.2, that models a single end-host directly attached to the ISP's link (for repeatable experiments, we emulate a 30Mbps access link by throttling

traffic at our home router). The only exception is §5.5, where we extend the home router logic to implement CRAB after the throttle point.

Unless otherwise specified, we compare CRAB with two baselines: (i) ideal WFQ (implemented at the access link emulated by our router), and (ii) status quo (i.e. no traffic shaping). We also conducted experiments using HTB with bandwidth borrowing *after* the bottleneck – since this produces almost exactly the same outcome as the status quo (as discussed in §2.2), we omit presenting those results.

Our workloads span real-world flows with a diverse set of sender-side rate control mechanisms: (i) bulk download flows that likely use TCP Cubic [30], (ii) YouTube video streaming that uses BBR [11, 22] along with an adaptive bitrate (ABR) algorithm for adapting video quality, and (iii) web page-loads over Google Chrome that potentially use a mix of BBR [22, 33] and Cubic [30] over QUIC [33, 39] and TCP [9].

5.1 CRAB in action

We design synthetic scenarios to visualize CRAB's fine-grained reaction to changes in flow demands and/or link bandwidth, using real-world flows that download large Linux images from different servers. We configure each download as a separate flow group with different weights. Each flow individually has a demand higher than 30 Mbps as it is a backlogged flow with no server-side bottleneck.

Testing Reallocation and Reclamation. We test a scenario with 4 flows sharing a 30 Mbps link, with a desired sharing ratio of 4:3:2:1 between them. We emulate dynamic flow demands by shaping traffic at two interfaces in the home router. The first interface throttles rates of individual flows (to emulate flow demands limited by low sending rates or other upstream bottlenecks). The second interface then cumulatively throttles all the traffic to 30 Mbps, emulating an ISP's access link (as mentioned before).

Figure 7a shows the flow shares when we do WFQ at the ISP, which sets a perfect, but impractical baseline. Flow 1, which has the lowest weight of 1, starts at 0 seconds. Because there is no other active flow, it gets to utilize the entire link bandwidth. Flows 2, 3, and 4 become active after every 30 seconds respectively, and at each point, the link is shared in the proportion of active flows' weights. At 125 seconds, flow 4 stops, and link bandwidth is redivided between the remaining three flows in the proportion to their weights. At about 155 seconds, flow 3's demand drops to 10.5 Mbps, and its remaining share is taken up by flow 1 and flow 2. At 185 seconds, flow 2's demand also drops to 10.5 Mbps, at which point flow 1 gets all the remaining unused share.

Figure 7b shows how the flows share the link arbitrarily without any shaping with the status quo.

Figure 7c shows that, on the whole, CRAB is able to imitate the ideal WFQ baseline very closely despite being at the other side of the bottleneck (although there are some transient, and inevitable, dips in link utilization).

[9] We can only guess the protocols used by different content providers based on public knowledge.

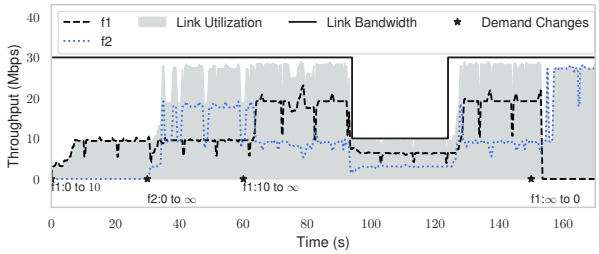


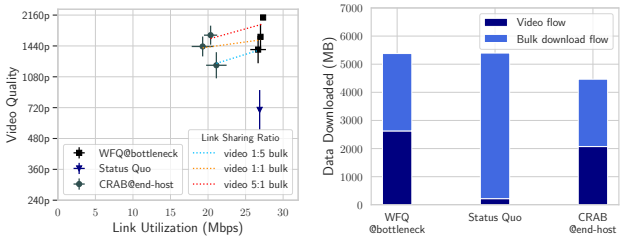
Figure 8: 2 flows sharing a link in 2:1 ratio with CRAB. Flow 1's demand and link bandwidth vary over time.

Testing Bandwidth Estimation. In Figure 8, we evaluate CRAB's reallocation and reclamation, in addition to bandwidth estimation due to varying link bandwidth. We have two flows configured to share a 30Mbps link in the ratio of 2:1. CRAB builds a spuriously low estimate of bandwidth when flow 1 (with demand limited to 10Mbps) starts at 0 seconds. CRAB is able to probe for more bandwidth once flow 2 (with a demand more than 20Mbps) starts at 30s. It reallocates the unused bandwidth of flow 1 to flow 2 as well. Flow 1's demand then increases at 60 seconds, CRAB reclaims its lent bandwidth and the link bandwidth is correctly shared in a ratio of 2:1 between flows 1 and 2 respectively. Then the link bandwidth drops to 10Mbps at 90 seconds, CRAB detects this change and adjusts the flows' rates to 6.66Mbps and 3.33Mbps respectively. When bandwidth increases again to 30Mbps at 125 seconds, CRAB is able to probe for more bandwidth and divide it according to the flows' weights. Finally, when flow 1 stops at 155 seconds, its bandwidth is reallocated to flow 2. Appendix D presents a similar experiment, except that instead of starting a single flow f2 at 30s with a demand of more than 20Mbps, we start 3 new flows, each with a demand of 10Mbps.

5.2 Video Streaming

We repeat the experiment in §2.2 with 7 different Youtube videos of varying playtime [10], competing with bulk download. Figure 9a reports the average video quality and link utilization across all experiments. We record the quality of each video over time using Youtube's API [13]. We then calculate the average video quality for each video by averaging the video quality weighted by the amount of time played at that quality. We calculate the average link utilization as the sum of data received during the video playback, divided by playback time. For CRAB and WFQ, we also try different weight assignments between video flow and bulk downloads, 5:1, 1:1, and 1:5 respectively. Figure 9a shows how CRAB maintains comparable video quality to WFQ for each weight assignment setting (i.e. within [92-94]% of WFQ), but with [15-20]% lower link utilization compared to status quo. The

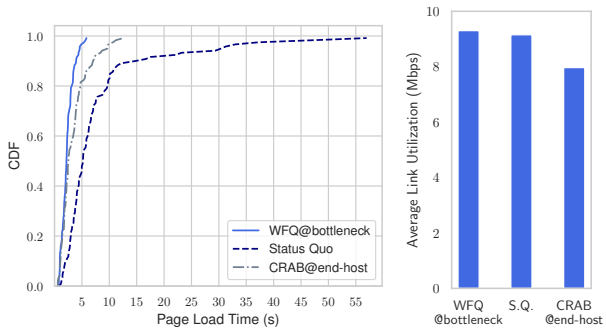
[10] shortest video is 1 minute, while longest is 10 minutes



(a) Average video quality vs Average throughput

(b) Distribution of downloaded data

Figure 9: Video streaming in presence of bulk downloads.



(a) Page Load Times

(b) Link Utilization

Figure 10: Page load times vs link utilization for CRAB, WFQ and Status quo

video quality achieved by status quo is worse than that with CRAB even with weighted sharing of 1:5 between video and bulk flows (this indicates that the video flow gets less than 17% of bandwidth share with the default status quo). [11]. Figure 9b shows the cumulative amount of video and bulk flow data downloaded across all videos for the experiment with a weighted sharing of 5:1 between video and bulk download. With CRAB and WFQ, video flow consumes almost half of the data which translates to much higher video quality. In comparison, with status-quo, video amounts to only 3% of total downloaded data.

5.3 Web Browsing

In this experiment, we show how CRAB helps improve web page load times despite background download flows. We emulate a user's browsing behavior by visiting 125 webpages (around 300 MBs of data) in total from 4 popular web domains (facebook.com, google.com, bbc.com, yahoo.com) in different sessions of browsing using Selenium [8] [12]. We separate each session by a Poisson inter-session time determined with a mean of 60 seconds. Within each browsing session, we separate each web page's download by a Poisson distribution with a mean of 5s to emulate a user's page read-time. We download two competing large files from two different servers. We throttle the access downlink to 10 Mbps at the router for these experiments and configure weights in the ratio of 7:3 between web traffic and bulk downloads. To fairly compare link utilization, we run the experiment for each baseline for the same amount of time. Figure 10a shows the CDF of page load times with CRAB, ideal WFQ, and status quo. The median page load time with CRAB is 2× smaller than with the status-quo and is within 15% of ideal WFQ. Figure 10b shows that CRAB under-utilizes the link by about 13%.

5.4 Alternative way of using CRAB's framework

We now evaluate the alternative mechanism for using CRAB's framework (referred to earlier in §2), where we directly throttle the cumulative rate of all flows arriving at the ingress to a value lower than the overall link capacity that CRAB estimates (via its throughput observation and slightly modified bandwidth probing logic), and then enforce the desired scheduling policy on the artificial bottleneck that gets created. To understand the trade-offs involved with this approach, we evaluate it under two different scenarios. We emulate (and assume) a static link capacity of 30Mbps, and do not implement bandwidth estimation for the alternative approach for simplicity. We also disable bandwidth estimation in the original CRAB implementation for a fairer comparison.

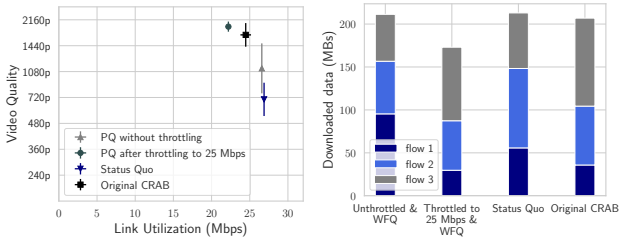
(a) We first consider the scenario from §5.2, where a YouTube video competes with bulk download on a bottleneck link with a capacity 30Mbps. We prioritize the video flow at the receiver without throttling the flows in one case, and after throttling the incoming flows to a cumulative rate of 25Mbps in the other. We compare these strategies with the status-quo (that does not enforce user preferences) and the original CRAB design. Figure 11a shows the results. Prioritizing the video flow without throttling cannot enforce user preferences very effectively (for reasons discussed in §2). However, prioritizing the video flow after throttling the incoming traffic to a rate of 25Mbps (which is lower than the link capacity) is effective. It results in slightly higher video quality but slightly lower link utilization than the original CRAB design. [13]

(b) We next evaluated a scenario where the 30Mbps bandwidth is to be divided across three backlogged flows in the ratio 1:2:3. We now apply weighted fair queuing at the receiver without throttling, and after throttling the incoming traffic to 25Mbps, and compare the outcomes with original CRAB and the status-quo (Figure 11b). Again, we observe that the desired shares could not be effectively enforced without throttling the flows to a rate lower than the link capacity. WFQ applied after throttling at 25Mbps was able to enforce the desired flow shares similar to the original CRAB. However, the original design achieved 19% higher link utilization

[11] Lower video throughput translated to lower resolution in all cases, and we did not notice any re-buffering events.

[12] Selenim allows us to automate webpage loads and user clicks.

[13] The difference in video quality potentially stems from the difference in scheduling policy – strict prioritization vs 5:1 weighted fair sharing with original CRAB.



(a) **(b)**

Figure 11: Creating chokepoint by throttling to less than known link capacity **(a)** helps control traffic (e.g. by using priority queues) **(b)** but results in avoidable bandwidth wastage.

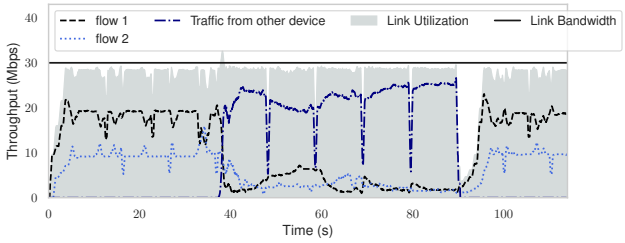
(which was very close to the status quo).

These results highlight that the original CRAB design *strives* to achieve maximal link utilization. The link underutilization is transient and less notable when flow demands are stable (as in the second scenario), and is more notable when flow demands vary due to more frequent re-allocation and reclamation (as in the first scenario). In contrast, the alternative approach of throttling the cumulative rate of incoming flows will consistently suffer from lower link utilization by design. The amount of link underutilization can be reduced by reducing the gap between the throttling rate and the link capacity, but this would also impact how effectively user preferences get enforced (e.g. resulting in lower video quality for the first scenario). This makes it difficult to correctly configure the cumulative throttling rate, especially as link capacity varies or is estimated imprecisely. Nonetheless, this alternative design effectively demonstrates the potential of using CRAB's framework in more than one way.

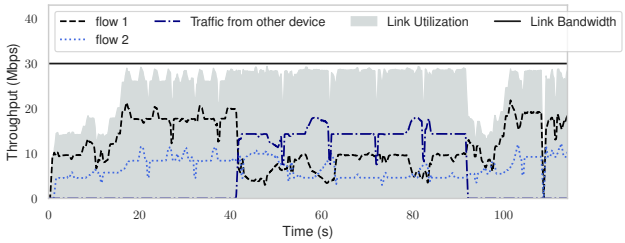
5.5 Multiple end-devices need CRAB at home-router

In this section, we show the need for CRAB at the home router to ensure proper enforcement of bandwidth shares when there are multiple devices actively using the Internet in the user's domain. We connect two machines (M1 and M2) to the home router. M1 runs CRAB to enforce 2:1 weights between two bulk download flows, while M2 does not run CRAB. Initially, we just have two flows from M1 sharing the bottleneck link in the 2:1 ratio enforced by CRAB. When the flow from M2 starts at around 40 seconds, in absence of CRAB support at the home router, it ends up stealing M1's bandwidth share (as shown in Figure 12a). When CRAB at M1 throttles its lower weight flow, the bandwidth yielded by this flow at the access link is taken up by the flow from M2, instead of the other higher-weighted flow at M1. [14] With CRAB enabled at the home-router, CRAB at an individual device can correctly control how its router-enforced bandwidth share is divided between its flows (as shown in Figure 12b). Thus, in case of multiple devices sharing the home Internet connection, it is important to enable CRAB at the home router to enforce

[14] Note that sender side protocols to yield bandwidth [41, 48] would suffer from a similar issue.



(a) Without CRAB@Router



(b) With CRAB@Router

Figure 12: With multiple active devices, CRAB at the home router is required to ensure correct working of CRAB at the end-host.

bandwidth shares across different devices, and to prevent the devices from stealing bandwidth from one another.

5.6 Impact of CRAB's Parameters

The value of n (number of observations) $\times$ t (observation interval) determines how long we spend in estimating throughput, before making a change in assigned rates. Figure 13 shows the effect of changing it from its default value of $(5 \times 0.2s)$ to higher $(10 \times 0.3s)$ and lower $(5 \times 0.1s)$ value for the video streaming experiment from §5.2. Higher value of $n \times t$ means we are much slower in our reactions – we reallocate late which improves video quality (very slightly) but at the cost of greater link under-utilization. In contrast, a smaller value of $n \times t$ implies quicker decisions – we have slightly better link utilization, but video quality also slightly drops. If we keep making observation length smaller, it would boil down to doing instantaneous reallocation similar to bandwidth borrowing with HTB (which, like status-quo, can maintain high link utilization, but cannot enforce bandwidth shares).

We also experimented with changing CRAB's lending headroom parameter from its default value of 0.25Mbps to higher (0.5Mbps) and lower (0.05Mbps) values. This had no significant impact on CRAB's performance – we present detailed results in Appendix E.

5.7 Other Results

We briefly summarize some of our other results, providing the details in the appendix:

• CRAB is quite robust to differences in RTTs and congestion control algorithms across flows, and it scales well with the increasing number of flow groups (Appendix C).

• CRAB has a negligible impact on packet delay and forwarding rates. It has a CPU utilization of 10.74% on a 2.4GHz

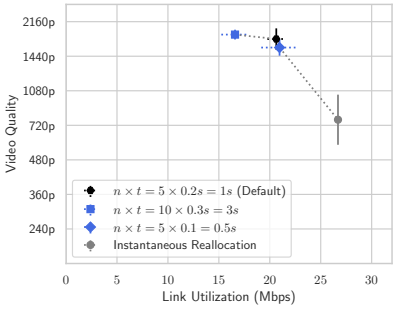


Figure 13: Effect of observation length ($n \times t$) on video quality vs link utilization.

8-core machine, which largely stems from the throughput measurement module (Appendix F). This is because our current implementation uses Scapy [16], a Python-based packet sniffer. Using a more efficient sniffer (e.g. libtins [4]) would reduce this overhead.

6 Related Work

There have been a number of proposals for enabling differentiated services (in the form of weighted fair sharing or prioritization) at network switches [17, 21, 25, 49, 53, 56]. However, these policies must be applied at the bottleneck, which is controlled by the ISP and not the users. There exist proposals that allow a user to send their preferences to the ISP [19, 24, 27, 35, 54] which are difficult to deploy in practice. CRAB allows the user to control the access bottleneck *without seeking any support from the ISP*.

There exist mechanisms for the device to control the uplink bandwidth usage when sending data [5, 44, 45], e.g. prioritizing latency-sensitive uploads over file backups [14] – the bottleneck occurs at the user device in these cases. Another category of work allows the end user to configure their home routers to do traffic prioritization [18, 40, 47], assuming that the bottleneck is at the wireless link in the home network. CRAB tackles the harder problem of controlling downlink bandwidth usage by shaping traffic *after* the bottleneck (that is likely to occur at the access link from the ISP), and naturally helps in scenarios where the bottleneck is at the home-router.

With bottlenecks at the ISP, it can even be challenging to do sender-side traffic prioritization. Bundler [20] solves this problem in context of site-to-site traffic by estimating the bottleneck rate in the ISP and enforcing that rate at the sender. This shifts the bottleneck at the sender's site instead of the ISP, which lets the sender enforce its desired scheduling policies. CRAB enforces desired bandwidth shares solely from the receiving domain, without seeking any explicit coordination with the senders.

Receiver-driven protocols [28, 42, 55] provide a receiver with greater control over their downlink bandwidth, by letting them explicitly dictate the sending rates. Some senders can also use bandwidth-yielding protocols (e.g. [41, 48]), if they know their flow has a relatively lower priority. However, the onus of using these receiver driven or yielding protocols is on the senders – a receiver can use these protocols only if the senders also support them. CRAB allows receivers to unilaterally control their access bandwidth shares.

7 Conclusion and Discussion

This paper presents CRAB, a system that enables end-user to unilaterally control how their Internet access bandwidth is shared across their incoming flows. In particular, we show how home users can exploit CRAB to enforce their preferences and achieve better performance for their video and web flows. Our source code is publicly available. [15] Our work opens up several interesting future directions:

Theoretical analysis of performance. Formal characterization of CRAB's performance, e.g. by analyzing the upper-bound on link utilization for effective enforcement of user-specified shares under different scenarios, can inform future designs for improved performance.

Other deployment modes. CRAB does not require any explicit coordination among the home router and the endpoints. This extends CRAB's utility to scenarios where multiple users share a common Internet connection, e.g. in coffee shops, enterprises, airports, etc. The domain owners can advertise their use of CRAB at the access routers for enforcing fairness across users (they can also use other scheduling mechanisms at the routers [1, 15, 18] if it is known that the bottleneck is at the downlink from the router to the end-devices). Each user can then use CRAB at the endpoint to independently control how their share of bandwidth is divided across their flows.

Setting Flow Weights. It might be difficult for users to set the appropriate weight for a flow group that CRAB requires as an input. Future work can explore how to design a more intuitive user interface. For instance, we can auto-classify incoming flows across broad categories (video streaming vs browsing vs downloads, etc), and then automate weight assignments based on coarse-grained user preferences across these categories and learned estimates of bandwidth requirements for different flows. Such bandwidth requirements are already known for many standard applications, e.g. video streaming [2, 9]. CRAB can also ship with some default configurations for popular traffic classes, which can be further customized by users according to their needs.

Support for phones. We currently implement CRAB on a Linux PC. We plan on porting our system to Android phones.

8 Acknowledgements

We would like to thank our shepherd, Srikanth Kandula, and the anonymous NSDI reviewers for their insightful comments. We would also like to thank Sachin Ashok, Hari Balakrishnan, Brighten Godfrey, Akshay Narayan, Aurojit Panda, Scott Shenker, Deepak Vasisht, and Tianyin Xu, for their helpful feedback on the paper. This work was supported by Intel, Facebook, and AG NIFA under grant 2021-67021-34418.

[15] https://projectcrab.web.illinois.edu.

References

[1] How qos improves performance? https://bit.ly/3f7IdXO.

[2] Internet connection speed recommendations. https://help.netflix.com/en/node/306.

[3] Internet speed around the world. https://www.speedtest.net/global-index.

[4] Libtins. http://libtins.github.io/benchmark/.

[5] Linux hierarchical token buckets. http://luxik.cdi.cz/~devik/qos/htb/.

[6] networking:ifb [wiki]. https://wiki.linuxfoundation.org/networking/ifb.

[7] Psutil. https://pypi.org/project/psutil/.

[8] Selenium with python¶. https://selenium-python.readthedocs.io/.

[9] System requirements - youtube help. https://support.google.com/youtube/answer/78358?hl=en.

[10] tc(8) - linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[11] Tcp bbr congestion control comes to gcp – your internet just got faster | google cloud blog. https://bit.ly/3qRKCsv.

[12] Tcpdump/libcap. https://www.tcpdump.org/.

[13] Youtube data api. https://developers.google.com/youtube/v3.

[14] 5 benefits and 3 drawbacks of using cloud storage for your baas offering. https://bit.ly/3qQSBGe, Mar 2018.

[15] Re: R6700v2 - where is downstream bandwidth control? https://bit.ly/3BYMAha, Dec 2018.

[16] Philippe Biondi and the Scapy community. Scapy. https://scapy.net/.

[17] Sj Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[18] Ilker Nadi Bozkurt and Theophilus Benson. Contextual router: Advancing experience oriented networking to the home. In *The Symposium on SDN research*, 2016.

[19] Ilker Nadi Bozkurt, Yilun Zhou, and Theophilus Benson. Dynamic prioritization of traffic in home networks. In *CoNEXT Student Workshop*, 2015.

[20] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Site-to-site internet traffic control. In *EuroSys*, 2021.

[21] Zhiruo Cao and E.W. Zegura. Utility max-min: an application-oriented bandwidth allocation scheme. In *IEEE INFOCOM*, 1999.

[22] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 2016.

[23] Pedro Casas, Michael Seufert, Florian Wamser, Bruno Gardlo, Andreas Sackl, and Raimund Schatz. Next to you: Monitoring quality of experience in cellular networks from the end-devices. *IEEE Transactions on Network and Service Management*, 2016.

[24] Saoussen Chaabnia and Aref Meddeb. Slicing aware qos/qoe in software defined smart home network. In *IEEE/IFIP Network Operations and Management Symposium*, 2018.

[25] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM CCR*, 1989.

[26] Sally Floyd, Tom Henderson, and Andrei Gurtov. The newreno modification to tcp's fast recovery algorithm. 1999.

[27] Hassan Habibi Gharakheili, Jacob Bass, Luke Exton, and Vijay Sivaraman. Personalizing the home network experience using cloud-based sdn. In *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2014.

[28] Rajarshi Gupta, Mike Chen, Steven McCanne, and Jean Walrand. A receiver-driven transport protocol for the web. *Telecommunication Systems*, 2002.

[29] Gabe Gurwin. Should you stick with console gaming, or make the jump into the cloud? https://bit.ly/3LxrHfX, Oct 2019.

[30] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 2008.

[31] Ada Ivanova. Streaming vs. downloading: Which one should you use? https://bit.ly/3QWgWox, Aug 2022.

[32] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM CCR*, 1988.

[33] Matt Joras, Matt Joras, and Yang Chi. How facebook is bringing quic to billions. https://bit.ly/3UoGRZ5, Apr 2022.

[34] Ravi Kokku. {TCP} nice: A mechanism for background transfers. In *OSDI*, 2002.

[35] Himal Kumar, Hassan Habibi Gharakheili, and Vijay Sivaraman. User control of quality of experience in home networks using sdn. In *IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, 2013.

[36] Ralf Kundel, Joerg Wallerich, Wilfried Maas, Leonhard Nobach, Boris Koldehofe, and Ralf Steinmetz. Queueing at the telco service edge: Requirements, challenges and opportunities. In *Workshop on Buffer Sizing*, 2019.

[37] Eymen Kurdoglu, Yong Liu, Yao Wang, Yongfang Shi, ChenChen Gu, and Jing Lyu. Real-time bandwidth prediction and rate adaptation for video calls over cellular networks. In *International Conference on Multimedia Systems*, 2016.

[38] Aleksandar Kuzmanovic and Edward W Knightly. Tcp-lp: A distributed algorithm for low priority data transfer. In *IEEE INFOCOM*, 2003.

[39] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*, 2017.

[40] Jake Martin and Nick Feamster. User-driven dynamic traffic prioritization for home networks. In *ACM SIGCOMM workshop on Measurements up the stack*, 2012.

[41] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. In *ACM SIGCOMM*, 2020.

[42] Venkata Padmanabhan. Coordinating congestion management and bandwidth sharing for heterogeneous data streams. In *NOSSDAV*, 1999.

[43] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Transactions on Networking*, 1994.

[44] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. {SENIC}: Scalable {NIC} for end-host rate limiting. In *USENIX NSDI*, 2014.

[45] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *ACM SIGCOMM*, 2017.

[46] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. Rtp: A transport protocol for real-time applications, 1996.

[47] M Said Seddiki, Muhammad Shahbaz, Sean Donovan, Sarthak Grover, Miseon Park, Nick Feamster, and Ye-Qiong Song. Flowqos: Qos for the rest of us. In *HotSDN*, 2014.

[48] Sea Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. Low extra delay background transport (ledbat). In *RFC 6817*, 2012.

[49] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 1996.

[50] Srikanth Sundaresan, Nick Feamster, and Renata Teixeira. Home network or access link? locating last-mile downstream throughput bottlenecks. In *International Conference on Passive and Active Network Measurement*, 2016.

[51] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *USENIX NSDI*, 2013.

[52] Gary R Wright and W Richard Stevens. *TCP/IP Illustrated, Volume 2 (paperback): The Implementation*. Addison-Wesley Professional, 1995.

[53] Haikel Yaiche, Ravi Mazumdar, and Catherine Rosenberg. A game theoretic framework for bandwidth allocation and pricing in broadband networks. *IEEE/ACM Transactions on Networking*, 2000.

[54] Yiannis Yiakoumis, Sachin Katti, Te-Yuan Huang, Nick McKeown, Kok-Kiong Yap, and Ramesh Johari. Putting home users in charge of their network. In *ACM Conference on Ubiquitous Computing*, 2012.

[55] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM*, 2015.

[56] Lixia Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM SIGCOMM CCR*, 1990.

[57] Xuan Kelvin Zou, Jeffrey Erman, Vijay Gopalakrishnan, Emir Halepovic, Rittwik Jana, Xin Jin, Jennifer Rexford,

and Rakesh K Sinha. Can accurate predictions improve video streaming in cellular networks? In *HotMobile*, 2015.

Appendix

A Pseudocode for Redivision of Excess Bandwidth

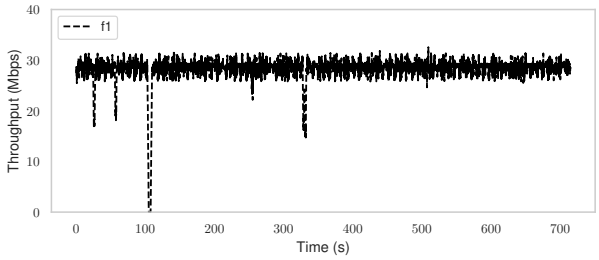
We first calculate the demand of each flow based on the amount of bandwidth it lends out. Then excess is divided based on this demand. If a flow's demand is fulfilled with bandwidth less than its share of excess, we can redivide this residual excess share between other flows.

Algorithm 1 Rediviging Excess Bandwidth between all flows

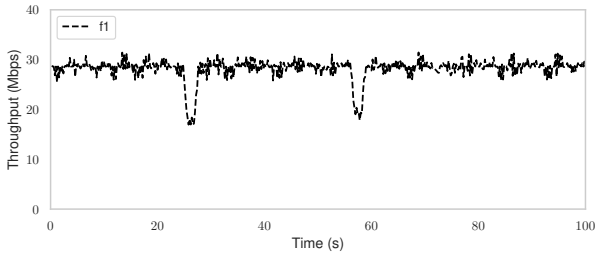
```
1:  procedure REDIVIDE(excess)
2:      // First we calculate demand of each flow based on
        the bandwidth it lends out
3:      for f in flows do
4:          if f.lent_bw > 0 then
5:              f.demand ← f.true_bw + f.borrowed_bw -
                f.lended_bw
6:          else
7:              f.demand ← ∞
8:          f.assigned_bw ← f.true_bw
9:          if f.demand > f.assigned_bw then
10:             f.lent_bw ← 0
11:             f.borrowed_bw ← 0
12:     // Based on the calculated demand, we divide excess
        between all flows. When a flow's demand is met, its
        residual excess is again divided between other flows.
13:     while excess > 0 do
14:         residual_excess = 0
15:         for f in flows do
16:             if f.demand > f.assigned_bw then
17:                 excess_share ← excess × (f.weight
                    /weight_sum)
18:                 f.assigned_bw ← f.assigned_bw + ex-
                    cess_share
19:                 f.borrowed_bw ← f.borrowed_bw + ex-
                    cess_share
20:                 if f.assigned_bw > f.demand then
21:                     residual_excess ← residual_excess +
                        (f.assigned_bw - f.demand)
22:                     f.lent_bw ← f.lent_bw +
                        (f.assigned_bw - f.demand)
23:         excess ← residual_excess
```

B Stability of Wifi Connection

Cellular networks are known to be highly unstable due to factors like high mobility and handovers. Wifi connections are relatively more stable. We evaluated this by sending IPerf data over UDP at a fixed rate of 30Mpbs to a Linux machine via a WiFi router. We measured the throughput over 200ms granularity at the ingress of the Linux end-host using tcpdump. Figure 14 shows the results. The observed throughput was



(a)



(b) Zoomed into first 100 seconds.

Figure 14: Throughput of a 30 Mbps flow over Wifi measured in 200ms intervals.

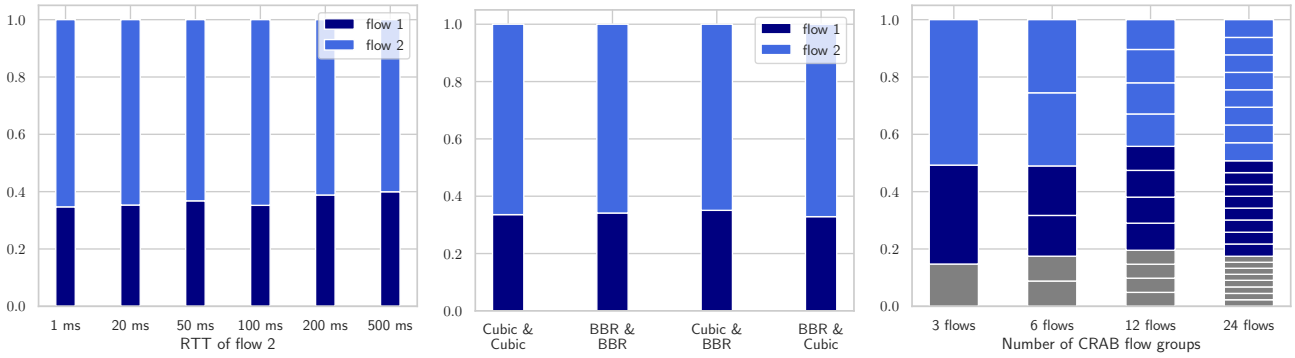
largely stable with minor fluctuations around 30Mbps and only a handful of dips.

C Robustness to Different Traffic Characteristics

We now evaluate CRAB's performance under diverse traffic characteristics – flows with different RTTs, using different congestion controllers, and varying the number of flow groups. For these experiments, we generated iPerf flows with different characteristics using a local server, which then arrived at our receiver side setup used in our other experiments so far.

In the first experiment, we vary the RTT of flows by adding artificial delay in packet delivery using Linux tc at the server that generates flows. We start three backlogged flows sharing a 30 Mbps link in a 1:2:3 ratio. We fix the delay of the first flow (with weight 1) and the third flow (with weight 3) to 1ms and 50ms respectively, and vary the delay of the second flow (with weight 2) from 1ms to 500ms. We stop the third flow after 30 seconds and continue to run the other two flows until 100 seconds. We then study the effect of different RTTs for the first two flows as CRAB redivides the third flow's share between them in a 1:2 ratio. As shown in figure 15a, CRAB is pretty robust to the difference in RTTs. The slight mismatch in flow shares seen with an extremely high RTT difference of 200-500ms stems from the natural RTT unfairness that occasionally manifests in CRAB during the bandwidth probing phase when both flows share the bandwidth increment in a non-isolated manner.

We use a similar setup as above for our second experiment, except that the flows now have the same RTTs (20ms), but use different congestion control algorithms. The third flow uses



(a) Varying RTT difference **(b)** Different congestion control algorithms for both flows **(c)** Number of active CRAB flow groups

Figure 15: CRAB maintains weighted sharing despite different characteristics of flows.

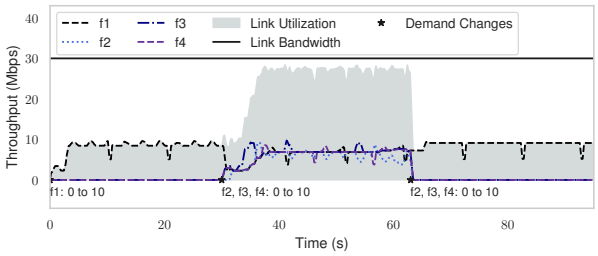


Figure 16: CRAB is able to estimate link capacity even with finite demand flows.

TCP Cubic and stops after 30 seconds. We vary the congestion control algorithms used by the other two flows as shown in figure 15b and observe how that impacts their flow shares. We find that CRAB's enforcement of weighted fair shares is robust to different congestion controllers. **[New]** CRAB is unaffected by the unfairness that could manifest because of RTT and congestion controller difference because it reacts at super-RTT time scales thus forcing underlying flows to adhere to throttled rates.

In the last experiment, we test CRAB's robustness as we increase the number of flow groups from 3 to 24. In the first run, we have three flows sharing a 60 Mbps link in a 3:2:1 ratio. In the next run, we double the number of flows associated with each weight, and so on. Figure 15c shows the bandwidth share received by each flow, with different colors indicating flows with different weights. We find that CRAB can effectively tackle a large number of flows. **[new]** As long as flows are large enough to react to CRAB, any number of flows can be handled by it. The only breaking point may be when a flow group consists of a large number of short-lived flows which finish before reacting to CRAB's throttling. However, such a case is unlikely to exist in our target scenario of a home network.

D Bandwidth Probing with Limited Demand Flows

Extending on our discussion in §5.1, here we evaluate the scenario when we do not have a convenient infinite demand flow to rely on for bandwidth probing. CRAB is still able to quickly probe for bandwidth by alternating between different finite demand flows for bandwidth probing. Figure 16 shows a scenario where we initially have one flow with a demand of 10 Mbps, at 30 seconds, 3 new flows each with a 10 Mbps demand start. Since their cumulative demand is more than 30 Mbps, the bandwidth probing algorithm is able to estimate link capacity by alternatively picking a flow for bandwidth probing and dividing capacity equally between them.

E CRAB's Sensitivity to Lent Bandwidth Headroom

The lent bandwidth headroom ensures that a flow has some room in the link to send at least a few packets so CRAB can detect it to be growing and reclaim for it. When the bandwidth of a flow is detected to be exceeding this headroom, CRAB quickly reclaims for it. Figure 17 shows CRAB's sensitivity to this parameter through the video experiment discussed in §5.2. Overall, CRAB is not very sensitive to this parameter, but

A larger value of headroom ensures better guarantees on early detection for reclamation, thus, slightly better video quality. However, overprovisioning may result in under-utilization, especially if we have a much higher number of flow groups. This effect can be avoided easily by having a cap on the collective headroom of all flow groups combined. A smaller value of headroom may not guard very well against pressure from other flows, which may result in CRAB not being able to detect flow growth in time and therefore slightly worse video quality. Another hidden effect that deteriorates link utilization in case of small headroom is spurious reclamations. Small values of headroom are not able to mask minor fluctuations and noise, which results in spurious reclamation, as a result,

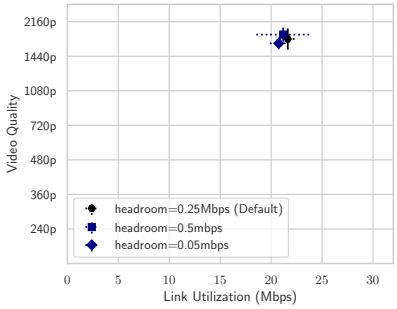


Figure 17: Effect of lent bandwidth headroom on video quality vs link utilization.

	With CRAB	No CRAB
Throughput	28.93 Mbps	28.98 Mbps
Delay	0.94 ms	0.88 ms
CPU Usage@end-host	10.74%	N/A

Table 2: Overheads of CRAB

we see slight link under-utilization. Overall CRAB is not very sensitive to this parameter.

F Overhead of CRAB

We evaluate CRAB's overhead by measuring the throughput and delay of a single bulk download flow with and without CRAB. To measure throughput, we record the flow's rate at ifb's egress (i.e. after shaping) with CRAB, and at eth0's ingress (as the raw arrival rate) without CRAB. We measure processing delay by recording the difference between timestamps for when a packet arrives at the eth0 and when its acknowledgment passes through eth0. Since CRAB's components are placed after the eth0 interface on the path of ingress traffic, this calculation captures any extra delay inflicted by CRAB. Table 2 shows that CRAB does not induce any significant overhead (the throughput remains largely unchanged, and the processing delay increases by only 0.06ms (i.e. 6.8% over baseline).

We also measure the CPU utilization of all CRAB threads during the experiment using Linux utility top. On a 2.4GHz 8-core machine, CRAB has an overall utilization of 10.34%. Almost all of the CPU usage stems from the throughput measurement thread of CRAB due to traffic sniffing. This is because Scapy, the Python-based packet sniffing library we use, copies the entire packet even though we just need access to a few packet header fields. The corresponding CPU overhead at the home router, which uses tcpdump for sniffing, is 16.65% on two cores at 1.8GHz. Writing a custom sniffer for CRAB that copies only a few packet header fields can potentially reduce the CPU overhead. We are working on shifting our throughput measurement module to a faster packet sniffing library like libtins [4].

*x*BGP: Faster Innovation in Routing Protocols

Thomas Wirtgen*
ICTEAM, UCLouvain

Tom Rousseaux
ICTEAM, UCLouvain

Quentin De Coninck
ICTEAM, UCLouvain

Nicolas Rybowski
ICTEAM, UCLouvain

Randy Bush
Internet Initiative Japan & Arrcus, Inc

Laurent Vanbever
NSG, ETH Zürich

Axel Legay
ICTEAM, UCLouvain

Olivier Bonaventure
ICTEAM, UCLouvain

Abstract

Internet Service Providers use routers from multiple vendors that support standardized routing protocols. Network operators deploy new services by tuning these protocols. Unfortunately, while standardization is necessary for interoperability, this is a slow process. As a consequence, new features appear very slowly in routing protocols.

We propose a new implementation model for BGP, called *x*BGP, that enables ISPs to innovate by easily deploying BGP extensions in their multivendor network. We define a vendor-neutral *x*BGP API which can be supported by any BGP implementation and an eBPF Virtual Machine that allows executing extension code within these BGP implementations. We demonstrate the feasibility of our approach by extending both FRRouting and BIRD.

We demonstrate seven different use cases showing the benefits that network operators can obtain using *x*BGP programs. We propose a verification toolchain that enables operators to compile and verify the safety properties of *x*BGP programs before deploying them. Our testbed measurements show that the performance impact of *x*BGP is reasonable compared to native code.

1 Introduction

Internet Service Providers (ISP) are continuously challenged by their users and customers to provide value-added services that go beyond best-effort connectivity. Among others, these new services include traffic engineering techniques to prioritize some flows over others and improve network load, fast reroute mechanisms to swiftly retrieve connectivity upon failures, or anycast routing. In addition, ISPs are trying to improve their internal operations in order to provide an ever better service to their customers. This can be done by implementing a monitoring system, re-architecting or tuning the internal network.

Almost invariably deploying these services require extending routing protocols. And among all protocols, the Border Gateway Protocol (BGP) is probably the most used one given its flexibility: for many network operators, BGP has become a true "Swiss-army knife". Originally designed to distribute interdomain routes, BGP has been extended several times to support different types of services [41, 55].

While extending BGP is possible, it is certainly not easy, for two main reasons. First, ISP networks often include routers from different vendors [17, 69]. This diversity is inherent and required for technical, safety, and economic reasons. Unfortunately, this diversity means that operators can only use the *intersection* of the features set across all their routers, hindering flexibility.

Second, it can take years for even a subset of the vendors to implement new features as these need to be first standardized by the Internet Engineering Task Force (IETF). Many view this as a form of ossification of the routing protocols. As an illustration, a recent paper [79] showed that the median delay before RFC publication of BGP extensions is *3.5 years*, and that some features required *up to ten years* before being standardized.[1] This is only the tip of the iceberg though: only a small subset of the BGP extensions proposed by network operators have been discussed and later adopted by the IETF.

Of course, this is not a new story. Frustrated by these delays and the difficulty to innovate in networks, researchers have argued for Software-Defined Networks (SDN) [48] for more than a decade. Instead of relying on a myriad of distributed protocols and features, SDN assumes that switches and routers expose their forwarding tables through a standardized API. This API is then used by logically centralized controllers to "program" routers and switches.

While SDN has enabled countless new research works [21, 42], it has not been widely adopted by ISPs. One of the main hurdles is that deploying SDN requires a major network overhaul, both at the control-plane level, to deploy scalable and robust logically-centralized controllers, *and* at the

*Thomas Wirtgen is supported by a grant from F.R.S.-FNRS FRIA.

[1] Note that this delay ignores the time elapsed between the initial idea and its first adoption by the working group, making the actual delay even longer.

data-plane level, to deploy compatible network devices. Thus far, only large cloud providers managed to perform this overhaul [36, 39].

Of course, instead of relying on commercial routers, network operators could decide to adopt open-source implementations of routing protocols [16, 23, 33, 67] running on servers or custom hardware [3]. A network operator could for instance fork a BGP implementation to add a desired feature. Maintaining this fork requires a lot of software development effort though. Such an approach is feasible for large cloud providers [62] but not for ISPs. Another approach is to use a modular routing implementation to take full control of the protocol. The network operator is responsible for the entire routing implementation. Unfortunately, it is too difficult to maintain and evolve because the network operator must have a complete understanding of the routing protocol and must have software programming skills, which they often do not have. To provide flexibility in the administration and automation of their routers, router vendors have added a Python interpreter to their operating systems [40]. However, the interpreter only handles the administration part of the router and does not provide an interface to add or modify protocol features. Finally, the use of active networks with centralized approaches or descriptive configuration languages [10, 27] is not possible in today's Internet, as autonomous systems still use decentralized protocols to establish peering links.

In this paper, we argue for much lighter weight and practical approach to network control plane programmability by *allowing the network operators to easily extend the distributed routing protocols that they already use*. Our new approach, which we call *x*BGP, is inspired by the success of the extended Berkeley Packet Filter (eBPF) in Linux [26, 35] and Windows [49]. eBPF is an in-kernel Virtual Machine (VM) that relies on a custom instruction set. Thanks to eBPF, programmers can easily (and securely) deploy new programs that can access a subset of the kernel functions and memory [26]. Similarly, in *x*BGP, different BGP implementations expose an API and an in-protocol VM with a custom instruction set to access and modify the intrinsic protocol functions and memory. Thanks to this API and the VM, the *same* code can be executed on different implementations. Note that the instructions set and the in-protocol VM still need to be adopted and implemented by each vendor, but this is a one-time effort, instead of a *per-feature* effort.

Naturally, opening up BGP implementations to external programs opens the door to many (research) questions: *Which API should BGP expose? How to implement this API efficiently* or *What about the correctness and the safety of these extensions?* We answer these questions in this paper and make four main contributions.

First, we introduce the x*BGP API* which defines a set of functions that should be supported by an extensible BGP implementation. We present this API in Section 2 and describe how we modified two different BGP implementations, BIRD and FRRouting, to support *x*BGP.

Second, we present a complete validation workflow that enables operators to validate that their extensions correctly terminate, do not interfere with the memory of the host implementation, produce syntactically valid BGP messages, or only use the *x*BGP API functions authorized by the network operator. We envision this workflow to become one element of the qualification tests that operators already carry out before deploying any new BGP feature in their network.

Third, we showcase the practicality of *x*BGP by implementing eleven use cases with *x*BGP to: support a new BGP attribute; introduce new selection rules; restrict the set of paths it can compute; detect unused routes (zombies); or monitor BGP operations. Each use case involves the same *x*BGP bytecode running on both FRRouting and BIRD.

Fourth, we demonstrate the practicality of *x*BGP by measuring its overhead compared to native implementations. Even for complex extensions (re-implementing BGP Route Reflection), our benchmarks show that the overhead of *x*BGP is always under 13%, a reasonable value given the flexibility benefits.

Similarly to what OpenFlow [48] achieved, we believe that programmable distributed routing protocols have the potential to open up *many* promising avenues for research, while being fundamentally more practical and deployable.

2 Architecture

At a high level, *x*BGP enables network operators to customize or extend any compatible BGP implementation by injecting and directly executing *x*BGP programs. As an illustration, we consider how to expand a BGP implementation to support a new BGP attribute, *GeoLoc*, that stores the geographic location (i.e., longitude and latitude) of where each BGP route was learned. Among others, this attribute can be used to adapt router decisions, e.g., by filtering away routes learned more than *x* kilometers away. Supporting such an attribute has been discussed within the IETF but never standardized [13]. Yet, large-scale ISPs reportedly use iBGP filters [71] to achieve the same effect. Using iBGP filters is risky though as doing so can lead to permanent oscillations [71].

To implement the *GeoLoc* extension, we need to support several operations in a BGP implementation. (1) When a route is received over an eBGP session, the router adds a new attribute, `Geo_Originator` that contains the geographic coordinates of the router that learns the BGP route in an import filter. (2) If the BGP route already contains the `Geo_Originator` attribute, the router needs to decode it. (3) When exporting the route to another peer, the router can use the `Geo_Originator` attribute to filter routes that are too far away. (4) To be usable by other iBGP peers, the attribute needs to be added to the BGP Update message.

To add this extension, we need to understand how BGP implementations are designed. There are many ways to organize

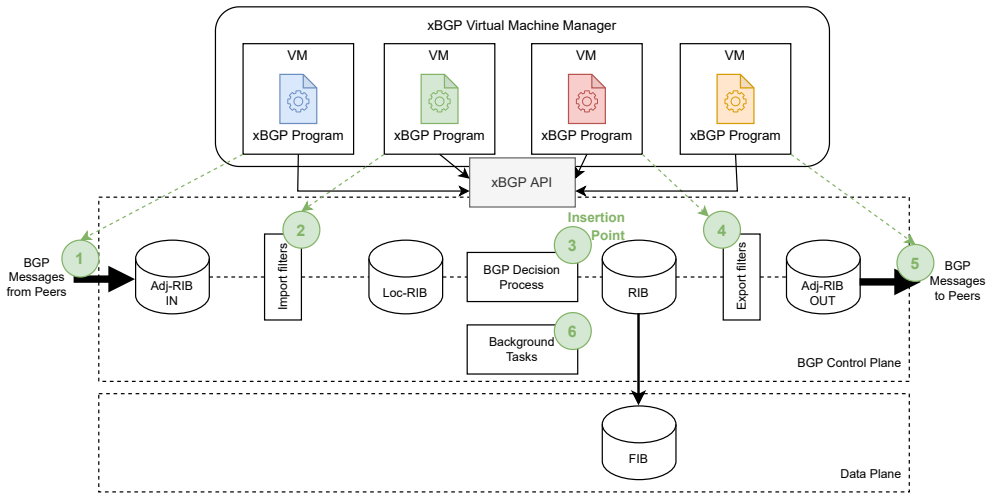


Figure 1: An *x*BGP compliant implementation exposes the abstract BGP data structures defined in RFC4271 through a generic API and uses `libxbgp`'s Virtual Machine Manager to attach the bytecode that implements extensions to specific insertion points (green circles). The four bytecodes in this example support a simple GeoLoc BGP attribute. For each *x*BGP program, we provide the set of helper functions used to retrieve data from the host implementation.

a BGP implementation. Each implementer selects a particular software architecture and the associated data structures based on their own requirements. However, all BGP implementations must adhere to the protocol specification [54]. This specification defines the format of the BGP messages, an abstract BGP Finite State Machine that manages each BGP session, and also an abstract workflow and data structures that describe how BGP update and withdrawal messages should be processed. This workflow is illustrated in black in Figure 1. Starting from the left, a received BGP message is stored in the `Adj-RIB-in` [2]. It then passes through the `import` filters that may decide to discard the message or modify attributes such as `local-pref`. If the route is accepted by the import filters, it is inserted in the `Loc-RIB`. The `Loc-RIB` contains all the BGP routes accepted by the router. The BGP decision process extracts from the `Loc-RIB` the best routes that are placed in the `RIB`. These routes then pass through the `export` filters before being advertised over BGP sessions.

Going back to our *GeoLoc* extension, we can see that it can be added to the different parts of the BGP workflow. (1) needs to be added to the part that parses a BGP attribute. (2) and (3) must be designed as import and export filters respectively. And (4) will be added to the serialization part of the BGP implementation. The question now is how to add those subcomponents to the main BGP implementation. To answer this, we defined the insertion points depicted in Figure 1 with the green circles on which functionalities can be added or modified. These insertion points correspond to the major BGP events. It is now easy to add the four components of our simple extension to the BGP implementation in their respective insertion points. (1) is attached to the `BGP_RECEIVE_MESSAGE` ① insertion point. First, it queries the BGP neighbor's table and determines the type of the eBGP session. Then, it retrieves the contents of the received BGP update in network byte order. Finally, it attaches the new GeoLoc attribute to the route. The second program (2) is attached to the `BGP_INBOUND_FILTER` ② insertion point. It retrieves the router coordinates from the router configuration to add them to the attributes of the route. The program (3) attached to the `BGP_OUTBOUND_FILTER` ④ retrieves the neighbor information and the *GeoLoc* attribute to check if the route can be advertised to the peer. Finally, the fourth program (4) is attached to the `BGP_ENCODE_MESSAGE` ⑤ insertion point. It uses the BGP GeoLoc attribute received over an iBGP session decoded by the first program and sends it to the peer.

To be able to dynamically augment the BGP implementation, the four *x*BGP programs are executed inside a Virtual Machine and are attached to specific *insertion points* in the BGP implementation. An *x*BGP program is composed of eBPF bytecode executed by a user space virtual machine that is included in any *x*BGP compliant implementation. Thanks to this eBPF virtual machine, the same *x*BGP program can be executed on the CPUs used by different router platforms.

An *x*BGP program is not a standalone executable that performs computations autonomously. It can interact with the underlying BGP implementation, access its data structures, and call some of its functions. In contrast with operating system kernels such as Linux, FreeBSD or macOS that expose a similar POSIX interface, there is no standard API for BGP implementations. *x*BGP must then propose a common API to support several BGP implementations. If we take our ex-

[2] Some implementations do not explicitly maintain a separate `Adj-RIB-{in,out}` to reduce their memory consumption and store everything in the `Loc-RIB`. We ignore this implementation detail in this paper.

tension, when the *GeoLoc* program has finished decoding the `Geo_Originator` attribute, it must update the BGP route stored in the BGP implementation. Thus, our extension needs to fetch or set data from the host implementation. For this, the BGP implementation must propose a set of functions, the *x*BGP API [75], that enable the interactions between the extension and the internal data structures. For example, with a call to the function `set_attr`, the extension can add a new attribute to the BGP route being processed.

An important data structure of a BGP implementation is its Routing Information Base (RIB). It contains the routes selected by the BGP decision process and pushed in the Forwarding Information Base (FIB). The BGP RIB stores, for each known destination prefix, its BGP route containing its BGP attributes, including its AS-path and the address of the BGP next hop. The RIB also contains information from the intradomain routing protocol such as the cost to reach each next hop. BGP implementations use various data structures to store their RIB. Some implementations simply store the BGP attributes as they were received from the wire. Others use a specific structure for each type of attribute. To ensure that the same *x*BGP program can be executed on any compliant implementation, *x*BGP defines its own representation for IP prefixes, next hops, and BGP attributes. For the latter, *x*BGP simply relies on the wire format [54]. *x*BGP also defines a neutral representation of the BGP neighbor's table. With these representations, *x*BGP programs can access the data structures of the underlying BGP implementation. When required, *x*BGP converts the internal representation to its own format before returning data to *x*BGP programs and vice versa.

The remaining of this section describes the composition of the *x*BGP API enabling *x*BGP programs to interact with BGP implementations in Section 2.1. Section 2.2 shows how we execute an *x*BGP program inside the BGP implementation. We explain in Section 2.3 which challenges we faced to make two BGP implementations, BIRD and FRRouting, *x*BGP compatible.

2.1 The *x*BGP API

Besides some utility functions (memory management, conversion between network and host byte orders, simple math functions, etc.), most of the *x*BGP API is specific to BGP [75].

To modify internal BGP data structures, *x*BGP programs rely on `getters` and `setters` to access data structures stored on the host implementation. This ensures (i) an isolation layer between the host and the *x*BGP program and (ii) a uniform method of accessing data regardless of the BGP implementation. These functions convert the internal representation into a universal one understood by *x*BGP programs. In addition, extension codes require access to the BGP internal state (e.g., list of peers, the route attributes, the route next hop). Hence, *x*BGP requires BGP implementations to provide routines translating their internal data structures into *x*BGP ones. These include `getters` and `setters` to access/modify a BGP route including its attributes, next hop and the data that identifies a BGP peer. We also provide functions to iterate the RIB. These enable searching for a route other than those provided by the insertion points, and therefore for searching routes already installed in the BGP routing table.

Existing router OSes do not provide a common way to access internal routing data. The *x*BGP API provides functions to access IGP data, e.g., to retrieve the next hop for routes and use them in use cases described in Section 5.

An *x*BGP program can deliberately send a custom BGP message to any peer it wants. Instead of relying on an insertion point to generate the message, the *x*BGP API contains functions to send BGP messages allowing a program to send an urgent message like a BGP notification because the *x*BGP program detected a problem with a given peer.

To access non-standard data such as the geographic coordinates of the router, an extension code may require additional configuration. One approach is to directly include the data inside the code of the *x*BGP program. However, this is not scalable if the operator wants to deploy it on a large number of routers. This induces a recompilation of the code for each of its router. Instead, the *x*BGP API proposes to the network operator to include a configuration data part in a structured textual file accompanying *x*BGP programs called *manifest*. The *x*BGP program uses it later to retrieve what it needs. This extra configuration part is not directly accessible to the *x*BGP program but can be accessed through a set of API functions.

Finally, *x*BGP programs can be executed as background tasks ⑥ that are called when a timer expires. These tasks are not triggered by a specific BGP event like an insertion point but are rather executed when a timer expires. Background tasks are only used for processes that do not interact with the BGP workflow. Each task controls its timer and *x*BGP deliberately restricts one timer per task to avoid timer explosions. However, the task may ask to queue forever as long as the BGP router is alive. This is particularly interesting for *x*BGP programs that make routine maintenance for example. Each background task is executed in a dedicated thread to allow the original BGP implementation to run in parallel. If the *x*BGP program must access or update data, the *x*BGP API must be thread safe. This constraint must be respected when implementing the *x*BGP API.

2.2 Executing *x*BGP programs

An *x*BGP program is a set of eBPF bytecodes, either attached to different insertion points or executing background tasks. Each *x*BGP bytecode has its own dedicated memory, including a stack and a heap that are automatically freed after execution. This memory isolation between extension codes is guaranteed by the eBPF virtual machine. This ensures that orthogonal extensions will not interfere with each other. Yet, *x*BGP programs may need to keep persistent storage or to

exchange data between the different bytecodes that compose a program. For this, the *x*BGP API provides a key-value store that is similar to the BPF maps used in the Linux kernel.

Each *x*BGP implementation includes userspace eBPF virtual machines that are controlled by a manager. The *Virtual Machine Manager* (VMM) attaches bytecode with an associated virtual machine to one specific insertion point exposed by the host implementation. Each *x*BGP program includes a manifest listing the extension codes and their insertion point. Different extension codes can be attached to the same insertion point, and the manifest defines in which order they are executed. The manifest also lists the different *x*BGP API functions that the bytecode may use.

An *x*BGP program can be attached at different insertion points, i.e., specific code locations in a BGP implementation from where the program can be called. These insertion points correspond to specific operations that are performed during the processing of BGP messages, enabling *x*BGP programs to modify the router's behavior. *x*BGP defines six generic insertion points (green circles in Fig. 1) based on the original definition of BGP [54]. The sixth insertion point is dedicated for background tasks.

By default, the VMM only runs one *x*BGP program per insertion point. *x*BGP programs must explicitly tell the host implementation to run the next *x*BGP program if any through the `next()` function. This mechanism avoids executing useless code. For example, if we attach two *x*BGP programs that parse different BGP attributes into the insertion point that processes a single BGP attribute, and the first program successfully parses the message, there is no need to run the second one.

2.3 Adding *x*BGP to BGP implementations

To demonstrate the feasibility of *x*BGP, we have adapted two open-source implementations: BIRD v2.0.7 [16] and FRRouting v7.3 [23].

Adding the *x*BGP API. Implementing the API induced a total of 400 and 589 additional lines of code [78] on BIRD and FRRouting, respectively. The difference between both is their internal representations of the BGP data structures. The *x*BGP functions that deal with BGP messages and attributes always manipulate them in network byte order (*x*BGP's neutral representation), performing the translation to the storage format used by the implementation if required. FRRouting uses an internal representation that is different from our neutral one. We thus had to implement several functions to do the conversion between the two representations. Another difference is the handling of BGP attributes. BIRD includes a flexible API to manage BGP attributes. *x*BGP simply extends this API. FRRouting does not include such an API, so we had to implement one to be able to manipulate BGP attributes in BGP updates.

Integrating `libxbgp`. `libxbgp` is a portable library, implemented as 432 lines of header code, which consists of two parts: (*i*) the utility functions of the *x*BGP API; and (*ii*) the VMM. The VMM is in charge of executing the right extension code according to the state of the host implementation. This layer acts as a multiplexer. To include *x*BGP operations, the BGP implementation calls the VMM to execute the associated extension codes. Then, the VMM proceeds as follows. It first checks if there are attached extension bytecodes to the called *x*BGP operation. If not, the VMM executes the default function provided by the implementation. Otherwise, it runs the first extension code mentioned in the manifest. Two outcomes are possible. First, the extension code provides a result for the operation and the VMM returns the output to the caller. Second, the extension code delegates the outcome to another one by calling the special `next()` function. In that case, the VMM checks whether there are remaining codes in the ordered queue. If there are, the VMM runs the next extension code in its virtual machine. Otherwise, the behavior of the *x*BGP operation falls back to the default function provided by the BGP implementation. For instance, two extensions can attach bytecode to the `BGP_RECEIVE_MESSAGE` operation that processes their own dedicated BGP attribute, calling `next()` once they are done.

Technical challenges. While adding the *x*BGP API and integrating `libxbgp`, we encountered some interesting technical issues. To successfully use the *x*BGP API, data must be available when the function is called. In some cases, data in the host implementation was not available when the insertion point was called to execute the extension code. For example, in FRRouting, export filters are applied to a set of peers sharing the same type of outbound policies. This set is not passed to the code checking the outbound policies but is required to implement the helper function that retrieves data about the BGP peers of the router. We had to write 5 extra lines of code to get the set of peers before calling the insertion point. Also, some data structures were not flexible enough to fully support the *x*BGP API such as the function that adds or modifies a new attribute to a BGP route. However, the internals of FRRouting do not allow adding unsupported attributes that are not defined by any standard (e.g., `ORIGINATOR_ID`). We rewrote this part of FRRouting. To address those issues, we had to add 30 and 10 lines of code to FRRouting and BIRD respectively.

Each API function is called within a context of execution. This context is hidden within the extension code but visible in the host BGP implementation. This makes it possible to control which extension code has called the function. The context is also used to retrieve variables that cannot be directly used inside the extension code. For example, if an extension code needs to allocate extra memory (ephemeral or not), the ephemeral memory is also automatically freed when the extension code terminates its execution. Similarly, the context enables helper functions to access data structures that are out of the extension code's scope. For instance, a dedicated helper

function enables an extension to add a new route to the RIB. When setting an insertion point, the BGP implementation can pass a set of arguments. While some are visible inside the extension code, others are not. The RIB function leverages such hidden arguments to access the data structure while being transparent to the extension code.

Limitation of *x*BGP. To better understand what can and cannot be done with *x*BGP, we analyzed the complete list of RFCs, which defines extensions to BGP, that have been published since the publication of RFC4271 [54]. The RFCs can be classified into two different types. (1) The RFCs that modify the original definition of RFC 4271 (6 RFCs) and (2) those that add features on top of BGP (30 RFCs). For (1), *x*BGP cannot be used to implement these types of RFCs because it requires a direct modification of the underlying BGP implementation. For example, increasing the internal buffer size of the BGP message size [9] is not feasible with *x*BGP. For (2), *x*BGP can be used. However, it turns out that our current prototype focuses only on the messages that BGP speakers exchange once the session is established (BGP Updates and BGP Withdraw). Not all session-level extensions to BGP can be handled by *x*BGP. For example, our current prototype cannot extend the BGP Open or BGP Route-Refresh [12] message. However, *x*BGP contains a generic insertion point, `DECODE_BGP_MSG`, that can handle future types of BGP messages. If the underlying BGP implementation does not support route refresh, we can implement it as an *x*BGP program. Modifying *x*BGP to allow it to support the session level could be implemented at a later time, but *x*BGP cannot change the architectural design of the underlying implementation. This is an important limitation of our solution. For example, no *x*BGP program can increase the size of the BGP transmit and receive buffers as defined in the corresponding RFC [9]. The internal structure of an implementation cannot be modified on the fly by a program since the definition of the structures is strongly integrated in the program binaries.

In addition to the limitation of the features that can be implemented, *x*BGP focuses mainly on the internal network, we assume that the network operator enables the necessary *x*BGP programs on the relevant routers. However, if the BGP routers decode an unknown message, it will be silently discarded and will not harm the router but will compromise the other router's computation. BGP capability negotiation messages can be exchanged to indicate whether the extension implemented by the *x*BGP program is supported by both routers implied in the BGP session. Capability support is beyond the scope of this paper.

3 Ensuring the safety of *x*BGP programs

BGP implementations generally run 24/7 and never stop. When operators deploy a new router or a new version of a router operating system, they typically run extensive tests to verify that the new feature will not break their network. From an operator's viewpoint, injecting an *x*BGP program is always risky since the program will be executed within the BGP implementation. A simple approach would consist in letting the VMM monitor their execution and stop them in case of error. This could be too late for errors that could disrupt BGP sessions. Network operators typically need some safety guarantees from the *x*BGP program. The Linux kernel copes with a similar problem by using a custom online verifier [1] that checks different aspects of eBPF programs *before* they are injected into the kernel.

Verifying *x*BGP programs. *x*BGP also relies on verification techniques to ensure that programs can be safely injected. However, instead of developing a custom verifier [24], we (*i*) establish a list of properties that an *x*BGP program should respect to be considered as safe and (*ii*) we build a toolchain embedding three existing and well-tested software verification tools allowing the verification of our properties. Our *x*BGP toolchain receives the *x*BGP programs as input. They consist of C code that uses the *x*BGP API and a manifest provided by the network operator containing the configuration data. This code is by nature untrusted and must be manually augmented with various annotations providing hints to the code verifiers, given the specificities of each one. Such annotated extensions can then enter the *x*BGP toolchain which executes in parallel each verifier. The bytecode is produced only if the code passes all of them. Once the bytecode is generated, it is added to the integrated *x*BGP store. A network operator can safely select and load *x*BGP programs coming from this store. We expect that initially each ISP will have its own store. Later, third parties or router vendors could also develop their own stores. We consider this toolchain as trusted, i.e., we select a particular compiler, `clang`, and specific verifiers, all considered as correct. Therefore, we do not need to reason about the produced bytecode and ignore problems such as handling maliciously formatted bytecode.

Embedded verification tools. The whole *x*BGP toolchain, illustrated in Figure 2, is designed to prevent four types of problems that a program can cause. First, if an *x*BGP program enters an infinite loop, it will block the underlying BGP implementation. We use the Terminator 2 (T2) automated termination checker [15] to verify the termination of *x*BGP programs.

The second set of possible problems is the way *x*BGP programs interact with the memory of the underlying BGP implementation. We use CBMC [43] and SeaHorn [31] to verify memory-related properties.

The third type of problem is related to *x*BGP and BGP themselves. *x*BGP programs can create new BGP attributes or messages that are sent over a BGP session. We use SeaHorn to verify that the BGP messages emitted by *x*BGP programs are fully compliant with the BGP RFCs and that their return values respect the *x*BGP requirements.

Finally, operators may want to be able to impose restric-

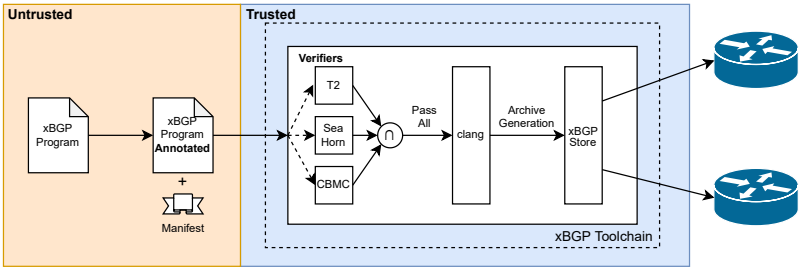


Figure 2: High-level view of the *x*BGP verification toolchain.

Property	Verifier	Type
Termination	T2	Safety
Reads/Writes within *x*BGP program's memory space	CBMC	Safety
No buffer overflow, use after free memory, invalid read, etc.	CBMC	Safety
All strings must be null terminated	SeaHorn	Safety
Correct size/buffer combination	SeaHorn	Safety
RFC-compliant syntax of BGP attributes	SeaHorn	BGP
Valid return value	SeaHorn	Safety
Checking attribute reads/writes	SeaHorn	BGP
Checking API function accesses	`libxbgp`	Safety + BGP
Call the `next()` function to trigger the next *x*BGP program	SeaHorn	Safety

Table 1: Properties that *x*BGP programs must satisfy.

tions on the *x*BGP functions and data structures that a given *x*BGP program can use. For example, a customer filter should only be able to set a `local-pref` value in a chosen range and to change nothing else. So it could never add a new BGP attribute to a route it filters. These restrictions are enforced with (*i*) SeaHorn that checks the validity of the arguments of the API functions and (*ii*) `libxbgp` which restricts the available API functions at loading time.

To be considered valid, any *x*BGP program must satisfy the properties listed in Table 1. If every *x*BGP bytecode satisfies this list, the router is guaranteed (*i*) not to crash and (*ii*) to still follow the definition of the protocol. These properties ensure the local stability of each router. Ensuring the global stability of BGP [28–30, 46] is a problem that goes beyond the scope of this paper.

Verification macros. Because of their diversity, the verification tools do not offer a common way to annotate programs. In the case of *x*BGP, this would mean annotating the plugin 3 times with different annotations and running the 3 tools manually. For a network operator, manually using several tools can be a long, tedious, and error-prone process. To ease the annotation process, we define a set of multipurpose macros `PROOF_INSTS_*()` abstracting the annotation syntax of the verifiers. Those are only expanded if the corresponding verifier is invoked. When the extension programs are compiled

```
buf[0] = attribute->flags;
buf[1] = attribute->code;
buf[2] = attribute->length;
buf[3] = attribute->data;

CHECK_ORIGIN(buf);
```

(a) Annotated Code.

```
assert(buf[0] == ATTR_TRANSITIVE);
assert(buf[1] == ORIGIN_ATTR_ID);
assert(buf[2] == 1);
assert(((buf)[3] == 0 || \
        (buf)[3] == 1 || \
        (buf)[3] == 2)));
```

(b) Expanded Code (verifier).

Figure 3: Example of a verification macro that checks the origin attribute of a BGP route. The macro can be extended or not according to its use. (a) is the original source code and (b) is the code viewed by a verifier.

for routers, the annotations are not expanded and thus will not interfere with the normal BGP execution. Figure 3 shows an example of such verification macro.

Aside from the verifier syntax abstraction, we mainly bring two contributions. First, we define a set of macros helping network operators to verify the properties listed in Table 1. Network operators can use them to annotate their xBGP programs. The macros are translated to their corresponding annotation to the right software verifier. For example, a network operator can use the `BUF_CHECK_*` macros to verify if the BGP attributes sent to a BGP peer are formatted as stated in the RFCs.

Second, we set up a verification toolchain that automatically performs verification on the xBGP programs. It automatically and transparently calls all the verification tools and verifies the annotations contained in the source code of the programs. If all properties are satisfied, the system stores the verified plugins in a "plugin store", which the programmer or network operator can use to inject into their routers. The routers will only accept plugins that have been verified and signed by the plugin store.

Those macros, in conjunction with our verification toolchain, allow a complete abstraction of the verification process. This makes the usage of *x*BGP simpler for network operators. The entire set of verification macros is defined in Appendix B.

3.1 Proving *x*BGP Programs' Termination

T2 (TERMINATOR 2) is a program analysis tool for termination [15] and temporal property [7] verification. We were

successfully able to prove the termination of every *x*BGP program that implements the use cases defined in this paper. Table 3 reports the total time taken by the verification toolchain to verify all the properties defined for the *x*BGP programs, including the termination checks. However, to check the termination we had to slightly modify the source code since some specific features of the C language were not supported by the prover. First, when using fixed-width integer types (e.g., `uint8_t`), T2 was not able to generate the proof of termination. We had to convert those types to their primitive type. Second, all the loops of the program must be explicitly bounded. For example, if the *x*BGP program needs to parse a BGP attribute, we must explicitly bound it to 4096 iterations, the maximum size of a BGP message [54]. Third, T2 does not handle bit shift operations. To solve this issue, we encapsulated the bit shift computation in a non-deterministic function. This is a function that is not defined in the source code of the *x*BGP program but simply tells T2 that it returns an arbitrary integer value that T2 can handle. Such non-deterministic function is also considered to terminate by T2.

3.2 Preventing Memory and C Errors

C is a permissive language and programmers can easily make mistakes in their programs while handling memory. A bug in an *x*BGP program that causes a buffer overflow or leads to using freed memory could crash the underlying BGP implementation. An earlier prototype automatically instrumented the eBPF code to verify these properties online [79]. The verification was made at runtime by adding memory check instructions to the *x*BGP bytecode. However, this had a performance impact. Our new *x*BGP toolchain uses the CBMC bounded model checker [43] to verify the absence of simple memory-related issues and SeaHorn [31] to detect more complex issues. That being said, the online verification of the memory bounds remains in *x*BGP and the operator can enable it or not at *x*BGP bytecode load time.

CBMC is a C Bounded Model Checker that uses loop unwinding methods. It requires that loops are strictly bounded [43] which implies that the code of *x*BGP programs must be adapted. It automatically annotates code, generates a formula and proves it *via* an integrated SAT solver. It can spot common C programming errors [14]. However, more complex properties cannot be checked automatically. For example, *x*BGP programs can log data to syslog. The functions used for that take a string as arguments. Nevertheless, C strings are not safe by design. We must ensure that each string is correctly null-terminated to prevent buffer overflows. Another example is the alteration of BGP attributes. An *x*BGP program needs to call an API function that takes as arguments a pair `<buffer, length>`. Those two values must correlate: if the actual buffer length is shorter than the announced length, the host implementation is vulnerable to a buffer overflow. We use SeaHorn [31] to prove properties written directly in the code as assertions.

We provide a set of C macros that operators call in the *x*BGP programs they verify. The first one is verified by searching for a null byte within the string. For the second one, we verify before each API call if the length passed to the function matches the buffer length. This is achieved by inserting custom annotations in the *x*BGP programs and passing them to SeaHorn.

3.3 Ensuring BGP and *x*BGP Compliance

*x*BGP programs can (i) send new BGP messages or (ii) modify the internal representation of BGP routes. If such a program sends a message deviating from the standardized BGP syntax [54], it could disrupt BGP sessions and have a huge impact [47]. For (i), we verify that the syntax of the BGP message generated by an *x*BGP program conforms to the BGP RFCs. For (ii), we check that the modification is correctly formatted. A corrupted BGP route accessed outside the *x*BGP program could result in a crash of the *x*BGP implementation. For this, the code is annotated with assertions representing BGP invariants that are checked by SeaHorn. We also created a set of C macros verifying that standard attributes comply with their definitions (correct flags, size, etc.). For non-standard attributes, we check that they respect a TLV format. For BGP messages, we check that the buffer containing the message conforms to the BGP syntax [54].

In addition, *x*BGP programs also have to comply to *x*BGP requirements. Some insertion points require a "communication channel" with `libxbgp` to change the behavior of the host BGP implementation. This is achieved by using the return values of the executed bytecode. Therefore, bytecodes cannot deviate from predefined values. For example, an *x*BGP filter returns a specific value to tell the host to reject the current route. This property is verified with SeaHorn by considering the *x*BGP program as a function called inside a "fake" `main`. The return value is then retrieved and verified using a custom assertion.

3.4 Enforcing Operator-Imposed Restrictions

Thanks to the manifest, the operator can list the *x*BGP API functions and the data structures that each *x*BGP program can use. Imagine a filter that only checks the validity of the route without modifying any data related to this route. To decrease the risk of introducing bugs in *x*BGP programs, the operator can restrict the set of API functions the program can call. In this example, the filter should have a read-only view, and thus should not call any function altering BGP data structures.

To settle this, we implemented a permission manager inside `libxbgp` that verifies, at load time, the functions that a given *x*BGP program calls according to its manifest. Just before being loaded, `libxbgp` checks the *x*BGP bytecode to look for unauthorized API function calls.

Network operators use BGP communities [20, 64] to enable their customer to activate specific features such as setting `local-pref`, AS-path prepending, or selective advertisements on a per-route basis. With *x*BGP they could provide even more advanced services. Imagine you are a network provider that proposes to attach filters developed by its clients to their eBGP sessions. You define a set of BGP attributes the client can modify such as MED, `local-pref`, etc. When they use communities, operators establish policies on the attributes which can be modified in a BGP route. For example, they define ranges of possible values for the `local-pref` attribute [20]. To modify an attribute for a route, an *x*BGP program calls the `set_attr` API function. When the *x*BGP toolchain processes such a program, SeaHorn verifies if the arguments of the API functions respect the policies defined in the manifest, i.e., if both the argument to change and its new value are legitimate. This is done by adding assertions in the source code of the *x*BGP program supplied by the customer.

4 Overhead of the current *x*BGP prototype

Using *x*BGP in BGP implementations brings flexibility for a network operator since they can use a simple abstraction to program their router. However, this flexibility has a price in terms of performance. To evaluate the overhead of `libxbgp`, we consider three different features that are already implemented in both native FRRouting and Bird to have a fair comparison with *x*BGP. The first is a simple filter adding an arbitrary MED value to all exported routes. The second provides support for extended communities [59]. The third is a complete implementation of Route Reflection [4]. While we expect operators to mostly develop simple plugins such as the first two, the Route Reflection extension demonstrates the of flexibility *x*BGP by covering the whole BGP workflow described in Section 2. Furthermore, since Route Reflection is supported by both FRRouting and BIRD, this extension enables us to compare the overhead of an *x*BGP implementation with native ones.

To evaluate the performance impact of *x*BGP, we use the simple network described in Figure 4. We measure the delay between the first BGP update sent by the Upstream router and the last update received by the Downstream one. This reflects the time needed for the Device under Test (DuT) router to process the routes sent by the Upstream router. The Upstream and Downstream routers are running an unmodified implementation of BIRD v2.0.8 while the DuT router is running the *x*BGP version of BIRD or FRRouting according to the test. The DuT router is running an Intel® Xeon® X3440 @2.53GHz with 16 GB of RAM, Linux kernel v5.15.29 and Debian 11.

The Upstream router sends a full routing table from a recent RIPE RIS snapshot (June 3, 2021, at 4:15 PM) containing 873k IPv4 routes and 120k IPv6 routes. We consider multiple executions of the BGP daemon located in the DuT router.

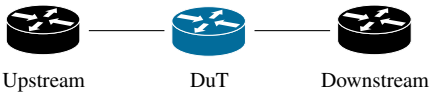


Figure 4: Simple network used for *x*BGP evaluations.

Use Case	Processing Time	
	xFRR	xBIRD
No *x*BGP program	+1.05%	+1.6%
Filter Set MED	+6.67%	+2.59%
Extended Communities	+5.93%	-0.67%
Route Reflection	+12.97%	+7.43%

Table 2: Performance impact of running *x*BGP programs to *x*BIRD and *x*FRR.

Table 2 shows the relative performance impact of running the extensions with *x*BGP programs compared to their native implementation in both BIRD and FRRouting. For each *x*BGP compatible implementation, we run 10 times the *x*BGP programs and compute the convergence time. The convergence time is the time between the first BGP update message is received from Upstream to DuT and the last BGP update message sent from router DuT to Downstream.

Before even loading any *x*BGP extensions, bringing support of *x*BGP in a BGP implementation involves an initial overhead. More specifically, the host implementation must first construct the argument to be passed to the *x*BGP program, then request execution of the corresponding insertion point, and finally execute the *x*BGP termination routine. These additional steps increase the total number of instructions to be executed compared to the native non-*x*BGP implementation. To quantify the cost that `libxbgp` takes in BIRD and FRRouting, we ran both implementations of xBIRD and xFRR without plugins and compared them to their non-*x*BGP compatible versions. Making both implementations of *x*BGP compatible adds a cost in the convergence time of 1% and 1.6% in FRR and BIRD respectively.

We now consider the MED filter (one insertion point) and the extended communities (two insertion points) extensions. When implemented as *x*BGP programs, these slightly increase the convergence time compared to their native version. The Just-In-Time compiler used inside the virtual machine does not optimize as efficiently as the one producing native code. In particular, computation-intensive bytecode involving additions, subtractions, and multiplications take 50% more time to run than native code. This overhead is even worse when considering division and modulo operations.

Yet, we observe a higher convergence time increase for FRRouting than BIRD. By analyzing the execution of each *x*BGP bytecode with a code profiler, we identified two main reasons for this difference. First, to communicate with the host implementation, the *x*BGP program must pass through a dedicated *x*BGP API. For security reasons and because of

the internal mechanism of `libxbgp`, the data of the host implementation are first translated into a neutral representation, then copied into a dedicated memory area, accessible in writing and reading by the bytecode. Translation and copying play an important role in the execution of a plugin but are needed to run the same *x*BGP program in several BGP implementation. BIRD internally uses data structures that are closer to the *x*BGP neutral representation than the FRRouting ones, hence involving less translation overhead. Second, FRRouting and BIRD have different internal architectures. The interactions between the `libxbgp` API and the BGP implementations are different. FRRouting is less flexible as its implementation is not designed to be quickly extended with new functionalities. While in BIRD, most of the insertion points map to a specific place, in FRRouting some insertions points must be repeated at different code places, involving up to four times more *x*BGP program calls than BIRD.

We now consider the Route Reflection extension covering the whole BGP workflow. Supporting this feature requires a list of all iBGP client peers. Routers' implementations use their dedicated CLI syntax to define all their iBGP client peers. `libxbgp` does not have access to this CLI configuration since it is implementation-dependent. Instead, it relies on its configuration data within the manifest that can be accessed at any time by the *x*BGP program. On average, the BIRD's convergence time is 7.5% slower than the native code while FRRouting's one is 13% slower. The previous elements still hold to explain the difference between BIRD and FRRouting. In particular, there are more calls to *x*BGP programs in FRRouting due to its code architecture than in BIRD (`BGP_ENCODE_MESSAGE` is called 4 times more), and the translation time to convert data structures is non-negligible in FRRouting (up to 40% overhead for the import filter). Still, the performance overhead of *x*BGP remains in acceptable bounds.

5 Use Cases

Section 2 presented the *GeoTLV* feature to demonstrate that *x*BGP programs can create new attributes that influences the router. Section 4 presented the MED filter, extended communities, and route reflectors to make a performance comparison. This section presents other use cases, which are not implemented natively in FRRouting and BIRD, that illustrate the advantages of *x*BGP for various classes of problems that the operator wants to solve. It is true that the features of this section can be implemented in any BGP implementation without *x*BGP. However, feature support depends on the pace of implementation by all vendors. Thanks to the *x*BGP design, an operator can quickly design its features and introduce them into the network before they are implemented by the vendor. xBGP is the first step to bring extensibility to the network. The first use case defines an *x*BGP program (Section 5.1) to influence the decision process and the import and export filters from the BGP client point of view. The second use case detects zombie routes (Section 5.2). These are routes that are installed in the routing table but are no longer reachable. Third, operators always try to understand the state of their network to improve it as much as possible. We present two use cases (Section 5.3 and 5.4), where BGP is monitored using communities. Due to space limitations, we detail three other extensions in appendix. The fifth use case is related to route filtering in data-centers (Section A.1). It demonstrates that *x*BGP can provide a programmable interface to design complex import and export filters. Our sixth *x*BGP program (Section A.2) gives another example of a special filter that checks the origin of a route. Finally, our seventh use case (Section A.3) shows that an *x*BGP compatible implementation can leverage IGP information to make routing decisions.

Use Case	C LoC	eBPF Insts	Total Verif Time(s)
Geo TLV (§2)	388	1340	664
MED Filter (§4)	55	149	79
Extended Communities (§4)	196	322	86
Route Reflection (§4)	509	3853	27
Route Selection (§5.1)	62	148	27
Zombie Detection (§5.2)	1071	5697	277
Decision Monitor (§5.3)	306	437	29
Propagation Time (§5.4)	560	805	73
Valley Free (§A.1)	143	960	182
Prefix Origin (§A.2)	150	661	57
IGP Data (§A.3)	36	149	3

Table 3: Verification of the *x*BGP programs supporting our use cases.

Table 3 reports the size of the *x*BGP bytecode, the number of lines of code and the time taken to validate every *x*BGP program according to the properties defined in Section 3.

5.1 Customer Selecting Routes

A BGP router only selects one route for each prefix even though it learns multiple routes. As a result, it will only send one route to each BGP neighbor, which decreases the path diversity. Consider Figure 5 to illustrate the situation. AS1, a multihomed stub network having peering links with `Transit 1` and `AS2`. We are interested in the propagation of the routes to the destination network depicted in gray. To maximize path diversity in AS1, it should learn the purple path from AS2 to leverage the two different transits. However, AS1 cannot influence the decision process of AS2's routers.

Enabling the dissemination of multiple routes can bring several benefits such as load-balancing [45], avoiding route oscillation [29] and faster local recovery upon a network failure [61]. With *x*BGP it becomes possible to influence the border router to announce the route the client prefers. To design such an *x*BGP program, all edge routers must enclose their BGP client to one Virtual Routing and Forwarding table

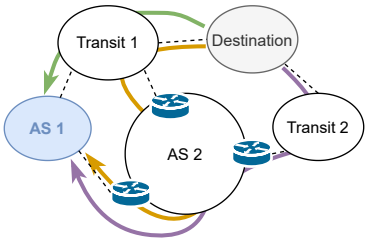


Figure 5: Path Diversity in a Network.

(VRF) [70]. All the routes learned from all neighbors will be exported to the main BGP-VPN RIB's router and then exported to the VRF of each client so that they can have a full view of the routes. Since all clients are in their respective VRFs, the BGP decision process is different for each of them and can therefore be influenced by an *x*BGP program that decides which route to advertise.

We designed a simple *x*BGP program that randomly selects one of the available routes in the VPN RIB thanks to the *x*BGP API function `get_vrf`. It demonstrates that *x*BGP allows the operator to create a customized and more powerful route selection compared to the traditional router CLI. Accessing the VPN RIB through a simple router configuration is something that cannot be done with traditional BGP implementation. Furthermore, an *x*BGP program has access to the entire BGP route and the internal data structure of the BGP router. *x*BGP therefore provides greater flexibility compared to the classical CLI.

We were successfully able to check the termination with T2, the C errors with CBMC, its compliance to the *x*BGP return values. We also verified that the program does not use API functions altering the BGP internal state.

5.2 Detecting BGP Zombies

When a route becomes unavailable, a BGP router sends a withdraw message to all its peers. Because of software bugs [22], it may happen that one of these BGP peers fails to process such a withdraw message. As a result, the route is still considered reachable by the failed router. This is an operational problem because the withdrawal is not propagated, and part of the network still believes that the route is available. If packets still follow this zombie route, they will be black holed. Measurements indicate that these zombie routes are common and affect many ASes [50].

To detect zombie routes, we designed an *x*BGP program that is executed periodically. It uses the timestamp of the arrival of a route in the RIB to detect the routes that are older than x days. Our threshold is arbitrarily fixed to a day. Our *x*BGP program is configured to be executed during the maintenance window. It parses the entire BGP RIB thanks to the API functions `*_rib_iterator`. If a route is older than the configured threshold, it is flagged as a possible zombie. To confirm the status of the route, the router needs to request it again from the peer that announced it. This could be done with standardized mechanisms such as Graceful Restart [58] or Route Refresh [12, 51]. However, those two approaches require the remote router to announce again its entire BGP routing table. For the sake of performance, we decided to only ask the remote peer to reannounce the routes flagged by the *x*BGP program. We introduce a new type of BGP message called `BGP Refresh`. It contains a list of prefixes that the router wants to confirm. The peer receiving the `BGP Refresh` message will announce a withdraw or an update message if the routes are not available anymore or still in its BGP routing table respectively. *x*BGP allows sending BGP messages via the `schedule_bgp_message` API function.

It is difficult with a traditional BGP implementation to detect such a zombie route. Indeed, there is no mechanism to analyze and perform an action according to the state of the BGP routing table. To include this feature, the network operator must convince each router vendor to add this feature into its implementation. This use case demonstrates that *x*BGP can outperform the current configuration method that is proposed in classical BGP implementation.

This *x*BGP program successfully passes the T2 and CBMC verifications. As it manages BGP messages, we verified their compliance to the RFC. We also checked that the size of the buffers announced to the API function matches their real size. This program *x*BGP is an example of functionality that cannot be performed with the traditional router CLI while the router is running.

5.3 Monitoring the BGP Routing Decision

Currently, if a network operator would like to debug its BGP routers, he only has monitoring information from the routers it directly controls. This is due to the fact that the traditional BGP specification only provides the exchange of local routing information but does not provide any abstraction to send monitoring information about the routing process. Yet, a support of a dedicated monitoring channel has been proposed [60] but this is still not implemented on all vendor's routers. In a nutshell, a BGP router can ask its neighbor to give different metrics such as its number of reachable prefixes, its ADJ-RIB-IN, its current state, etc. This shows that many network operators need to monitor the BGP session to enable better control of routing information in the network. Some router vendors actually provide commands to retrieve the local state of a router. However, the information is restricted to the router view only and does not include the status of routers that are outside the operator's management scope. Having statistics from other routers could bring many benefits such as the selection of a better route. More specifically, if the BGP router sends its best routes with the step at which routes have been decided, the remote BGP router can learn much information about the route diversity in a network. If the routes are al-

ways decided at the very end of the BGP decision process, it indicates a lack of diversity in the remote network. On the contrary, if the routes are decided early in the process such when the AS_PATH is shorter than the previous best route, it can indicate a higher level of diversity. Thanks to this information, the BGP router can adapt its behavior to prefer a path with more diversity to be more resilient to a router or link failure.

We leverage *x*BGP to instrument the BGP implementation to retrieve at which step of the BGP decision process the route has been chosen. Each time it runs, an *x*BGP program retrieves the reason of the decision in the process. It can be retrieved through the arguments passed to the *x*BGP program. To inform other BGP routers, this information is added as a BGP community when sent to other BGP speakers. This is done by the API function `set_attr_to_route`. This way, other routers can parse and use this information to adapt their routing strategies. This *x*BGP program also collects statistics about the other steps of the BGP decision. Each time a route is selected, the *x*BGP program increments an internal counter. It repeats the operation for each decision step. When a route is sent to any peer, these statistics are attached as a community. The BGP router receiving the statistics can have a broader view of the current routing table of its peer. If all the routes have been decided by the BGP tiebreaker that compares the IP address of the router that sends the route then it shows a lack of path diversity. The network operator could then attribute a lower preference to the route advertised by the remote router.

This *x*BGP program successfully passes all the verifiers. Since it handles the BGP community attribute, we verified if the format is respected according to the corresponding RFC.

Since this information cannot be retrieved with traditional router CLI, this new approach could enable more fine-grained routing decisions. Indeed, this new type of active monitoring cannot be achieved with traditional monitoring tools such as BMP, SNMP, etc. as these later tools do not modify the BGP message they sent to the BGP neighbor. Network operators have thus at their disposal new information from outside their network.

5.4 Measuring BGP Route Propagation Times

For mission critical systems, the convergence time of a routing protocol is an important metric to know. It helps to better understand what could be the cause of a slow convergence. Discussions with network operators indicated that commercial router vendors provide undocumented CLI commands to access profiling points. However, this profiling information is local to each router. It could be useful to exchange such information within an entire network. This could open new opportunities to better understand the current state of the network. One example of such monitoring is the time taken by a BGP route to traverse an AS. To support such monitoring information, BGP must be augmented to add in each route its arrival time at each AS border router. Our *x*BGP program defines a new non-transitive BGP attribute, called `RECEIVED_TIME`. It adds this attribute when a route is received over an eBGP session (thanks to the `set_attr` *x*BGP API function family). It traverses the AS with the BGP route until it reaches an edge router. The `RECEIVED_TIME` attribute is removed when the associated route is sent over an eBGP session and the border router computes the difference between its current NTP time and the one of the attribute. As for the previous use case, exchanging such monitoring information is not currently feasible with traditional routers. These two use cases show that *x*BGP can perform a new type of active monitoring by exposing the internal data of the BGP implementation itself to inform the other neighbor of the current BGP routing state.

6 Related Work

Protocol programmability. In the late nineties active networks were proposed as a solution to bring innovation back inside the network that was perceived as being ossified [65]. Most of the work in this area focused on the possibility of placing bytecode inside network layer packets. PLAN [66], ANTS [73] and router plugins [19] are examples. In the control plane, researchers built upon this idea to propose new solutions such as the 4D architecture [25], the Routing Control Platform that centralizes routing [11] or Metarouting [27] that proposed to open the definition of routing protocols using a declarative language. While these previous works propose configuration languages or centralized approaches to deal with network programmability, *x*BGP relies on an existing decentralized control plane protocol on which an operator can add its new functionality to locally influence the routing.

Bringing flexibility to an implementation of a network protocol has been studied in the literature. Researchers have proposed using extension codes to extend transport protocols like STP [52], QUIC [18] and the FRRouting implementation of OSPF and BGP [80]. However, the architecture of these pluginized approaches is close to the internal architecture of a single protocol implementation and does not offer the flexibility to pluginize different implementations of the same protocol. *x*BGP goes one important step further by enabling very different implementations to execute the same *x*BGP program. *x*BGP tries to determine what all implementations of a protocol have in common to try to find a common usable interface.

To ease the automation and the configuration of their devices, routers vendors added scripting languages that enable the network operator to execute recurrent tasks [6]. However, this acts as a simple shell that cannot be used to extend the router implementation. Other vendors integrated the python language into their router OS [40] to perform automation task more easily, such as configuring the router or executing a monitoring routine when a particular event occurs.

Reducing the BGP implementation to its minimum has been studied with CoreBGP [74]. However, it only manages the basic BGP Finite State Machine on which plugins written in the Go language are inserted. The remainder of the BGP logic such as sending BGP messages or managing the routing table is passed to the plugins. CoreBGP plugins react to an FSM events while *x*BGP programs react to protocol events defined by the insertion points depicted in Figure 1.

XORP [33, 34] was introduced to propose an open-source software router platform. This solution has been designed to allow researchers to easily develop their own extensions to a routing protocol. Other open-source routing stacks have been developed such as Quagga [2], FRRouting [23] or BIRD [16]. While these open-source stacks allow modifying the source code of the routing software, *x*BGP goes one step further by introducing a simple API to interact with the routing software. There is no need to look directly in the code of the implementation to understand how to integrate an extension. Anyone who wants to add their own extension will interact with the router through *x*BGP. Throughout this paper, we demonstrated that an *x*BGP extension code written only once can be successfully executed by two open-source routing stacks, FRRouting and BIRD.

Virtual Machines. `libxbgp` is based on a user-space implementation of the kernel eBPF VM [53]. In recent years, Linux kernel developers have integrated a virtual machine called eBPF [63] which enables programs to inject executable bytecode at specific locations inside the kernel. It was initially targeted at monitoring kernel operations [35], but also for fast packet processing [35]. Researchers have used eBPF to support networking programming with IPv6 Segment Routing [83] and extend TCP [68]. Other frameworks could have been used such as WebAssembly [32] or lua [38] that is widely used in industrial systems. Using another type of VM can be studied to measure its performance and its relevance to routing protocols.

Verification tools. The PDS (Plugin Distribution System) [57] provides secure verification and distribution of extension code for Pluginized QUIC [18]. It allows the automation of different types of verification for several extension codes at the same time. Our *x*BGP toolchain includes more verifiers and checks more properties. While the PDS uses a Merkel tree to secure the distribution of plugins, the *x*BGP toolchain simply keeps them in a store that is used by the network operator.

7 Conclusion

We presented *x*BGP, a new paradigm that enables network operators to innovate in routing protocols. *x*BGP allows them to write their extensions or modifications in the form of an *x*BGP program that can be executed inside the protocol implementation. This programmability could help network operators innovate with existing distributed routing protocols as Software Defined Networking lead to the development of programmable switches. Our solution has been proposed for BGP but could also be adapted to support other routing protocols. We further introduced the *x*BGP toolchain that allows operators to annotate *x*BGP programs to verify their safety. It checks if the *x*BGP program meets the local properties of the router such as the termination, the memory constraints and if the *x*BGP program meets the definition of BGP. If it passes the verification step, the *x*BGP program can be safely added to the BGP implementation and is guaranteed not to corrupt the router. Finally, we demonstrated *x*BGP's capabilities by proposing several use cases that have been implemented with our solution. Among them, *x*BGP enables the operator to add new attributes to a BGP route, implementing complex filters, allowing a client to influence the BGP decision process and executing background tasks.

Future Directions. We see two directions to improve *x*BGP. The first would be to look at how to structure an existing BGP implementation to support *x*BGP more efficiently. The second is related to the virtual machine used. eBPF was the most mature virtual machine during the development of *x*BGP. However, other virtual machines such as WebAssembly seem more promising and start to perform well. It might be interesting to see the advantages of using them in the context of *x*BGP.

Software artifacts

To encourage other researchers to reproduce and extend our results we provide the entire source code of `libxbgp` [78] composed of 3506 LoC, the eBPF virtual machine we use (2236 LoC), the two versions of FRRouting [77] (+2675 LoC) and BIRD [76] (+2083 LoC) *x*BGP compatible, the whole *x*BGP programs (15 programs) we developed on top of *x*BGP [81], the experimental scripts we use to evaluate the impact of the performance with our approach (853 LoC) [78] and our verification toolchain based on the PDS [56]. We will also provide the set of annotation to verify *x*BGP programs (1121 LoC) [82].

Acknowledgments

This work has been partially supported by the French Community of Belgium through the funding of a FRIA (Fund for Research training in Industry and Agriculture) grant. We thank the anonymous reviewers and our shepherd, Phillipa Gill, whose helpful comments improved the quality of this paper.

References

[1] The linux kernel static checker. https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c.

[2] Quagga software routing suite. https://www.nongnu.org/quagga/.

[3] "Microsoft Azure". Software for open networking in the cloud. https://azure.github.io/SONiC/.

[4] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456 (Draft Standard), April 2006. Updated by RFC 7606.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–451, 2017.

[6] Raymond Blair, Arvind Durai, and John Lautmann. *Tcl scripting for Cisco IOS*. Cisco Press, 2010.

[7] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 387–393. Springer, 2016.

[8] R. Bush and R. Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol. RFC 6810 (Proposed Standard), January 2013.

[9] R. Bush, K. Patel, and D. Ward. Extended Message Support for BGP. RFC 8654 (Proposed Standard), October 2019.

[10] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28. USENIX Association, 2005.

[11] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28, 2005.

[12] E. Chen. Route Refresh Capability for BGP-4. RFC 2918 (Proposed Standard), September 2000. Updated by RFC 7313.

[13] Enke Chen, Naiming Shen, and Robert Raszuk. Carrying Geo Coordinates in BGP. Internet-Draft draft-chen-idr-geo-coordinates-02, Internet Engineering Task Force, October 2016. Work in Progress.

[14] Edmund Clarke and Daniel Kroening. Ansi-c bounded model checker user manual. *School of Computer Science, Carnegie Mellon University*, 2006.

[15] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: beyond safety. In *International Conference on Computer Aided Verification*, pages 415–418. Springer, 2006.

[16] CZ.NIC, z.s.p.o. BIRD internet routing daemon. https://gitlab.nic.cz/labs/bird.

[17] Guy Davies. *Designing and Developing Scalable IP Networks*. John Wiley & Sons, 2004.

[18] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 59–74. 2019.

[19] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. *SIGCOMM Comput. Commun. Rev.*, 28(4):229–240, October 1998.

[20] Benoit Donnet and Olivier Bonaventure. On BGP communities. *ACM SIGCOMM Computer Communication Review*, 38(2):55–59, 2008.

[21] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[22] Romain Fontugne, Esteban Bautista, Colin Petrie, Yutaro Nomura, Patrice Abry, Paulo Gonçalves, Kensuke Fukuda, and Emile Aben. BGP zombies: An analysis of beacons stuck routes. In *International Conference on Passive and Active Network Measurement*, pages 197–209. Springer, 2019.

[23] The Linux Foundation. FRRouting project. https://frrouting.org/.

[24] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084, 2019.

[25] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[26] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[27] Timothy G. Griffin and Joäo Luís Sobrinho. Metarouting. *SIGCOMM Comput. Commun. Rev.*, 35(4):1–12, August 2005.

[28] Timothy G Griffin and Gordon Wilfong. An analysis of bgp convergence properties. *ACM SIGCOMM Computer Communication Review*, 29(4):277–288, 1999.

[29] Timothy G Griffin and Gordon Wilfong. Analysis of the med oscillation problem in bgp. In *10th IEEE International Conference on Network Protocols, 2002. Proceedings.*, pages 90–99. IEEE, 2002.

[30] Timothy G Griffin and Gordon Wilfong. On the correctness of ibgp configuration. *ACM SIGCOMM Computer Communication Review*, 32(4):17–29, 2002.

[31] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[32] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[33] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: An open platform for network research. *SIGCOMM Comput. Commun. Rev.*, 33(1):53–57, jan 2003.

[34] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible ip router software. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 189–202, 2005.

[35] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress Data Path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[37] G. Huston and G. Michaelson. Validation of Route Origination Using the Resource Certificate Public Key Infrastructure (PKI) and Route Origin Authorizations (ROAs). RFC 6483 (Informational), February 2012.

[38] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[40] Junos. Junos PyEZ developer guide. https://www.juniper.net/documentation/en_US/junos-pyez/information-products/pathway-pages/junos-pyez-developer-guide.html, July 2021.

[41] K. Kompella, B. Kothari, and R. Cherukuri. Layer 2 Virtual Private Networks Using BGP for Auto-Discovery and Signaling. RFC 6624 (Informational), May 2012.

[42] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[43] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[44] P. Lapukhov, A. Premji, and J. Mitchell (Ed.). Use of BGP for Routing in Large-Scale Data Centers. RFC 7938 (Informational), August 2016.

[45] Petr Lapukhov and Jeff Tantsura. Equal-Cost Multipath Considerations for BGP. Internet-Draft draft-lapukhov-bgp-ecmp-considerations-07, Internet Engineering Task Force, June 2021. Work in Progress.

[46] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[47] Robert Mc Millan. Research experiment disrupts internet, for some. *Computerworld*, pages August, 28, 2010. https://www.computerworld.com/article/2515036/research-experiment-disrupts-internet--for-some.html.

[48] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[49] Microsoft Corporation. ebpf for windows. https://github.com/microsoft/ebpf-for-windows.

[50] Porapat Ongkanchana, Romain Fontugne, Hiroshi Esaki, Job Snijders, and Emile Aben. Hunting BGP zombies in the wild. In *Proceedings of the Applied Networking Research Workshop*, ANRW '21, page 1–7, New York, NY, USA, 2021. Association for Computing Machinery.

[51] K. Patel, E. Chen, and B. Venkatachalapathy. Enhanced Route Refresh Capability for BGP-4. RFC 7313 (Proposed Standard), July 2014.

[52] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. Upgrading Transport Protocols using Untrusted Mobile Code. *ACM SIGOPS Operating Systems Review*, 37(5):1–14, 2003.

[53] IO Visor Project. Userspace eBPF VM. https://github.com/iovisor/ubpf.

[54] Y. Rekhter (Ed.), T. Li (Ed.), and S. Hares (Ed.). A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006.

[55] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks (VPNs). RFC 4364 (Proposed Standard), February 2006. Updated by RFCs 4577, 4684, 5462.

[56] Nicolas Rybowski. Plugin Distribution System. https://github.com/nrybowski/SPMS.

[57] Nicolas Rybowski, Quentin De Coninck, Tom Rousseaux, Axel Legay, and Olivier Bonaventure. Implementing the plugin distribution system. In *Proceedings of the SIGCOMM '21 Poster and Demo Sessions*, page 39–41. Association for Computing Machinery, New York, NY, USA, 2021.

[58] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. Graceful Restart Mechanism for BGP. RFC 4724 (Proposed Standard), January 2007. Updated by RFC 8538.

[59] S. Sangli, D. Tappan, and Y. Rekhter. BGP Extended Communities Attribute. RFC 4360 (Proposed Standard), February 2006. Updated by RFCs 7153, 7606.

[60] Rob Shakir, Robert Raszuk, Rob Shakir, and David Freedman. BGP OPERATIONAL Message . Internet-Draft draft-frs-bgp-operational-message-00, Internet Engineering Task Force, July 2011. Work in Progress.

[61] M. Shand and S. Bryant. IP Fast Reroute Framework. RFC 5714 (Informational), January 2010.

[62] Rachee Singh, Muqeet Mukhtar, Ashay Krishna, Aniruddha Parkhi, Jitendra Padhye, and David Maltz. Surviving switch failures in cloud datacenters. *ACM SIGCOMM Computer Communication Review*, 51(2):2–9, 2021.

[63] A Starovoitov. BPF-in-kernel virtual machine. *Linux Kernel Developers' Netconf*, 2015.

[64] Florian Streibelt, Franziska Lichtblau, Robert Beverly, Anja Feldmann, Cristel Pelsser, Georgios Smaragdakis, and Randy Bush. BGP communities: Even more worms in the routing can. In *Proceedings of the Internet Measurement Conference 2018*, pages 279–292, 2018.

[65] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, October 2007.

[66] D.L. Tennenhouse and D.J. Wetherall. Towards an active network architecture. In *Proceedings DARPA Active Networks Conference and Exposition*, pages 2–15, 2002.

[67] The OpenBSD Project. Openbgpd. http://openbgpd.com/.

[68] Viet-Hoang Tran and Olivier Bonaventure. Beyond socket options: making the linux TCP stack truly extensible. In *2019 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2019.

[69] Yves Vanaubel, Jean-Jacques Pansiot, Pascal Mérindol, and Benoit Donnet. Network fingerprinting: Ttl-based router signatures. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 369–376, 2013.

[70] Laurent Vanbever. Customized BGP route selection using BGP/MPLS VPNs. In *Routing Symposium, Cisco Systems*, 2009.

[71] Stefano Vissicchio, Luca Cittadini, and Giuseppe Di Battista. On iBGP routing policies. *IEEE/ACM Transactions on Networking*, 23(1):227–240, 2014.

[72] Matthias Wählisch, Fabian Holler, Thomas C Schmidt, and Jochen H Schiller. Rtrlib: An open-source library in c for rpki-based prefix origin validation. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*, 2013.

[73] D.J. Wetherall, J.V. Guttag, and D.L. Tennenhouse. Ants: a toolkit for building and dynamically deploying network protocols. In *1998 IEEE Open Architectures and Network Programming*, pages 117–129, 1998.

[74] Jordan Whited. Corebgp - plugging in to bgp. https://github.com/jwhited/corebgp, July 2020.

[75] Thomas Wirtgen. xBGP api documentation. https://github.com/pluginized-protocols/xbgp_plugins/blob/master/xbgp_compliant_api/xbgp_plugin_api.h.

[76] Thomas Wirtgen. xBGP bird. https://github.com/pluginized-protocols/xbgp_bird.

[77] Thomas Wirtgen. xBGP frrouting. https://github.com/pluginized-protocols/xbgp_frr.

[78] Thomas Wirtgen. xBGP source code. https://github.com/pluginized-protocols/libxbgp.

[79] Thomas Wirtgen, Quentin De Coninck, Randy Bush, Laurent Vanbever, and Olivier Bonaventure. xBGP: When you can't wait for the ietf and vendors. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 1–7, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Thomas Wirtgen, Cyril Dénos, Quentin De Coninck, Mathieu Jadin, and Olivier Bonaventure. The case for pluginized routing protocols. In *27th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2019.

[81] Thomas Wirtgen and Tom Rousseaux. xBGP plugins source code. https://github.com/pluginized-protocols/xbgp_plugins.

[82] Thomas Wirtgen and Tom Rousseaux. xBGP verification. https://github.com/pluginized-protocols/xbgp_plugins/tree/master/prove_stuffs.

[83] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. Leveraging eBPF for programmable network functions with ipv6 segment routing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 67–72, New York, NY, USA, 2018. Association for Computing Machinery.

A Additional use cases

This section describes some additional use cases for *x*BGP.

A.1 BGP in data centers

Although BGP was designed as an interdomain routing protocol, it is now widely used as an intradomain routing protocol in data centers [44]. This is mainly because BGP scales better since it does not rely on flooding in contrast with OSPF or IS-IS. Another benefit of BGP is its ability to support a wide range of configuration knobs and policies. However, BGP suffers from several problems that forces the network operators to tweak their BGP configurations [44]. These tweaks make BGP configurations complex and more difficult to analyze and validate [5]. To illustrate this complexity, let us consider the data center shown in Fig. 6. Routers $S1$ and $S2$ are the Spine routers, $L10 \ldots L13$ the leaf routers, and $T20 \ldots$ the top-of-the rack routers. In such a data center, there is no direct connection between the routers at the same level in the hierarchy. Data center operators usually want to avoid paths that include a valley (e.g. $L10 \rightarrow S1 \rightarrow L11 \rightarrow S2$). To achieve this, they usually run eBGP between routers, but configure the same AS number on $S1$ and $S2$ (even if these routers are not connected). Similarly, $L10$ and $L11$ (resp. $L12$ and $L13$) use the same AS number. With this configuration, when $S2$ receives a BGP update with an AS-Path through $S1$, it recognizes its AS number and rejects the route. This automatically blocks paths that include a valley and also helps to prevent path hunting.

Unfortunately, using the same AS number on separate routers can cause problems. First, operators can no longer

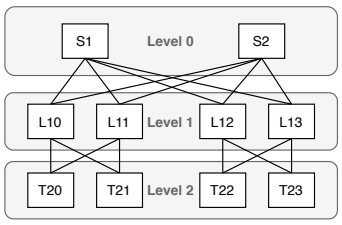


Figure 6: A simple data center.

look at the AS Paths to troubleshoot routing problems since different routers use the same AS number. Second, by prohibiting valley-free paths, the operator implicitly agrees to partition the network when multiple failures occur. Consider again Figure 6. If both links $L10-S1$ and $L13-S2$ fail, then the only possible path between $L10$ and $L13$ is $L10 \rightarrow S2 \rightarrow L12 \rightarrow S1 \rightarrow L13$. If the same AS number is used on $S1$ and $S2$, this path will never be advertised.

With *x*BGP, a network operator can use different AS numbers for their routers and implement specialized filters on the spine and leaf routers. For example, if $S1$ and $S2$ are both connected to transit providers and can reach the same prefixes, then $L10$ should never reach $S2$ via $S1$ and $L11$. However, this path should remain valid if the final destination is a prefix attached to below $R13$.

To implement such a filter, we load a manifest containing every eBGP session from a router of level i to a router of level $i+1$ in a pair having the following form: $(AS_{li}, AS_{l(i+1)})$. For each route, the filter checks each consecutive pair of the AS-Path. If a pair of this manifest is included in the AS-Path, the filter rejects the route since it is not valley-free.

This *x*BGP program successfully passes T2 and CBMC checks. SeaHorn confirms its *x*BGP compliance relative to its use of the API functions and the return values.

A.2 Validating BGP Prefix Origins

The interdomain routing system is regularly affected by disruptions caused by invalid BGP advertisements originated from ISPs. Examples include the AS7007 incident in 1997, the announcement of a more specific prefix covering the YouTube DNS servers by Pakistan Telecom in 2008, or BGP prefixes leaked by Google in 2017 that disrupted connectivity in parts of Asia. These problems and many similar ones were caused by configuration errors.

To illustrate the flexibility of *x*BGP, we consider a RPKI-based route origin [37] validation variant. The network operator includes in the configuration data of the manifest all prefixes it knows the origin. We assume the operator has themself validated the ROA signatures before generating the file. This file is used by the *x*BGP program each time a BGP route is received by a peer to check if the origin AS of the route matches with the one contained in the file.

To evaluate the performance of our prefix origin validation, we use the same testbed as in Section 4 excepted that we use eBGP sessions for links L1 and L2. Our *DuT* does not implement the RPKI-Rtr protocol [8, 72] but loads configuration data that considers 75% of the injected prefixes as valid. For this test, our extension code checks the validity of the origin of each prefix but does not discard the invalid ones.

Table 2 compares our extension codes running on BIRD and FRRouting to their native implementations without any prefix validation. We do not compare our solution with the RPKI-Rtr protocol since we do not totally implement the RPKI protocol. We only check the origin of the route. The difference in execution between the two implementations is also explained by the difference in the internal representation of the data structures used.

The termination and absence of C errors were proved with T2 and CBMC. SeaHorn also confirms that the *x*BGP program does not write any data in the memory of the host implementation and its compliance on the return values.

A.3 Filtering Routes Based on IGP Costs

Since the *x*BGP API provides access to the data structures maintained by a BGP implementation, network operators can leverage it to implement new filters. As a simple example, consider an ISP having a worldwide presence that wants to announce to its peers the routes that it learned in the same continent as the advertising BGP. This policy can be implemented by tagging routes with BGP communities on all ingress routers and then filtering them on export. While being frequently used [20], this solution is imperfect. Consider an ISP having two transatlantic links terminated in London, UK, and Amsterdam in The Netherlands. This ISP has a strong presence in Europe and two links connect the UK to other European countries. If these two links fail, packets between Germany and London will need to go through Amsterdam, the USA, and then back to the UK. When such a failure occurs, the ISP does not want to advertise the routes learned in the UK to its European peers. With BGP communities, it would continue to advertise these routes after the failure.

Using the *x*BGP API, the operator could implement this policy as follows. First, he configures the IGP cost of the transatlantic links at a high value, say 1000 to discourage their utilization. Second, he implements a simple export filter that checks the IGP cost of the next-hop before announcing a route. The complete source code of such a filter is shown in Listing 1. It is attached to the `BGP_OUTBOUND_FILTER` ④ insertion point. If the IGP cost to the BGP next hop distance is acceptable, the function calls the special function `next()`. This informs the VMM to execute the next bytecode attached to the insertion point. If the extension code is the last to be executed, the insertion point proposes to fall back to the native code. To reject the route, the extension code returns the special value `FILTER_REJECT` to the host implementation.

For this *x*BGP program, we used SeaHorn to ensure return

Flag	Enables
PROVERS	Verification macros.
PROVERS_ARGS	next() call verification macros.
PROVERS_T2	T2 related macros.
PROVERS_CBMC	CBMC related macros.
PROVERS_SEAHORN	SeaHorn related macros.

Table 4: Verification flags.

Macro	Code only provided to
PROOF_INSTS_SEAHORN	SeaHorn.
PROOF_INSTS_CBMC	CBMC.
PROOF_INSTS_T2	T2.

Table 5: Macro allowing to provide pieces of code for a specific verifier.

values were meaningful to `libxbgp`. T2 and CBMC are also used to check the termination and the absence of any C errors. We also verify that the *x*BGP program has only a read-access to the host implementation.

```
uint64_t export_igp(bpf_full_args_t *args UNUSED) {
  struct ubpf_nexthop *nexthop = get_nexthop(NULL);
  struct ubpf_peer_info *peer = get_peer_info();
  if (peer->peer_type != EBGP_SESSION) {
    next(); // Do not filter on iBGP sessions
  } if (nexthop->igp_metric <= MAX_METRIC) {
    next(); // the route is accepted by this filter;
  }         // next filter will decide to export route
  return FILTER_REJECT;
}
```

Listing 1: An export filter rejecting BGP routes having a too large IGP nexthop metric.

B Verification macros

This appendix provides in Tables 5, 6, and 7 exhaustive lists of our custom-made verification macros. Those are enabled at compile time with different flags described in Table 4.

Macro prefix	Attribute name/macro suffix	Check
BUF_CHECK_*	LENGTH ORIGIN ASPATH NEXTHOP MED LOCAL_PREF ATOMIC_AGGR AGGREGATOR COMMUNITY ORIGINATOR CLUSTER_LIST EXTENDED_COMMUNITIES AS4_PATH AS4_AGGREGATOR AIGP LARGE_COMMUNITY	The correct formatting of the attribute which is stored in a buffer.
CHECK_*	LENGTH ORIGIN ASPATH NEXTHOP MED LOCAL_PREF ATOMIC_AGGR AGGREGATOR COMMUNITY ORIGINATOR CLUSTER_LIST EXTENDED_COMMUNITIES AS4_PATH AS4_AGGREGATOR AIGP LARGE_COMMUNITY	The correct formatting of the attribute which is stored in a `path_attribute` structure.
CHECK_IN_BOUNDS_*	LOCAL_PREF MED	The given attribute lies in the range specified by the operator.
CHECK_*	ARG ARG_CODE OUT RET_VAL_FILTER	The next() function is called if the xBGP program cannot parse the current attribute.

Table 6: BGP attributes verification macros used by SeaHorn.

Macro prefix	Target	Check
CHECK_*	BUFFER	The given buffer respects the specified size.
	STRING	The given string is null-byte terminated.
	COPY	The copied buffer is unchanged.

Table 7: Memory check macros used by SeaHorn.

TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches

Aashaka Shah*
University of Texas at Austin

Vijay Chidambaram
University of Texas at Austin and VMware Research

Meghan Cowan
Microsoft Research

Saeed Maleki
Microsoft Research

Madan Musuvathi
Microsoft Research

Todd Mytkowicz
Microsoft Research

Jacob Nelson
Microsoft Research

Olli Saarikivi
Microsoft Research

Rachee Singh
Microsoft and Cornell University

Abstract

Machine learning models are increasingly being trained across multiple GPUs and servers. In this setting, data is transferred between GPUs using communication collectives such as ALLTOALL and ALLREDUCE, which can become a significant bottleneck in training large models. Thus, it is important to use efficient algorithms for collective communication. We develop TACCL, a tool that enables algorithm designers to guide a synthesizer into automatically generating algorithms for a given hardware configuration and communication collective. TACCL uses a novel *communication sketch* abstraction to get crucial information from the designer to significantly reduce the search space and guide the synthesizer towards better algorithms. TACCL also uses a novel encoding of the problem that allows it to scale beyond single-node topologies. We use TACCL to synthesize algorithms for three collectives and two hardware topologies: DGX-2 and NDv2. We demonstrate that the algorithms synthesized by TACCL outperform the Nvidia Collective Communication Library (NCCL) by up to 6.7×. We also show that TACCL can speed up end-to-end training of Transformer-XL and BERT models by 11%–2.3× for different batch sizes.

1 Introduction

Machine-learning models have been dramatically increasing in size over the past few years. For example, the language model MT-NLG has 530 billion parameters [31] and the Switch-C mixture-of-experts model has 1.6 trillion parameters [18]. Model sizes are expected to further grow to increase model accuracy and perform more complex tasks. These models are too large for the resources of a single GPU and have to be distributed across multiple servers, each with several GPUs, using different parallelism strategies like data, model, pipeline, and expert parallelism [18, 27, 43] for training and inference. Intermediate data and parameters of the model at each GPU are accumulated, shuffled, and transferred over the network between other GPUs for distributed machine learning, depending on the type of parallelism strategy used.

*Work was partially done during an internship at Microsoft Research.

The inter-GPU communication bottleneck. Recent work has shown that GPU idle time spent waiting for network communication can be significant in practice [2, 19, 26, 28]. For instance, BERT [15] and DeepLight [14] spent 11% and 63% of time, respectively, with GPUs idle on a 100 Gbps Ethernet cluster of P100 GPUs [2]. Newer generations of faster GPUs will only make this problem worse. This inefficient use of GPUs shows that there is significant model performance to be gained by optimizing inter-GPU communication.

Collective communication primitives and algorithms. Efficient communication between GPUs is the key to enabling fast distributed ML training and inference. Modern GPU systems use message passing interface (MPI)-based *collective communication primitives*, such as ALLREDUCE, ALLGATHER, and ALLTOALL to perform inter-GPU communication (Figure 2 in §2). *Collective algorithms* implement collective communication primitives. They route data along various paths in the network and schedule the necessary computation (e.g., a sum in ALLREDUCE) while optimizing for latency and bandwidth characteristics of each link in the network. For example, a common collective algorithm for ALLGATHER (all GPUs gather data from all GPUs) is a Ring algorithm, in which all GPUs are logically arranged in a ring and each GPU receives data from its predecessor in the ring and sends a previously received data to its successor. Inefficiencies in collective communication algorithms can cause poor network utilization, causing GPUs to remain idle until inter-GPU transfers complete [53], and thus reducing the overall efficiency of distributed training and inference.

Challenges in designing GPU communication algorithms. Designing algorithms for efficient collective communication on GPU topologies is challenging. First, these algorithms have to strike the right balance between latency and bandwidth optimality. For instance, the commonly used Ring algorithm for ALLREDUCE is not efficient for small input sizes as it has a high latency. Second, GPU communication algorithms have to manage the heterogeneity of connectivity in the underlying topology. For instance, GPUs within a machine (also referred to as a node) are usually connected using fast NVLinks [38] (up to 300 GBps aggregate bidirectional bandwidth per GPU) while GPUs across nodes are connected using slow Infini-

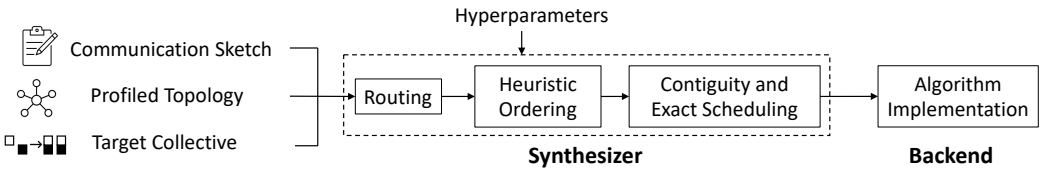


Figure 1: TACCL's novel synthesizer takes as input a communication sketch, profiled topology, and target collective along with synthesizer hyperparameters to generate an algorithm for the collective. The synthesized algorithm is implemented on the hardware cluster using TACCL's backend.

Band [36] links (12.5-25 GBps per NIC). Moreover, these topologies vary significantly between vendors. And finally, searching over the entire space of routing and scheduling algorithms to find optimal ones for communication collectives is computationally prohibitive. In fact, previous approaches that synthesize collective communication algorithms are limited to single-node topologies [9] or 8 GPUs at most [51].

Managing scale for automated algorithm design. Our goal is to automatically obtain efficient algorithms for a given hardware configuration and communication collective. We encode the problem of finding optimal algorithms for communication collectives into a mixed integer linear program (MILP) with the goal of minimizing the overall execution time. Unfortunately, this problem is NP-hard; state-of-the-art commercial solvers like Gurobi [20] can spend several days exploring the search space without finding an optimal algorithm. In this work, we propose a *human-in-the-loop* approach that incorporates high-level inputs of an algorithm designer to efficiently synthesize collective communication algorithms for heterogeneous GPU topologies. We argue that it is easy for algorithm designers to provide a few simple inputs that constrain the search space of algorithms which allows synthesis engines to scale to large GPU topologies.

Communication sketches as user input. It is crucial that the input required from algorithm designers is simple and intuitive. For this, we introduce a new abstraction: *communication sketches* (§3). Inspired by the technique of *program sketching* [47] from program synthesis, in which developers supply a partially specified program with holes that capture the high level structure of the desired program, communication sketches allow algorithm designers to provide high-level intuitions that constrain the search space of algorithms. A synthesis engine fills in the remaining details such as routing and scheduling of the final collective communication algorithm, analogous to how a constraint solver in program synthesis searches the reduced space to fill the holes.

Our solution. We develop TACCL (Topology Aware Collective Communication Library), a system that synthesizes communication algorithms for a given topology and a target collective communication primitive. Algorithm designers can use communication sketches to guide TACCL into synthesizing efficient algorithms for a large range of hardware topologies. We develop a novel encoding of the problem in TACCL's synthesizer to scale beyond single-node topologies. Figure 1 shows an overview of TACCL's design.

Synthesizing algorithms from communication sketches. TACCL's synthesis approach builds on the solver based synthesis approach in SCCL [9], where the space of possible algorithms is directly encoded in a *satisfiability modulo-theories* (SMT) solver. SCCL does not scale to the sizes of clusters used by modern machine learning workloads. We present a novel *mixed integer linear programming* (MILP) encoding of the collective algorithm synthesis problem that improves scalability by first solving a bandwidth-relaxed version of the problem to decide on *routing*, followed by ordering heuristics and a second bandwidth-constrained problem to find a valid *scheduling* (§5). In addition to improving scalability, TACCL's MILP formulation allows modeling of heterogeneous links with different per-message overhead characteristics. This overcomes the limitation in SCCL [9] that prevents it from faithfully targeting distributed GPU clusters.

Results. We use TACCL to synthesize efficient algorithms for a range of collectives like ALLGATHER, ALLTOALL, and ALLREDUCE, and for different hardware backends like Azure NDv2 [6] and Nvidia DGX-2 [35] (§7). We compare TACCL to the state-of-the-art Nvidia Collective Communication Library (NCCL). TACCL synthesized an ALLGATHER algorithm for two Nvidia DGX-2 nodes (32 GPUs). This algorithm is up-to 6.7× faster than NCCL for small-to-moderate input sizes. For large input sizes on the same hardware, TACCL synthesized a different ALLGATHER algorithm that nearly saturates the inter-node bandwidth and is up-to 25% faster than NCCL. TACCL synthesized an ALLTOALL algorithm for two Azure NDv2 nodes (16 GPUs) that is up-to 66% faster than NCCL. Finally, we replaced NCCL with TACCL using only a two-line code change in PyTorch and found that TACCL achieves a speed-up of 17% in end-to-end training of a mixture-of-experts model that uses ALLTOALL and ALLREDUCE, and a speed-up of 11% - 2× in end-to-end training of a Transformer-XL model distributed over 16 GPUs for varying batch sizes. TACCL's codebase is open-source and is actively in use by researchers at universities and practitioners at Microsoft for Azure's GPU virtual machines. [1]

[1] https://github.com/microsoft/taccl

2 Background and Motivation

Collective communication in distributed ML workloads. Multi-GPU ML workloads typically communicate using MPI-style collectives like ALLGATHER, ALLTOALL, and ALLREDUCE shown in Figure 2. These primitives capture the application's intent behind the communication, thus allowing collective communication libraries to optimize for specific hardware configurations. In ALLGATHER, every GPU receives the data buffers of all other GPUs (left diagram in Figure 2). In ALLTOALL, every GPU receives different parts, or chunks, of the data buffers present on all GPUs. This effectively transposes the data chunk from buffer index to GPU index as can be seen in center diagram in Figure 2. In ALLREDUCE, every GPU ends up with a data buffer that has the results of performing a point-wise computation (e.g., sum in right diagram in Figure 2) over the same data index of all GPUs.

The parallelism strategy for the distributed ML workload determines which collective communication primitive is used. Data parallelism and some tensor model parallelisms [43] make use of the ALLREDUCE collective to aggregate gradients and intermediate data respectively from multiple GPUs. Expert parallelism [18, 27] and common deep learning recommendation models (DLRM) [32] make use of the ALLTOALL collective to shuffle intermediate data between experts and embedding lookup data between GPUs respectively. DLRMs [32] also make use of the ALLGATHER collective and another REDUCESCATTER collective to perform embedding lookups from embedding tables sharded over multiple GPUs.

Existing approaches to collective algorithms. Collective algorithms must be designed considering the target input sizes and the heterogeneity of the target topology. However, most collective communication libraries used for distributed ML today, including the state-of-the-art NCCL, use pre-defined templates of collective algorithms superimposed onto a target topology. For example, for collectives like ALLGATHER and REDUCESCATTER, NCCL identifies rings in the target topology and uses the Ring algorithm. For n GPUs, this algorithm requires $n-1$ link transfer steps per data chunk and is not ideal for smaller data sizes where link transfer latencies dominate. Further, this algorithm treats the slow inter-node and fast intra-node links similarly, scheduling equal number of data transfers across both. The communication is thus bottlenecked on the slower inter-node links, when it could have benefitted by sending more node-local data (i.e. data of GPUs local to the node) over the faster intra-node links instead.

For the ALLTOALL collective, NCCL implements the collective algorithm as peer-to-peer data transfers between all pairs of GPUs. This algorithm is topology-agnostic and often inefficient. For the ALLREDUCE collective, NCCL chooses between two algorithms — Double-Binary-Tree [34] and Ring. This decision is made according to the communication input size and number of nodes, but might not be most accurate, as it is based on hardcoded latency and bandwidth profiling done previously by Nvidia on their machines.

Designing efficient collective algorithms requires careful analysis of the topology and its performance with different buffer sizes. Recent work [9, 51] has shown that synthesis is a promising approach for generating collective algorithms for different topologies and to achieve bandwidth and latency optimality. However, scaling these approaches to multi-node (i.e. multi-machine) distributed GPU topologies has been a challenge. We measured the synthesis time for ALLGATHER and ALLTOALL collectives on topologies of two Azure NDv2 nodes and two Nvidia DGX2 nodes (Figure 5) using SCCL [9, 30]. We modified the codebase to include both topologies and attempted to synthesize the collectives with a 24-hour time limit set for each synthesis query. Given a 24-hour time limit, SCCL's `pareto-optimal` solver strategy did not finish synthesis for any combination of collective and topology. The only algorithm that SCCL could synthesize within the time limit was a latency optimal algorithm for ALLGATHER on two NDv2 nodes.

Low-effort inputs from algorithm designers. The search space of possible algorithms to implement a collective is intractably large and cannot be explored via brute-force. Deciding whether or not to route data chunks from n GPUs over l links in a topology has $O(2^{n\times l})$ combinations. As we scale to multi-node topologies, n as well as l will also scale, increasing the exponent quadratically. The search space explodes further if we consider the problem of ordering data sends at each link along with deciding routing for the data. We argue that high-level inputs from a human algorithm designer help reduce the search space to make algorithm synthesis more tractable. In the most extreme case, the designer would hand-write the entire algorithm. However, handcrafting data routing and scheduling over links to implement a collective is complex and requires many design choices. Instead, designers only provide input in the form of a communication sketch around which TACCL synthesizes an algorithm. Our goal is to ensure that providing inputs is a low-effort activity, but can discard large parts of the search space to achieve improvements in running-time of the synthesis engine.

Synthesis technique. TACCL synthesizes a collective algorithm by deciding the route that each data chunk in the collective should take in the topology as well as the ordering of chunks at every link. Even with communication sketches which reduces the search space for the synthesizer, this decision problem is NP-hard and the complexity increases exponentially with number of GPUs. To make the problem more tractable, we first relax the synthesis problem to solve just the routing of all data chunks and then heuristically order chunks sent over the same links according to bandwidth constraints. TACCL's synthesizer design along with communication sketches help TACCL synthesize efficient collectives for multi-node topologies.

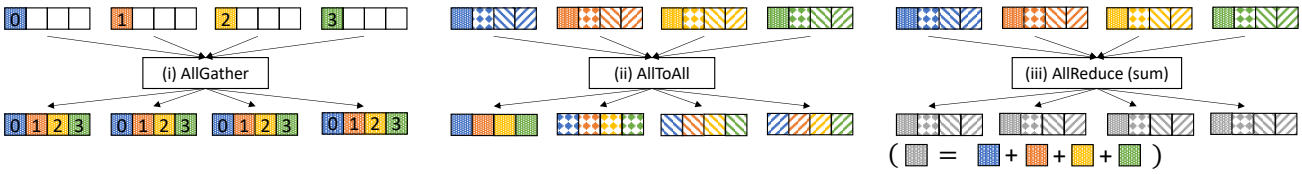


Figure 2: The initial and final data buffers on four GPUs participating in different collectives.

3 Communication Sketches

This paper proposes a new form of sketching [47] as an effective tool for users to communicate interesting aspects of collective communication algorithms to synthesis backends. Sketching approaches must strike a balance between allowing users to omit implementation details while still providing enough direction for the synthesis to scale. In our experience, *routing* is an aspect of collective communication that we often have intuitions about, while reasoning about *scheduling* tends to be tedious and better left to synthesis. Moreover, properties about scheduling are routing dependent since the order of operations is only relevant when routes intersect, which makes them harder to express. Meanwhile, interesting properties about routing are expressible globally, e.g., "never send over the InfiniBand NIC from this GPU". Therefore, we ask the algorithm designer (user) for four low-effort inputs as a part of the communication sketch:

- Specify the *logical topology* as a subset of the actual physical topology that the algorithm will operate on. This constrains the routes chosen by the communication algorithm and alleviates over-subscription of low-bandwidth links. For example, the outgoing links of all but one GPU can be removed in the logical topology to force all data going to remote GPUs to be relayed through one GPU.
- The logical topology abstracts away switches (e.g., NVSwitches, IBSwitches) in the GPU network. Users can annotate switches in the topology for the synthesizer to use certain *switch-hyperedge policies*, enabling it to apply synthesis policies that help algorithms avoid contention.
- Provide *algorithm symmetry* based on the symmetries in the topology and the collective.
- Specify the expected *input size* of the data, which is used as a part of the synthesis engine's built-in cost model.

We explain all parts of the communication sketch and provide an example sketch written for TACCL in Appendix A.

3.1 Logical Topology

The core of TACCL's communication sketch is a *logical topology* consisting of a subset of the physical topology after excluding links that the user prefers TACCL to avoid. A logical topology has as many nodes as the physical topology and inherits the cost model produced by TACCL's profiler for the physical topology. Logical topologies omit NICs and switches and use switch-hyperedges (Section 3.2), abstracting them away into links between GPUs. The reason is two-fold: TACCL runtime is embedded in NCCL runtime and NCCL has no direct control over NIC or switch use, and it allows TACCL to reason over a smaller graph thus enhancing scalability. Section 3.2 discusses implications of this abstraction.

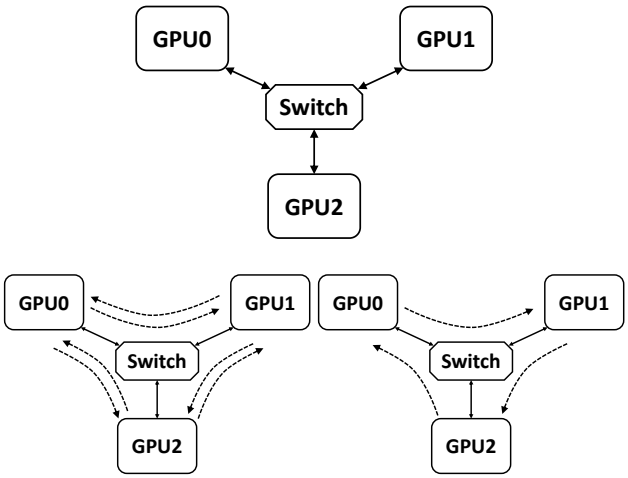
Example 3.1 (Sketching inside an NDv2)**.** Consider the physical topology of an Azure NDv2, given by the union of Figure 5a and Figure 5b. While NCCL is able to communicate over both the NVLink and PCIe connections, the bandwidth offered by the NVLinks is much higher than that of PCIe, and thus it is reasonable to set the logical topology to just the NVLink subgraph in Figure 5a.

Example 3.2 (Distributed sketching for NDv2 clusters)**.** It is essential to use PCIe connectivity for distributed collective communication with multiple NDv2 systems since the NIC is connected to GPUs over PCIe (Figure 5b). Due to lack of GPUDirect RDMA [1] on these systems, all communication over PCIe must pass through host memory. Therefore, care must be taken in choosing which links to use, as the PCIe links between PCIe switches and the CPU are oversubscribed. Obtaining maximum throughput communication requires a logical topology that avoids conflicting flows on the oversubscribed PCIe links. To build a logical topology for a cluster of NDv2 systems, a pair of receiver and sender GPUs is selected for each NDv2 such that the selected GPUs and the NIC are connected to the same PCIe switch.

3.2 Switch-Hyperedges

In a switched fabric with full bisectional-bandwidth, like the NVSwitch or IBSwitch fabrics in DGX-2 and NDv2 systems, nodes can simultaneously communicate at the full bandwidth of their ingress or egress links. However, as the number of connections through a switch, originating from a single GPU or NIC increases, the resulting queuing delays increase the latency. Figure 4 plots the accumulated ingress/egress bandwidth of exchanging varying volume of data (up-to 200-400 MB) for different number of connections over NVSwitches in a DGX2 node (left) and over IBSwitches among four DGX2 nodes (right). In both cases, the bandwidth drops as the number of connections increases despite the volume of data re-

(a) Physical topology with a switch.



(b) Max. connections strategy. (c) Min. connections strategy.

Figure 3: Effects of switch-hyperedge policies.

maining constant. However, for small input sizes, the difference for different number of connections is not significant. TACCL's logical topology does not model switches and does not capture the effect of number of connections.

TACCL incorporates the effect of multiple connections using *switch-hyperedges* in the synthesizer to control the number of connections between GPUs and switches. A switch-hyperedge replaces a switch with a set of direct links in the logical topology for the entire runtime of an algorithm. The synthesizer still has the freedom to select which direct links are imposed. To control the number of direct links for each switch-hyperedge, TACCL provides three policies for a user: (1) *maximize* the number of links, (2) *minimize* the number of links, and (3) freely choose any number of links. These policies are enforced by adding the number of Connections to the objective function (see Appendix B.1 for details).

Example 3.3 (Sketching for congestion)**.** Figure 3a shows a physical topology of three GPUs connected by a switch, where each GPU can communicate with any other GPU.

Figure 3b shows a logical topology with a switch-hyperedge that TACCL may choose with maximizing number of connections policy. This is desirable for small data sizes that result in low likelihood of congestion at the switch with large number of connections as shown in Figure 4.

In Figure 3c TACCL has minimized the number of connections, effectively resulting in a Ring topology. This is desirable for larger data sizes, as restricting the number of logical connections limits the congestion in the switch (Figure 4).

3.3 Algorithm Symmetry

Many collective communication algorithms are symmetric in a variety of ways. For example, ring algorithms follow a ring symmetry or in hierarchical algorithms, the local phases inside all machines are symmetric to each other. Inspired by this, TACCL offers a generic way to enforce algorithms to be symmetric.

The user may enforce a symmetry by supplying an *automorphism* of the logical topology and collective, i.e., a permutation of the ranks and chunks that maintains the structure of the topology and the collective pre- and post-conditions, and a *partition* of the ranks such that the automorphism maps each subset of ranks to some subset of the partition. TACCL will then restrict synthesis to algorithms with the same symmetry for all chunk transfers.

Example 3.4. Consider a cluster of two NDv2 systems and the task of synthesizing an ALLGATHER. A hierarchical symmetry may be specified with an automorphism composed of a the permutation $[8,\ldots,15,0,\ldots,7]$ for both chunks and ranks, and a partition $\{\{0,\ldots,7\},\{8,\ldots,15\}\}$. Now if an algorithm performs a send of chunk 0 from rank 0 to rank 1, then it must also include a send of chunk 8 from rank 8 to rank 9. However, sends between GPUs in different NDv2s, e.g., between 0 and 8, are not affected by the symmetry.

Since the internal topologies of NDv2 systems are identical, enforcing this symmetry is reasonable and helps TACCL scale to larger distributed topologies. Meanwhile, TACCL still has the freedom to synthesize the top-level algorithm and connect the systems to each other as it best can.

4 Physical Topologies of GPU systems

ML engineers use a variety of multi-GPU systems to meet the scaling challenges posed by growing ML models. Before users can effectively sketch algorithms for TACCL to synthesize, they must understand the *physical topology* of the target multi-GPU system. However, the performance characteristics of their heterogeneous links are sparsely documented and for some cloud offerings [5] even the topology is not given. To address this, TACCL includes a *physical topology profiler* to measure performance characteristics of links (§4.1) and to disambiguate the topology of some multi-GPU systems (§4.2). This section also serves as a concrete introduction into two target systems: Azure NDv2 and Nvidia DGX-2.

4.1 α-β Cost Model and Link Profiling

In the well-known α-β [21] cost model, α is the latency of a link and β is the inverse of its bandwidth. The cost of sending a chunk of size s along a link is $\alpha+\beta\cdot s$. TACCL's synthesizer adopts the α-β cost model for simplicity of encoding and tractability, but TACCL's communication sketches expose features that provide users additional control to avoid excessive concurrency and congestion (see Section 3), which are not modeled by the α-β cost model. α and β are affected by both the interconnect hardware and the software stack running

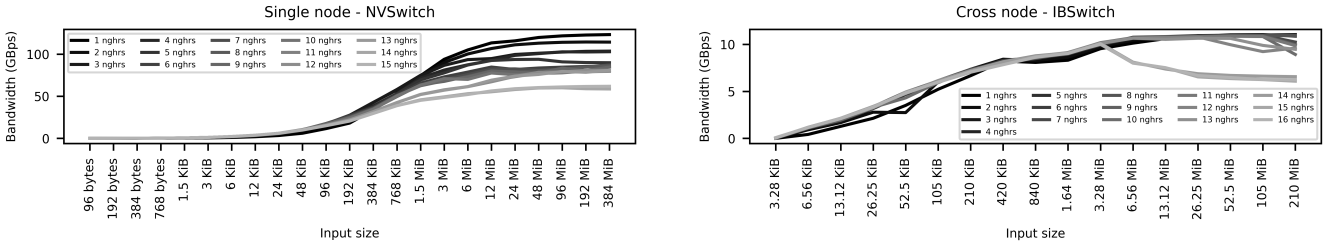
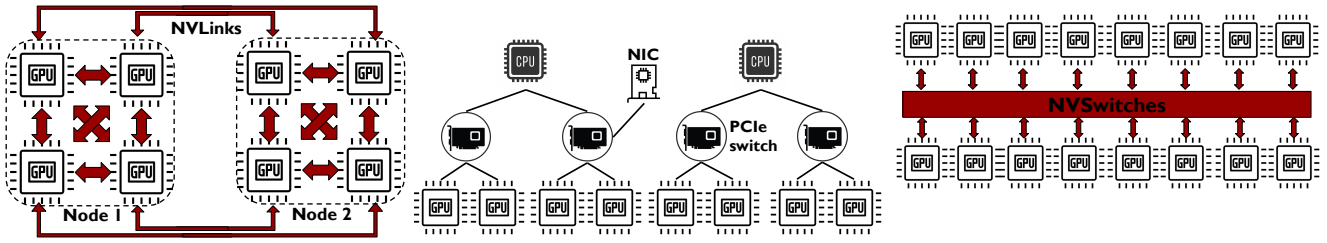


Figure 4: Multi-connection with varying number of GPU neighbors and data volume.



(a) NVLink connectivity of a NDv2. (b) PCIe connectivity of a NDv2. (c) NVLink connectivity of a DGX-2.

Figure 5: Aspects of physical topologies in various GPU systems.

the collective algorithm (for example software thread fences). TACCL's topology profiler measures and infers the α and β costs of different types of links in a GPU system.

Modern GPU systems, e.g., Azure NDv2 (Figure 5a) and Nvidia DGX-2 (Figure 5c), have the following types of interconnects: (1) **Peripheral Component Interconnect Express (PCIe)**, (2) **NVLink** [38], (3) **Infiniband** (IB) NICs [36]. A PCIe bus connects GPUs to CPUs with limited shared bandwidth (PCIe Gen3 offers $\approx$ 13 GBps). PCIe connections often form a hierarchy with PCIe switches (Figure 5b). NVLink [38], however, is a GPU to GPU *intra-node* connection with dedicated bandwidth. NVLinks are either directly connected to other GPUs (NDv2 in Figure 5a) or they are connected to other GPUs via NVSwitches [39] (DGX2 in Figure 5c). NVSwitches enable fully-connected GPU-GPU communication through NVLinks. IB is an inter-node interconnect which allows GPUs to communicate with GPUs in other nodes like in the Azure NDv2 (Figure 5b). IB NICs are usually connected to PCIe switches and GPUs may communicate directly with the NICs through Remote Direct Memory Access (RDMA) or indirectly via host memory.

The profiler empirically derives the α and β parameters of different links in the network by performing peer-to-peer data transfers between GPUs. We send n chunks one after another on a link and measure the time to transfer. As per the $\alpha-\beta$ cost model, the time to transfer is $n \cdot (\alpha + \beta \cdot s)$. We then send n chunks all at once on the link and attribute that time to be $\alpha + n \cdot \beta \cdot s$. Using several measurements of time to transfer, we solve for α and β. Table 1 shows the α and β values for

	Azure NDv2		Nvidia DGX-2	
Link	α (us)	β (us/MB)	α (us)	β (us/MB)
NVLink	0.7	46	0.7	8
InfiniBand	1.7	106	1.7	106

Table 1: Experimentally obtained α and β costs for Azure NDv2 and Nvidia DGX-2 nodes.

NDv2 and DGX-2 systems. Using these values, we expect that for transfers between two Azure NDv2 nodes over InfiniBand (IB), a sending two 32 KB chunks together as a single 64 KB chunk will be 17% faster as compared to sending two 32 KB chunks one after the other. However, chunks sent together can only be forwarded once the last chunk is received. Based on the α-β values, TACCL's synthesizer determines if and when to send chunks together on a link.

The α-β cost model causes TACCL's synthesizer to formulate an MILP formulation as opposed to an LP since an algorithm has to be expressed in terms of discrete chunks.

4.2 Inferring Multi-GPU Topologies

For Azure NDv2 systems the physical topology was not fully documented: while the NVLink topology (Figure 5a) is known to match that of Nvidia DGX1, we did not know how GPUs and the one 12.5 GBps Infiniband NIC were connected with PCIe. PCIe peer-to-peer communication (and thus GPUDirect RDMA [1]) is not enabled on these machines, meaning that all communication happens through buffers in

CPU memory over potentially shared PCIe links. Further, virtualization obscures the true PCIe topology (all 8 GPUs and the NIC appear directly connected to one CPU) and NUMA node and GPU IDs are not assigned consistently from VM to VM. This means that, without additional information, software cannot avoid contention over shared PCIe links, creating interference and high variance in performance.

To determine the PCIe topology, TACCL's profiler sends bandwidth and latency probes between the two CPUs, between pairs of GPUs, and between CPUs and the NIC. It answers the following questions:

- Which CPU is nearest to the NIC? We answer this using the latency of loopback operations between the NIC and each CPU.
- Which GPUs share a PCIe switch? We find all pairs of GPUs that get low bandwidth in a simultaneous copy to the CPU, indicating contention.
- Which GPUs share a PCIe switch with the NIC? We find which GPUs get low GPU to CPU bandwidth while the CPU is doing a loopback with the NIC. The CPU in this case is the one that is closer to the NIC.

With this profiling information we were able to deduce the PCIe topology (Figure 5b). Each CPU has two PCIe switches connecting to two GPUs each, and the Infiniband NIC is connected to one of these switches. Additionally, by running the profiler on every new NDv2 VM TACCL is able to select one of the NVLink topology's four automorphisms and set the `CUDA_VISIBLE_DEVICES` environment variable such that the NIC is always placed close to GPU 0.

5 TACCL Synthesizer

Once the user has written a communication sketch, they are ready to call TACCL's synthesizer. This section describes the synthesis process TACCL uses, as well as additional hyperparameters available to the user.

5.1 Problem Formulation

GPUs participating in a communication collective partition their initial data into C equal chunks where C is a hyperparameter selected by the user. TACCL's synthesizer routes and schedules these chunks. Given a communication sketch and a collective, the synthesizer decides chunk transfer schedules across every link in the network, such that each chunk reaches its destination GPUs as specified by the collective.

TACCL encodes this problem as a mixed integer linear program (MILP) with binary and continuous decision variables. The encoding has a continuous variable called *start_time* for every chunk and GPU to indicate when a chunk is available at a GPU. A binary variable *is_sent* for all chunk and link pairs denotes if a chunk is sent over a link. Another continuous variable *send_time* indicates when a chunk is sent over a link. The encoding has bandwidth and correctness constraints to ensure the correctness of a chunk transfer schedule. The objective of the MILP is to minimize *time* which is a continuous variable indicating the maximum time among all chunks that must reach their destination GPUs. Details of these variables and constraints are in Appendix B.

Additionally, TACCL's synthesizer also decides if it should merge some chunks and transfer them contiguously as one large buffer over a link. Sending n chunks contiguously in one send instruction over a link requires paying only one α latency cost whereas sending n chunks one after the other requires paying $n \times \alpha$ latency costs. Note that this does not change the β bandwidth cost. However, sending n chunks separately over a link enables TACCL to order them such that subsequent dependent sends from the destination of the link could be scheduled earlier. TACCL's synthesizer navigates this tradeoff to minimize the time. TACCL uses this feature only for IB transfers due to their high α cost and ignores it for NVLinks due to their lower latency.

MILP problems in general are NP-hard. Luckily, there are solvers such as Gurobi [20] that apply heuristics to solve MILPs in a feasible way. However, this requires careful consideration regarding the number of variables and constraints in the formulation. In TACCL's formulation, transferring chunks over a link cannot overlap and an ordering among them is required. Therefore, potentially a binary decision is needed for every pair of chunks that may traverse a link. If we assume there are C chunks for a collective problem, there are $O(C^2)$ such decisions per link. Moreover, as the number of nodes increase, the number of links increase linearly (larger topology) and the number of chunks for a collective increases linearly (ALLGATHER) or even quadratically (ALLTOALL). This large set of variables and constraints leads to infeasible solver time and memory requirements.

To solve this problem, we divide the synthesis into three parts. First, the synthesizer solves an optimization problem to determine the path used by every chunk without fixing any ordering among chunks, then it heuristically orders the chunks over every link, and finally, it solves another optimization problem to determine chunk contiguity. Complete formal descriptions of each step are in Appendix B.

Step 1: Routing solves a MILP for finding the path of each chunk independent of other chunks, allowing chunks sent over a link to overlap. The objective of this MILP is to minimize the time, which we constrain to be the maximum of two sets of variables. (1) for each link, the number of chunks that traverse that link multiplied by the transfer time of a chunk over that link. (2) for the path of each chunk, the summation of transfer times of the chunk along every link in the path. Note that this is only a lower bound on the time since we do not consider link contention or chunk ordering. TACCL also constrains each chunk's path to be via GPU ranks that are on the shortest paths from their sources to their destinations using the links the user decided to include in the logical topology. If

the communication sketch specifies an algorithm symmetry, TACCL adds the constraints for the symmetric sends. Replacing switches with switch-hyperedges is also applied in this step. For each switch-hyperedge, a user-provided policy on the number of unique connections to/form a switch is applied (see Section 5.2).

TACCL uses Gurobi [20] to solve this MILP and the solution gives every chunk a start_time for each GPU along its path. Clearly this step solves chunk routing, but only partially solves the chunk scheduling and contiguity problem and requires follow-up steps (explained next) to account for ordering the chunks sent over a link as well as minimizing α costs of sends. However, by using this technique, TACCL's synthesizer is able to reduce binary variables needed from $O(C^2)$ to $O(C)$ per link.

Step 2: Heuristic Ordering decides the chunk ordering sent on each link based on a heuristic. Note that this step is not an MILP and solely solved by a greedy algorithm. Regardless of when each chunk becomes available at a GPU, this step assigns a total order on the chunks sent over a link $l = (src, dst)$. This is decided by two heuristic functions. (1) chunks which need to traverse the longest path from *src* to their final GPU, have higher priority. (2) In case there is tie in (1), chunks which have traversed the shortest path from their initial GPU to *src*, have higher priority. This ordering will be used in Step 3 to assign the final schedules.

Step 3: Contiguity and Exact Scheduling solves an MILP problem to decide which chunks to send contiguously and gives the exact schedule. The path to be taken by chunks and their ordering over links have already been determined by the previous steps which are added as constraints to this MILP. The start_time and send_time variables are reassigned in this step by considering both the α and β costs for each transfer. In this step, the synthesizer allows either sending one chunk at a time or sending multiple chunks contiguously. This offers a trade-off between (1) sending the chunks that are available at the same time for a link according to the ordering in step 2 so that the subsequent sends can be scheduled earlier or (2) sending the chunks contiguously in one send instruction to save the latency cost. The objective of this MILP is to minimize the total time by enforcing all constraints which in TACCL solved by Gurobi [20]. The solution gives the exact schedule for each chunk. The details of these constraints and their formulation are in Appendix B.

5.2 Synthesizer Hyperparameters

TACCL's synthesizer has some additional parameters that control the synthesis process. These are provided by the user to the synthesizer (see Figure 1) through the communication sketch. Details of each parameter is described in Appendix A.

Buffer Size. TACCL needs the size of input/output buffers of a collective for the α-β cost model. In ML workloads the input/output buffer size is a known fixed value.

Chunk Partitioning. The data buffer at each GPU at the start of the collective can be partitioned into multiple equal chunks. Each chunk is considered as an atomic scheduling unit by the synthesizer and different chunks of the same data buffer can be routed over different links. The semantics of a collective forces a minimum number of chunks such as ALLTOALL which needs at least as many chunks as the number of GPU for each buffer. On one hand, using the minimum number of chunks is often times ideal for finding latency-optimal algorithms. On the other hand, providing a higher number of chunks allows the synthesizer to better utilize the links that might be idle otherwise which is better for finding bandwidth-optimal algorithms.

Switch-Hyperedge Policy. TACCL can enforce policies for the number of connections established over a set of links in a switch-hyperedge by counting links utilized for data transfer and setting this count as a part of the MILP objective. The `uc-max` policy will maximize the number of connections, which performs best for small data sizes, while `uc-min` will minimize the number of connections, which works well when the data size is large and congestion is a concern.

5.3 Synthesizing combining collectives

TACCL synthesizes combining collectives (i.e., collectives that combine chunks like REDUCESCATTER and ALLREDUCE) by utilizing synthesis of non-combining collectives, similar to the technique used by SCCL [9]. REDUCESCATTER can be implemented as an "inverse" of ALLGATHER— a send from a source GPU in ALLGATHER is instead received and reduced on the source GPU. However, simply inverting the sends does not work — a GPU may simultaneous send on different links in an ALLGATHER, but it cannot reduce all receives together in the inverse case. We thus order the inverse sends using heuristic ordering followed by contiguity encoding in order to synthesize REDUCESCATTER. ALLREDUCE is synthesized directly by concatenating REDUCESCATTER with an ALLGATHER algorithm.

6 Backend

The synthesizer described above generates an abstract algorithm that specifies the order in which the nodes communicate the various chunks. The goal of the backend is to implement this abstract algorithm. To do so, we extend NCCL [37] with an *interpreter* which we call TACCL runtime. While any communication algorithm can be trivially implemented using NCCL's point-to-point sends and receives, TACCL runtime enables us to execute the entire algorithm in a single kernel launch, eliminating multiple launch overheads. In addition, by reusing NCCL transport mechanisms, TACCL runtime is able to support all of NCCL's communication backends such as IB, Ethernet, NVLink, and PCIe.

6.1 TACCL runtime

The input to TACCL runtime[2] is a TACCL-EF program, which is an XML format for representing collective algorithms. TACCL-EF programs operate on three buffers: input, output and scratch. For each buffer, the program specifies the number of chunks it will be sliced into such that all chunks are equal size. Every step of the algorithm is expressed in terms of these chunks.

The program is divided into a set of GPU programs made up of threadblocks. Each threadblock is made up of a series of steps that are executed sequentially, with each step specifying an instruction and operands as indices into the input/output/scratch buffers. The current instruction set includes sends, receives (with optional reduction), and local copies. To simplify the implementation of TACCL runtime, each threadblock can send to and receive from at most one GPU. Additionally, threadblocks within a GPU can synchronize by indicating that one step depends on another step, which will cause the interpreter to wait until the dependency has completed before executing the dependent step.

The TACCL runtime extends NCCL and it is backward compatible with its API. Therefore, integrating TACCL runtime into machine learning frameworks such as PyTorch is a single line change wherein that change swaps the third-party NCCL library for TACCL runtime. This allows TACCL to dynamically swap in collective algorithms generated for any training/inference workload using `torch.distributed`.

6.2 Lowering to TACCL runtime

To target TACCL-EF, abstract algorithms are lowered to the executable format. The sets of sends operating on abstract chunks that comprise the steps of the algorithm are transformed into pairs of send and receive operations operating on concrete buffer indices. Furthermore, these operations are placed sequentially into threadblocks and any necessary dependencies recorded between them.

Buffer allocation. Input and output buffers are preallocated by the user and passed to the collective. Scratch buffers are allocated by the TACCL runtime per TACCL-EF. Chunks are indices in the input, output and scratch buffers. For chunks that are common for both the input and the output buffers (e.g. as in ALLGATHER) a local copy from input to the output buffer is performed at the end.

Instruction generation. The operations of the abstract algorithm are split into two instructions for the sender and receiver GPU, and chunks are translated into buffer references and indices according to the buffer allocation.

Dependency insertion. TACCL transforms a synthesized algorithm into the asynchronous execution model of TACCL-EF and dependencies for each buffer index are inserted to ensure that the data dependencies present in the abstract algorithm are honored.

Threadblock allocation. Instructions are grouped such that all of them are either sending to at most one GPU and/or receiving from at most another GPU (possibly different). Order of the instructions inside a group should follow the order of the abstract algorithm. TACCL allocates a threadblock for each group of instructions.

Instances. NCCL and consequently TACCL runtime cannot saturate the bandwidth of a link in a topology using a single threadblock. Thus, TACCL generates multiple instances of the algorithm to maximize the performance. This is done by subdividing each chunk into n subchunks that follow the same path as the parent chunk. All groups of instructions and their threadblocks are duplicated n times and executed in parallel. §7.2 explores the performance implications of choices of n.

7 Evaluation

We evaluate algorithms obtained with TACCL for ALLGATHER, ALLTOALL, and ALLREDUCE collectives on a cluster of 32 GPUs comprised of two Nvidia DGX-2 nodes or up-to four Azure NDv2 nodes. To compare performances, algorithm bandwidth [33] measurement is used which is calculated by input buffer size divided by execution time. We synthesize TACCL algorithms by exploring different communication sketches and compare them against the popular Nvidia Collective Communication Library (NCCL) (v.2.8.4-1). This section analyzes how different communication sketches impact the performance of the algorithms synthesized by TACCL. In particular, we perform ablation studies by varying the inter-node connections in the logical topology, changing synthesizer hyperparameters, and changing the number of instances used when lowering to TACCL-EF. To evaluate how TACCL's speedups translate to end-to-end performance, we use algorithms generated by TACCL in two large language models, Transformer-XL and BERT. Finally, we discuss the synthesis time required by TACCL to generate these algorithms.

We believe our focus on up to 32 GPUs covers a large section of important use cases: in an internal cluster of DGX-2 nodes at Microsoft, the sum of GPUs in jobs of at most 32 was 93.7% of all jobs in the second half of 2021.

7.1 Standalone Experiments

All our communication sketches for DGX-2 and NDv2 use a hierarchical symmetry like the one in Example 3.4.

7.1.1 ALLGATHER

ALLGATHER on DGX-2. Figure 6(i) shows the algorithm bandwidth for TACCL's synthesized algorithms on two DGX-2 nodes for each output buffer size and plots it against that of

[2]Link to code: https://github.com/microsoft/msccl

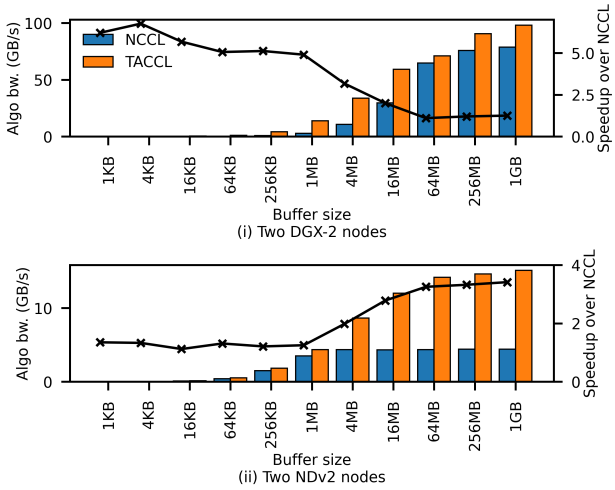


Figure 6: ALLGATHER comparisons of NCCL to TACCL's best algorithm at each buffer size.

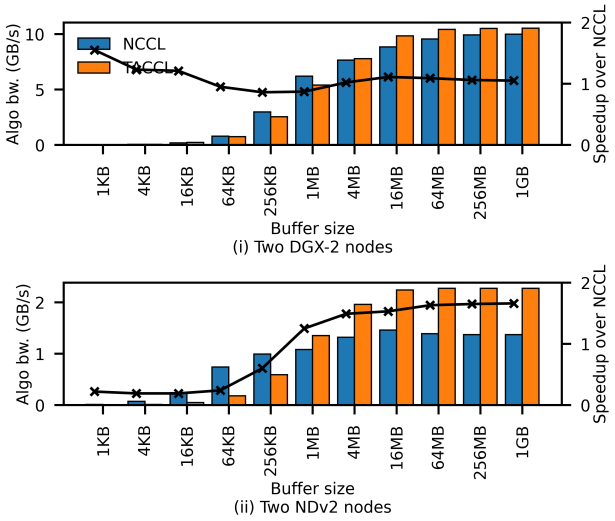


Figure 7: ALLTOALL comparisons of NCCL to TACCL's best algorithm at each buffer size.

NCCL. We show the speedup of TACCL's algorithms over NCCL on the right Y-axis of the plot. We used two different sketches for this topology which will be explained next.

A DGX-2 node has 16 V100 GPUs (Figure 5c) where each pair of GPUs share a PCIe switch with a NIC. This makes it natural to assign one GPU in a pair to be a receiver and the other to be a sender by eliminating outgoing and incoming links, respectively, in the logical topology. We design a sketch (*dgx2-sk-1*) that uses this logical topology, sets chunk size to 2MB, uses two chunk partitions for each buffer, and the sets switch-hyperedge policy to `uc-min`. With this sketch, TACCL synthesizes an ALLGATHER algorithm for two DGX-2 nodes. This algorithm almost saturates the inter-node bandwidth during the entire run of the algorithm and provides a $20\% - 25\%$ speedup over NCCL for large buffer sizes in the 256MB - 1GB range.

Next, we design a sketch (*dgx2-sk-2*) for smaller sizes. This sketch allows both GPUs in a pair to utilize the shared NIC. However, local GPU i on each node is only allowed to send/receive to/from local GPU i on the other node. Since the IB is shared, we double the β cost for each IB transfer to $2 * \beta_{IB}$ cost. In this sketch, chunk size is set to 1KB and the switch-hyperedge policy is `uc-max`. Using this sketch TACCL synthesizes an algorithm that is $4.9\times - 6.7\times$ faster than NCCL in the 1KB - 1MB range, and $10\% - 3.8\times$ faster than NCCL in the 2MB - 64MB range. On inspecting this algorithm, we found that TACCL's synthesized algorithm overlaps inter-node sends with intra-node all-pair ALLGATHER of node-local data chunks followed by an intra-node all-pair ALLGATHER of the node-external chunks received over IB.

Figure 6(i) shows the algorithm bandwidth and the speedup over NCCL baseline for the best of these two sketches for each output buffer size.

ALLGATHER on NDv2. The sketch we used, *ndv2-sk-1*, uses the logical topology discussed in Example 3.2, in which a sender and a receiver GPU were dedicated such that they are on the same PCIe switch as the NIC. We use a single instance when lowering algorithms into TACCL-EF for data sizes 1MB and below, and use 8 instances for larger data sizes. Figure 6(ii) compares the synthesized algorithms to NCCL on two Azure NDv2 nodes. TACCL's synthesized algorithms are $12\% - 35\%$ faster than NCCL for buffer sizes of 1KB - 1MB, and $61\% - 3.4\times$ faster than NCCL for sizes larger than 1MB. These algorithms better saturate the inter-node bandwidth thanks to the dedicated send/receiver GPUs.

We similarly synthesize ALLGATHER algorithms for four NDv2 nodes and present the results in Figure 11(i) in Appendix C. These algorithms are $10\% - 2.2\times$ faster than NCCL depending on buffer size.

7.1.2 ALLTOALL

ALLTOALL on DGX-2. We explore the synthesis of ALLTOALL algorithms by reusing the *dgx2-sk-2* communication sketch designed in the previous section. Figure 7(i) compares the resulting algorithm on two DGX-2 nodes. The synthesized algorithm using this sketch performs up-to 15% faster than NCCL for batch sizes of 2MB and larger. For this sketch, TACCL's synthesizer coalesces chunks sent in inter-node transfer in this algorithm, which reduces the latency of transfers over IB. TACCL also uses a communication sketch with chunk size set as 1KB and a logical topology where GPUs have links to all other GPUs connected via the NIC (*dgx2-sk-3*). This algorithm is up-to 55% faster than NCCL for small buffer sizes ranging from 1KB to 16KB.

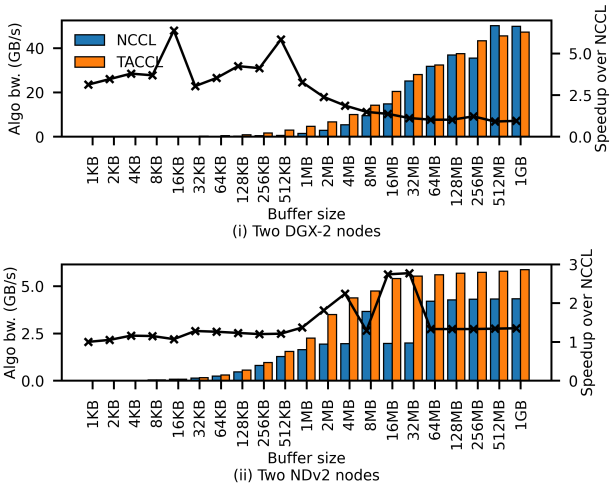


Figure 8: ALLREDUCE comparisons of NCCL to TACCL's best algorithm at each buffer size.

ALLTOALL on NDv2. Figure 7(ii) shows a comparison of TACCL's best algorithms for ALLTOALL on two Azure NDv2 nodes against NCCL. We reuse the communication sketch *ndv2-sk-1* and set the chunk size to 1MB. The generated algorithms run $53\% - 66\%$ faster than NCCL for buffer sizes between 16MB - 1*GB* We explore another sketch (*ndv2-sk-2*) with a logical topology in which all GPUs in a node are fully-connected to all the GPUs in the other node and set chunk size as 1KB. The algorithm generated by TACCL using this sketch performs up-to 12% faster than NCCL for buffer sizes from 1KB to 128KB.

For four NDv2 nodes, TACCL's synthesized algorithms uses communication sketch *ndv2-sk-1* and they are up-to 46% faster than NCCL for buffer size greater than 1MB, as shown in Figure 11(ii) in Appendix C.

7.1.3 ALLREDUCE

ALLREDUCE on DGX-2. As discussed in Section 5.3, TACCL composes REDUCESCATTER with ALLGATHER to implement ALLREDUCE and an algorithm for REDUCESCATTER can be constructed by inverting an ALLGATHER algorithm. Figure 8(i) shows the performance of TACCL algorithms on two DGX-2 nodes. The ALLREDUCE synthesized from the ALLGATHER using *dgx2-sk-2* is $49\% - 6.4\times$ faster than NCCL for buffer sizes ranging from 1KB - 4MB. TACCL's generated algorithms by using other communication sketches like *dgx2-sk-1* are $2\% - 37\%$ faster than NCCL for buffer sizes ranging from 16MB - 256MB. For buffer sizes of 512MB and greater, our algorithms are at most 9% slower than NCCL. This is because NCCL uses the more optimized fused communication instructions (such as receive-reduce-copy-send) in its ALLREDUCE communication which are unavailable in TACCL's lowering. We leave these such further optimizations for future work.

ALLREDUCE on NDv2. These algorithms are based on the ALLGATHER synthesized from the *ndv2-sk-1* sketch and use two versions with 1 and 8 instances. Figure 8(ii) compares them to NCCL on two NDv2 nodes. The single instance TACCL algorithm outperforms NCCL's ALLREDUCE by up to 28% for buffer sizes of up to 1MB, while the 8 instance algorithm outperforms NCCL by $28\% - 2.7\times$ for larger sizes.

On 4 NDv2 nodes, as shown in Figure 11(iii) in Appendix C, the TACCL algorithms are up to 34% faster than NCCL for small buffer sizes and $1.9\times - 2.1\times$ faster than NCCL for larger buffer sizes.

7.2 Impact of Varying Synthesizer Inputs

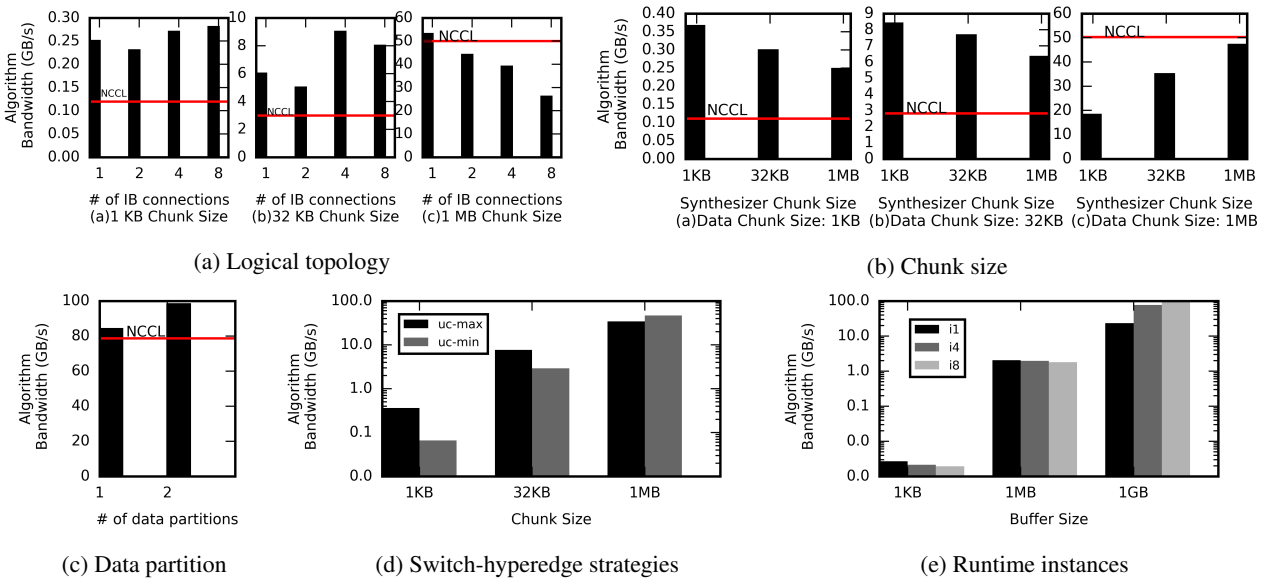
In this section, we explore modifications to communication sketches, as well as the synthesizer hyperparameters and the instances for the lowering, in order to understand their impact on the performance of the synthesized algorithms. Our aim is to demonstrate that the controls offered by TACCL have intuitive effects on the resulting algorithms, which is necessary for effectively communicating user intuition to TACCL.

We present our analysis for the ALLGATHER collective on two Nvidia DGX-2 nodes. Unless mentioned otherwise, we use the following communication sketch as the baseline: same logical topology as *dgx2-sk-1*, chunk size set to 1MB, data partitioning set to 1, and the switch-hyperedge policy set to `uc-max`.

Changing logical topology. We create a logical topology with a dedicated sender and receiver GPU similar to *dgx-sk-1* except we allow a sender to be connected to n different receivers in the other node. Figure 9a shows the algorithm bandwidth of ALLGATHER obtained by varying n, the number of IB connections per GPU, for a fixed chunk size of 1KB, 32KB, and 1MB. For a 1KB chunk size, we found the algorithm that uses 8 IB connections per NIC performs better than algorithms using fewer connections. As the chunk size increases to 32KB and 1MB, the optimal number of IB connections per NIC reduces to 4 and 1, respectively. The benefits of link sharing shrink as the chunk size increases and β-cost starts dominating over the α-cost.

Changing transfer cost using chunk size. We analyze the sensitivity of TACCL's synthesizer to the data size provided in the communication sketch when its algorithms are applied on a communication using a different data size. Figure 9b shows the performance of ALLGATHER algorithm for three different chunk sizes (1KB, 32KB, and 1MB). Algorithms generally perform well for a range of data sizes close to what they have been synthesized for. We recommend trying a small set of nearby sizes to ensure the best performance.

Changing data partitioning. Figure 9c shows the algorithm bandwidth of algorithms generated by partitioning data on



(a) Logical topology

(b) Chunk size

(c) Data partition

(d) Switch-hyperedge strategies

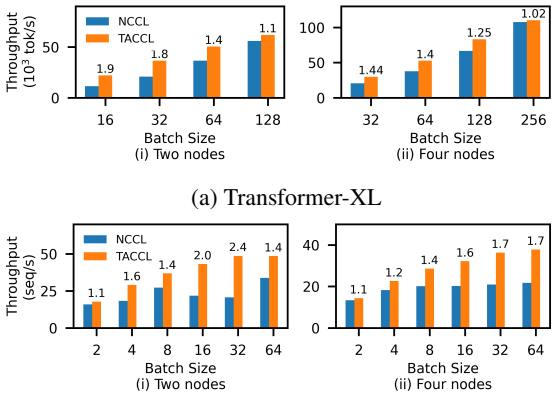
(e) Runtime instances

Figure 9: Algorithm bandwidth of ALLGATHER algorithms on DGX-2 by varying different inputs to TACCL

each GPU into a single or two chunks. We set the switch-hyperedge policy to `uc-min` and fix number of instances to 8. At a large buffer size of 1GB, the algorithm generated for two data chunks utilizes bandwidth better as compared to the algorithm generated for a single data chunk per GPU.

Changing switch-hyperedge policy. Figure 9d shows the algorithm bandwidth for algorithms generated and evaluated for 1KB, 32KB, and 1MB chunks. The algorithm bandwidth is displayed in log-scale. We vary the switch-hyperedge policy between `uc-max` and `uc-min`. For smaller buffer sizes, the `uc-max` configuration performs better than `uc-min`, whereas for larger buffer sizes, `uc-min` performs better than `uc-max`.

Changing number of instances. Figure 9e shows algorithm bandwidth with instances ranging from 1 to 8. The switch-hyperedge policy for these algorithms is set to `uc-min`. Increasing the number of instances improves bandwidth utilization — multiple threadblocks seem to be needed to keep the six NVLinks in a V100 busy. However, a larger number of threadblocks also increases latency, which we suspect is due to unfavorable scheduling of synchronization related memory operations onto the NVLinks at the start of each send. Since latency cost dominates for small buffer sizes, using a large number of instances only increases the latency cost. As the buffer size increases, the bandwidth improvements due to more instances become predominant. Since switch-hyperedge policy and number of instances have a similar relation with chunk sizes, we always run `uc-max` algorithms with a single instance and `uc-min` algorithms with 8 instances.



(a) Transformer-XL

(b) BERT

Figure 10: Training throughput using TACCL's collective algorithms on Transformer-XL and BERT compared against NCCL on 2 and 4 Azure NDv2 nodes. Speedup over NCCL is mentioned on top of the bars.

7.3 End-to-End Training.

We evaluate TACCL on distributed training of two large language models, Transformer-XL [4, 13] and BERT [3, 15], on two (and four) Azure NDv2 nodes, i.e. 16 (and 32) GPUs. Transformer-XL uses data parallelism and whereas BERT uses model parallelism. The typical transfer sizes for ALLREDUCE in Transformer-XL is in the 20 - 40MB range, and for BERT it is about 2MB. Both models communicate with `torch.distributed` and, as explained in Section 6, using TACCL algorithms in them is quite straightforward.

We lower the algorithm synthesized by the synthesizer

AllGather		AlltoAll		AllReduce	
Sketch	Time(s)	Sketch	Time(s)	Sketch	Time(s)
dgx2-sk-1	35.8	*dgx2-sk-2*	92.5	*dgx2-sk-1*	6.1
dgx2-sk-2	11.3	*ndv2-sk-1*	1809.8	*dgx2-sk-2*	127.8
ndv2-sk-1	2.6	*ndv2-sk-2*	8.4	*ndv2-sk-1*	0.3

Table 2: Synthesis time for TACCL algorithms for different collectives using different communication sketches.

into TACCL-EF with 1 and 8 instances, and show the performance of both against NCCL. Figure 10a and Figure 10b show the training throughput obtained by using TACCL's collective algorithms for communication instead of NCCL for Transformer-XL and BERT respectively for different batch sizes. TACCL speeds up training of Transformer-XL by $11\% - 1.94\times$ on 2 nodes and by $2\% - 1.44\times$ on 4 nodes. The speedup for BERT is $12\% - 2.36\times$ on 2 nodes and $7\% - 1.74\times$ on 4 nodes. Depending on the memory available per GPU and on how the batch size affects model accuracy, any of these batch sizes might be chosen for use in practice.

We also use algorithms synthesized by TACCL for ALLTOALL and ALLREDUCE collectives for training an internal Microsoft's mixture-of-experts workload on two NDv2 nodes. The ALLTOALL and ALLREDUCE sizes required for this model are $\approx$ 6MB and $\approx$ 256MB, respectively. TACCL improves the end-to-end throughput of this model by 17%.

7.4 Synthesis Time

Table 2 shows the total time it takes for TACCL to synthesize algorithms for different collectives using some of the communication sketches mentioned in Section 7.1. In most cases synthesis takes from seconds to a few minutes, making it amenable to a human-in-the-loop approach. When synthesizing an ALLTOALL collective using some communication sketches, TACCL's contiguity encoding may take more time in proving the optimality of a feasible solution. We put a time limit of 30 minutes on the contiguity encoding in these cases. The contiguity encoding for sketch *ndv2-sk-1* reaches this timeout, but a feasible solution was already found in 4min 14s. We have also been able to synthesize an ALLGATHER for 80 GPUs (10 NDv2 nodes) in under 8 minutes.

8 Related Work

The MPI standard provides a set of collective communication algorithms that enable efficient distributed computations of interconnected nodes [16]. The HPC community has focused on the efficient implementation of these MPI collective algorithms [40, 50] and demonstrated how to build optimized algorithms for specific interconnects, like mesh, hypercube, or fat-tree [7, 8, 41]. In contrast to TACCL, these prior works assume homogeneous interconnects and are often only focused on bandwidth optimality. Hybrid algorithms [7, 10] combine bandwidth- and latency-optimal algorithms based on input sizes, but only qfor mesh networks.

NCCL [37] is a GPU implementation of a subset of the standard MPI collectives, optimized for NVLINK and Infiniband interconnects. While NCCL uses the topology of GPU connections and NIC placement along with buffer size to decide between two main types of communication algorithms — Ring and Tree, it is agnostic to the exact performance profile of the links, and thus (as we show) is often multiple times slower than TACCL's topology aware collectives.

Recent works like SCCL [9], Blink [51], and Plink [29] specialize algorithms for the underlying topology. SCCL solves an integer programming encoding based on discrete-time values in the form of steps and rounds of the algorithm in order to achieve the pareto-frontier of latency- and bandwidth-optimal algorithms. SCCL is able to synthesize a novel pareto-optimal ALLGATHER algorithm for an Nvidia DGX1 node, but its restrictive formulation constrains it to only only synthesize algorithms for single-node topologies. TACCL on the other hand synthesizes collective algorithms for multi-node topologies. Blink uses a heuristic spanning-tree packing algorithm to maximize bandwidth utilization within a node and a hierarchical approach across. Blink has good performance over NCCL in the case when NCCL cannot create rings spanning all GPUs inside a node. TACCL, on the other hand, outperforms NCCL when using the entire node of GPUs. Plink constructs a logical topology based on bandwidth and latency probes of the physical topology to avoid oversubscribed and congested links and searches for a reasonable clustering of nodes for a two-level hierarchical reduction strategy. Plink builds that hierarchical reduction from known primitives and does not search over the space of possible algorithms.

There are also hierarchical approaches to implement collectives [12, 29, 42, 51]. For example, Horovod [42] implements an ALLREDUCE by a local ReduceScatter, a global ALLREDUCE, and then a local ALLGATHER. These methods do not search over possible algorithms, but instead pick from a known set of decompositions. Concurrent to our work, Ningning et al. [52] use syntax guided synthesis to combine base MPI primitives among a subset of nodes to hierarchically generate larger MPI primitives for the entire network. In contrast, TACCL uses a fine grained approach for algorithm synthesis while using communication sketches for scalability. Combining these two complementary approaches is an interesting opportunity for future work.

Program sketching [24, 47, 49] is a popular technique that has been applied to a variety of problems from synthesizing stencil computations [48], converting hand drawings to images [17] to social media recommendations [11]. Our work builds on this body of work to use sketching to effectively search a large space of communication algorithms.

Lastly, network flow problems have used linear programming to solve routing and scheduling problems for traffic engineering [22, 23, 25, 44, 46] and topology engineering [45].

These techniques, however, cannot be used for generating collective algorithms since communication collectives do not follow all flow properties. Non-source GPUs in a collective can send the same chunk over different links in parallel while having received that chunk only once, which violates an important flow-conservation property used extensively in network flow problem literature. TACCL on the other hand makes use of communication sketches and an encoding relaxation technique to solve a continuous-time integer linear programming that faithfully models communication collectives.

9 Conclusion and Future Work

TACCL is a topology and input-size aware collective communication library for multi-node distributed machine learning training and inference. TACCL uses user-provided communication sketches to guide synthesis of collective algorithms. Using a three-step technique of relaxed routing, heuristic ordering, and contiguity and exact scheduling, TACCL generates efficient collectives for multi-node topologies. We also make some brief observations about TACCL below:

Scalability. TACCL can synthesize algorithms for large-scale nodes - we have been able to synthesize an ALLGATHER algorithm for 8 Azure NDv2 nodes using TACCL in under 5 minutes. As compared to NCCL, this algorithm has up-to $1.7\times$ higher algorithm bandwidth for different data sizes. We also evaluated TACCL's synthesis for 8 Nvidia DGX-2 nodes (128 GPUs) and found a solution in around 11 hours. While TACCL scales to multi-node topologies, the synthesis technique is still based on solving an NP-hard problem that grows exponentially with a quadratic power with scale. As a future work, we would like to scale TACCL further by hierarchically composing synthesized algorithms.

Generality across different topologies. Apart from hierarchical topologies like Nvidia DGX-2 and Azure NDv2, TACCL can also be applied to non-hierarchical topologies like a 2D-Torus. We were able to synthesize an ALLGATHER algorithm for a 2D 6×8 Torus using TACCL. We made use of the symmetry attribute in communication sketches to explore synthesis for this topology. However, the amount of exploration we can do with different communication sketches may be more limited in these cases than for hierarchical topologies.

Exploring communication sketches. Communication sketches have proven effective in narrowing the search space of algorithms. Interestingly, different communication sketches can optimize different ranges of input sizes. Communication sketches reflect the intuition of developers, and by intelligently exploring the space of communication sketches we can obtain a range of collective algorithms with different performance characteristics. Learning an automated controller for exploring communication sketches is an interesting direction for collective algorithm synthesis in the future.

To conclude, TACCL uses the abstraction of communication sketches and a novel problem formulation to generate efficient algorithms for collectives like ALLGATHER, ALLTOALL, and ALLREDUCE. The algorithms thus generated are up-to $6.7\times$ faster than the state-of-the-art NCCL and result in $11\% - 2.4\times$ faster end-to-end training time.

Acknowledgements

We would like to thank our shepherd, Aurojit Panda, the anonymous reviewers at NSDI'23, and the members of the Systems and Storage Lab at UT Austin for their insightful comments and suggestions. This work was partially supported by NSF CAREER #1751277, the UT Austin-Portugal BigHPC project (POCI-01-0247-FEDER-045924), and donations from VMware.

References

[1] GPUDirect RDMA, 2021. https://developer.nvidia.com/gpudirect.

[2] Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[3] Megatron-LM. https://github.com/NVIDIA/Megatron-LM, 2022.

[4] Transformer-XL. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/Transformer-XL, 2022.

[5] Azure ND-series, 2021. https://docs.microsoft.com/en-us/azure/virtual-machines/nd-series.

[6] Azure NDv2-series, 2021. https://docs.microsoft.com/en-us/azure/virtual-machines/ndv2-series.

[7] Michael Barnett, Rick Littlefield, David G Payne, and Robert van de Geijn. Global combine on mesh architectures with wormhole routing. In *[1993] Proceedings Seventh International Parallel Processing Symposium*, pages 156–162. IEEE, 1993.

[8] Shahid H Bokhari and Harry Berryman. Complete exchange on a circuit switched mesh. In *1992 Proceedings Scalable High Performance Computing Conference*, pages 300–301. IEEE Computer Society, 1992.

[9] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming, pages 62–75, 2021.

[10] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[11] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, page 1732–1736, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1:1–1:11, 2019.

[13] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[14] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. Deeplight: Deep lightweight feature interactions for accelerating ctr predictions in ad serving. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, WSDM '21, page 922–930, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[16] Jack Dongarra et al. MPI: A message-passing interface standard version 3.0. *High Performance Computing Center Stuttgart (HLRS)*, 2(5):32, 2013.

[17] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

[18] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[19] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 829–844, 2021.

[20] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[21] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994.

[22] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, August 2013.

[23] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, August 2013.

[24] Jinseong Jeon, Xiaokang Qiu, Jeffrey S Foster, and Armando Solar-Lezama. Jsketch: sketching for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 934–937, 2015.

[25] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *SIGCOMM'14*.

[26] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[27] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020.

[28] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: A rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 41–54, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In *Proceedings of Machine Learning and Systems 2020*, pages 82–97. 2020.

[30] Microsoft SCCL, 2021. https://github.com/microsoft/sccl.

[31] Using deepspeed and megatron to train megatron-turing nlg 530b, the world's largest and most powerful generative language model. https://www.microsoft.com/en-us/research/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/. Accessed October 2021.

[32] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[33] NCCL Tests, 2021. https://github.com/NVIDIA/nccl-tests.

[34] NCCL Tree Algorithm, 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4.

[35] Nvidia DGX Systems, 2021. https://www.nvidia.com/en-us/data-center/dgx-systems/.

[36] Nvidia InfiniBand, 2021. https://www.nvidia.com/en-us/networking/infiniband-adapters/.

[37] Nvidia NCCL, 2021. https://github.com/nvidia/nccl.

[38] Nvidia NVLink and NVSwitch, 2021. https://www.nvidia.com/en-us/data-center/nvlink/.

[39] NVIDIA NVSWITCH The World's Highest-Bandwidth On-Node Switch , 2021. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[40] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[41] David S Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, pages 398–399. IEEE Computer Society, 1991.

[42] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[44] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 201–216. USENIX Association, April 2021.

[45] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-Effective Capacity Provisioning in Wide Area Networks with Shoofly. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 534–546, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. Radwan: Rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 547–560, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008. AAI3353225.

[48] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 167–178, New York, NY, USA, 2007. Association for Computing Machinery.

[49] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 404–415, New York, NY, USA, 2006. Association for Computing Machinery.

[50] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[51] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 172–186, 2020.

[52] Ningning Xie, Tamara Norman, Dominik Grewe, and Dimitrios Vytiniotis. Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning. *CoRR*, abs/2110.10548, 2021.

[53] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pages 8–13, 2020.

Appendix

A Communication Sketch Input

TACCL adopts a user-in-the-loop approach where algorithm designers provide a communication sketch to guide communication algorithm synthesis by TACCL. TACCL's synthesizer takes in a profiled topology provided by TACCL profiler along with a communication sketch provided by a human-in-the-loop. A communication sketch comprises of a logical topology, switch-hyperedge strategy, symmetry information, input size, and other hyperparameters. Listing 1 gives an example of how users can provide a communication sketch input to the TACCL synthesizer. Here, we show an example of the communication sketch *dgx2-sk-1* used in the evaluation to synthesize an ALLGATHER algorithm for 2 Nvidia DGX-2 nodes (each node has 16 GPUs and 8 NICs, every two GPUs in the node share a NIC).

The sketch annotates the NVSwitch in each node and sets a `uc-min` switch-hyperedge strategy. Further, the inter-node sketch fixes the sender and receiver GPUs in a node for inter-node data transfers. In our example, the odd-numbered GPUs sharing a NIC are chosen as senders and the even-numbered GPUs are chosen as receivers for inter-node communication. The user also annotates how the inter-node relay GPUs would split the inter-node bandwidth using a *beta_split* attribute. Since only a single GPU per NIC is chosen in our example to perform inter-node send and similarly receive, the bandwidth is not split. Optionally, the user can also map chunks to sender GPUs so that only mapped GPUs are used for inter-node transfers for the chunk. The *chunk_to_relay_map* attribute defines the parameters for the mapping function. The communication sketch also allows users to play with rotational symmetry for data routing. Given a symmetry offset and a group size, a chunk transfer over a link is set to be equivalent to a rotationally symmetric chunk over a rotationally symmetric link. In our example of the *symmetry_offset* attribute, using $[2,16]$ fixes an intra-node symmetry with an offset of two, and using $[16,32]$ fixes a symmetric data transfer pattern between the two DGX-2 nodes. Hyperparameters like input data partitioning and input size can also be provided via the communication sketch.

Listing 1: Example sketch *dgx2-sk-1* for ALLGATHER

```
{
    // sketch for intra-node policy
    "intranode_sketch": {
        "strategy": "switch",
        "switches":
            [[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]],
        "switch_hyperedge_strategy": ["uc-min"]
    },

    // sketch for communication policy between any
        two nodes
    "internode_sketch": {
        "strategy": "relay",
        "internode_conn": {"1" : [0], "3" : [2], "5"
            : [4], "7" : [6], "9" : [8], "11" :
            [10], "13" : [12], "15" : [14]}, // "i":
            [j1, j2] implies GPU i in a node will
            only send data to GPU j1 and j2 of
            another node
        "beta_split": {"1": 1, "3": 1, "5": 1, "7" :
            1, "9" : 1, "11" : 1, "13" : 1, "15" :
            1}, // "i": n implies inter-node sends
            from a GPU i of a node will use 1/n-th
            of the inter-node bandwidth
        "chunk_to_relay_map": [2,1] // maps chunk to
            a sender relay GPU. [r1,r2] means chunk
            c will be send to another node via GPU
            (rp//r1)*r1 + r2, where rp is the
            precondition GPU for chunk c
    },

    // enforces rotational symmetry.
    // [(o,g), ..]: o is the rotational offset and
        g is the group size for the rotational
        symmetry.
    // : eg. send(c,src,r) == send( (c + o)%g, (src
        + o)%g, (r + o)%g)
    "symmetry_offsets": [[2, 16], [16, 32]],

    "hyperparameters": {
        "input_chunkup": 2, // Data at each GPU is
            partitioned into 2 chunks that can be
            independently routed
        "input_size": "1M"
    }
}
```

B TACCL Synthesizer in Detail

As explained in Section 5, TACCL's synthesizer has routing, heuristic ordering, and contiguity and exact scheduling stages. We provide a detailed description of each of these stages in this section. We first formally introduce some terms that we will use later. Let $\mathcal{C}$ denote the set of chunks that are required to be routed in the algorithm for collective *coll*. Let $\mathcal{R}$ denote the set of GPU ranks involved in *coll*. Let *coll*.precondition and *coll*.postcondition denote the precondition and post-condition of the collective respectively.The tuple $(c,r) \in$ *coll*.precondition, $c \in \mathcal{C}, r \in \mathcal{R}$, if chunk c is present at rank r at the start of the collective. Similarly, the

$(c,r) \in coll$.postcondition if chunk c has to be present at rank r at the end of the collective. Further, let $\mathcal{L}$ denote the set of links, such that $(r1,r2) \in \mathcal{L}, r1 \in \mathcal{R}, r2 \in \mathcal{R}$ if there exists a link from rank $r1$ to rank $r2$ in the logical topology determined by the topology and communication sketch. Let $\mathcal{S}_r^{send}$ denote the set of switched destinations for rank r, such that $dst \in \mathcal{S}_r^{send}$ if link (r,dst) is a part of a switch-hyperedge. Similarly, $\mathcal{S}_r^{recv}$ denotes the set of switched sources for rank r, such that $src \in \mathcal{S}_r^{recv}$ if link (src,r) is a part of a switch-hyperedge. $\alpha(r1,r2)$, $\beta(r1,r2)$ are the alpha and beta costs respectively of the link $(r1,r2) \in \mathcal{L}$. The term $lat(r1,r2)$ is the sum of $\alpha(r1,r2)$ and $\beta(r1,r2)$ cost of the link, which denotes the total transfer cost of a single chunk over link $(r1,r2)$. Table 3 lists the variables that the TACCL's synthesizer solves for. We will describe each variable in detail in this section.

B.1 Routing

The main aim of the routing stage is to give us the path that every chunk takes in the collective. Our objective is to minimize the time (denoted by continuous variable *time*) it takes to reach the post-condition of the collective.

$$\text{Minimize} \quad time \tag{1}$$

The time taken for the collective algorithm is the latest time at which a chunk becomes available on a rank that is in the post-condition of the collective. We use a continuous variable $start[c,r]$ to denote the time that chunk c becomes available on rank r, and end up with the following constraints for *time*

$$time \geq start[c,r] \quad \forall (c,r) \in coll.\text{postcondition} \tag{2}$$

For chunks on ranks that belong to the collective's precondition, we set the start time to zero.

$$start[c,r] = 0 \quad \forall (c,r) \in coll.\text{precondition} \tag{3}$$

We also add correctness constraints in our formulation for routing - chunks are sent from a GPU rank only after they have been received on that rank. We introduce a continuous variable $send[c,src,r]$ to denote the time of sending chunk c from rank src to rank r and add the following constraint to our formulation:

$$send[c,src,r] \geq start[c,src] \quad \forall c \in \mathcal{C} \quad \forall (src,r) \in \mathcal{L} \tag{4}$$

We use a binary variable $is_sent[c,src,r]$ to indicate if chunk c is sent over the link (src,r) in our algorithm. We note that the routing stage does not strictly respect bandwidth constraints of any link - the generated solution may send two chunks simultaneously over a link at the time cost of one chunk. The chunk start time on a rank will be determined only by the chunk send time on the source, independent of other chunk transfers on the link (eq. 5). LHS→RHS in the equation signifies an indicator constraint, i.e., if LHS is 1, RHS will hold.

$$is_sent[c,src,r] \rightarrow start[c,r] = send[c,src,r] + lat(src,r) \quad \forall c \in \mathcal{C} \quad \forall (src,r) \in \mathcal{L} \tag{5}$$

Instead of bandwidth constraints, this encoding uses *relaxed bandwidth constraints*. They are expressed by aggregating the link transfer time of all chunks sent over a link and using it to to lower bound the total time of the algorithm (eq. 6). For switched connections, the total time is lower bounded by the sum of link transfer times of all chunks sent over all switched outgoing links from a source, and also by the sum of link transfer times for chunks received from all incoming links to a destination (eq. 7 and eq. 8).

$$time \geq \sum_{c \in C} (lat(src,r) * is_sent[c,src,r]) \quad \forall (src,r) \in \mathcal{L} \tag{6}$$

$$time \geq \sum_{c \in C} \sum_{dst \in \mathcal{S}_r^{send}} (lat(r,dst) * is_sent[c,r,dst]) \quad \forall r \in \mathcal{S}_{send} \tag{7}$$

$$time \geq \sum_{c \in C} \sum_{src \in \mathcal{S}_r^{recv}} (lat(src,r) * is_sent[c,src,r]) \quad \forall r \in \mathcal{S}_{recv} \tag{8}$$

Based on the communication sketch, we also add constraints for `uc-max` and `uc-min` strategies for switch-hyperedges to maximize and minimize the number of links utilized in a switch respectively. We introduce a new binary variable $is_util[src,r]$ for links (src,r) that are a part of a switch-hyperedge. This variable is 1 if any chunk is sent over link (src,r), and 0 otherwise.(eq. 9 and eq. 10). According to the switch-hyperedge strategy, we add this variable, weighted with a small constant γ, to the objective function (eq. 11). γ is negative for `uc-max` and positive for `uc-min`.

$$is_util[src,r] >= is_sent[c,src,r] \quad \forall c \in \mathcal{C} \forall (src,r) \in \mathcal{L} \tag{9}$$

$$is_util[src,r] <= \sum_{\forall c \in \mathcal{C}} is_sent[c,src,r] \quad \forall (src,r) \in \mathcal{L} \tag{10}$$

$$\text{Minimize} \quad time + \gamma \times \Big(\sum_{(src,r):\text{switched links}} is_util[src,r] \Big) \tag{11}$$

We also add symmetry constraints according to the symmetry offsets provided by user in the communication sketch. For a chunk c and link (src,r), we identify a rotationally symmetric chunk $\hat{c}$ and link $(s\hat{r}c,\hat{r})$ and add the following constraints:

$$start[c,r] = start[\hat{c},\hat{r}] \tag{12}$$

$$send[c,src,r] = send[\hat{c},s\hat{r}c,\hat{r}] \tag{13}$$

$$is_sent[c,src,r] = is_sent[\hat{c},s\hat{r}c,\hat{r}] \tag{14}$$

MILP Variables	Explanation
Routing	
$time$	time spent in the collective algorithm
$start[c,r]$	time at which chunk c becomes available at GPU r
$send[c,src,r]$	time at which chunk c is sent from GPU src to GPU r
$is_sent[c,src,r]$	indicates if chunk c is sent from GPU src to GPU r
$is_util[src,r]$	indicates if any chunk is sent from GPU src to GPU r
Contiguity	
$is_together[c,o,r]$	indicates if chunks c and o are sent to GPU r together from the same source, thus sharing the bandwidth and reducing the latency cost of transfer

Table 3: Variables used in TACCL's MILP formulation. Variables with prefix *is_* are binary variables and others are continuous variables.

Further, for chunks that start on one node and have a final destination on another node, we add inter-node transfer constraints which specify that at least one inter-node link will be used to transfer that chunk.

$$\sum_{(r_1,r_2)\in\mathcal{L}: r_1\in \text{node}_1, r_2\in \text{node}_2} is_sent[c,r_1,r_2] \geq 1 \qquad (15)$$

B.2 Ordering Heuristics

We start the heuristic ordering by determining the paths each chunk takes using the solution of the path encoding. We then consider the first link in every path as a candidate for scheduling a chunk transfer. Using heuristics like *chunk-with-shortest-path-until-now-first* and *chunk-with-longest-path-from-now-first*, we select a path (and thus a chunk) which should be scheduled in this round. We keep a running estimate of link time, which is the earliest time at which a chunk can be scheduled over the link. We also keep a running estimate of chunk time, which is the earliest time at which the next link transfer can be scheduled for a chunk. At the start, the link time for every link is 0 and the chunk time for every chunk is 0. When a path is chosen in the first round, the chunk associated with the path is scheduled to traverse the first link in the path. The link time of that link increases by link latency and chunk time of that chunk increases by link latency. The link candidate from the selected path is also updated to be the next link in the path. For the next rounds, we decide which path's candidate link to schedule next using the tracked link and chunk times along with the scheduling heuristics. This keeps going until we have scheduled a data transfer over all the links in all the paths. We find that the best heuristics differ for architectures with NVLinks and those with NVSwitches, in terms of whether to start selecting links to schedule in the same order as the paths or in the opposite order of the paths. The heuristic ordering has the following three outputs:

- chunk_order(r_1,r_2), an ordered list of chunks transferred along each link (r_1,r_2). If chunk c_1 is present before chunk c_2 in chunk_order(r_1,r_2), it denotes that c_1 is scheduled to be sent before c_2 over link (r_1,r_2).
- switch_send_order(r), an ordering on the chunks sent from a switch source r to any of the switch destinations $dsts$. If (c_1,dst_1) is present before tuple (c_2,dst_2) in switch_send_order(r), it means that a send of c_1 over link (r,dst_1) should be scheduled before a send of chunk c_2 over link (r,dst_2).
- switch_recv_order(r), an ordering on the chunks received on a switch destination r from any of the switch sources $srcs$. If (c_1,src_1) is present before tuple (c_2,src_2) in switch_recv_order(r), it means that a receive of c_1 over link (src_1,r) should be scheduled before a receive of chunk c_2 over link (src_2,r).

B.3 Contiguity and Exact Scheduling

Finally, we describe the formulation for the contiguity and exact scheduling stage. Given the link and switch ordering from the heuristic ordering stage, the aim of this stage is to find the sweet spot in the trade-off between lower link latency by sending multiple data chunks contiguously as a big data chunk and reduced pipelining benefits due to the big data-chunk transfer. We provide the main set of constraints in our formulation below, leaving out other less important constraints.

Our objective is still to minimize the time of the collective and constraints eq. 1-eq. 4 must still hold in this formulation. We add a new binary variable $is_together(c_1,c_2,r)$ for all chunks c_1 and c_2 that are sent over the same link to rank r. If $is_together(c_1,c_2,r)$ is 1, chunks c_1 and c_2 are sent as a single data-chunk over a link to rank r.

$$\begin{aligned} is_together[c,o,r] \rightarrow send[c,src,r] = send[o,src,r] \\ \forall c,o \in \text{chunk_order}(src,r) \quad \forall (src,r)\in\mathcal{L} \end{aligned} \qquad (16)$$

The transfer time of a data chunk c along a link (src,r) will be determined by all other data chunks that it has to travel together with:

$$\begin{aligned} lat[c,src,r] =& \alpha(src,r) + \beta(src,r) * \\ & \Big(\sum_{o\in \text{chunk_order}(src,r)} is_together[c,o,r]\Big) \\ & \forall c \in \text{chunk_order}(src,r) \quad \forall (src,r)\in\mathcal{L} \end{aligned} \qquad (17)$$

$$\begin{aligned} start[c,r] =& send[c,src,r] + lat[c,src,r] \\ & \forall c \in \text{chunk_order}(src,r) \quad \forall (src,r)\in(L) \end{aligned} \qquad (18)$$

We also add strict bandwidth constraints for this formulation, allowing only one data chunk per link transfer time

if the data chunks are not sent contiguously over the link. Let $pos(c, src, r)$ determine the position of chunk c in the chunk_order(src, r), then

$$\begin{aligned} \neg is_together[c, o, r] \rightarrow send[o, src, r] \geq send[c, src, r] \\ +lat[c, src, r] \quad \forall c \in \text{chunk_order}(src, r) \\ \forall o \in \text{chunk_order}(src, r) \\ \text{if} \quad \text{pos}(o, src, r) \geq \text{pos}(c, src, r) \quad \forall (src, r) \in \mathcal{L} \end{aligned} \tag{19}$$

Similarly, we add bandwidth constraints for switch, allowing a source to send data to only one switched destination at a time, and a receiver to receive data from only one switched sender at a time. Let $sw-pos-send(c, r, dst)$ determine the position of tuple (c, dst) in the switch_send_order(r), and let $sw-pos-recv(c, src, r)$ determine the position of tuple (c, src) in the switch_recv_order(r), then,

$$\begin{aligned} send[o, r, dst_o] \geq send[c, r, dst_c] + lat[c, r, dst_c] \\ \forall (c, dst_c) \in \text{switch_send_order}(r) \\ \forall (o, dst_o) \in \text{switch_send_order}(r) \\ \text{if} \quad \text{sw-pos-send}(o, r, dst_o) \geq \text{sw-pos-send}(c, r, dst_c) \\ \forall r \in \mathcal{S}^{send} \end{aligned} \tag{20}$$

$$\begin{aligned} send[o, src_o, r] \geq send[c, src_c, r] + lat[c, src_c, r] \\ \forall (c, src_c) \in \text{switch_recv_order}(r) \\ \forall (o, src_o) \in \text{switch_recv_order}(r) \\ \text{if} \quad \text{sw-pos-recv}(o, src_o, r) \geq \text{sw-pos-recv}(c, src_c, r) \\ \forall r \in \mathcal{S}^{recv} \end{aligned} \tag{21}$$

C Standalone Experiments on Four Azure NDv2 Nodes

Figure 11 shows additional algorithm bandwidth and the speedup over NCCL graphs of TACCL for ALLGATHER, ALLTOALL, and ALLREDUCE on 4-node NDv2 cluster. We synthesize all collectives using the *ndv2-sk-1* communication sketch (see Section 7.1 for details), and lower them using 1 or 8 instances. We plot the best of the two algorithms over different buffer sizes.

TACCL's ALLGATHER algorithms are $10\% - 2.2\times$ faster than NCCL across all buffer sizes. For ALLTOALL, the *ndv2-sk-1* sketch is most effective for large buffer sizes, and helps generate algorithms that are up-to 46% faster than NCCL for buffer size greater than 1MB. TACCL ALLREDUCE algorithms are up-to 34% faster than NCCL for small buffer sizes and $1.9\times - 2.1\times$ faster than NCCL for larger buffer sizes.

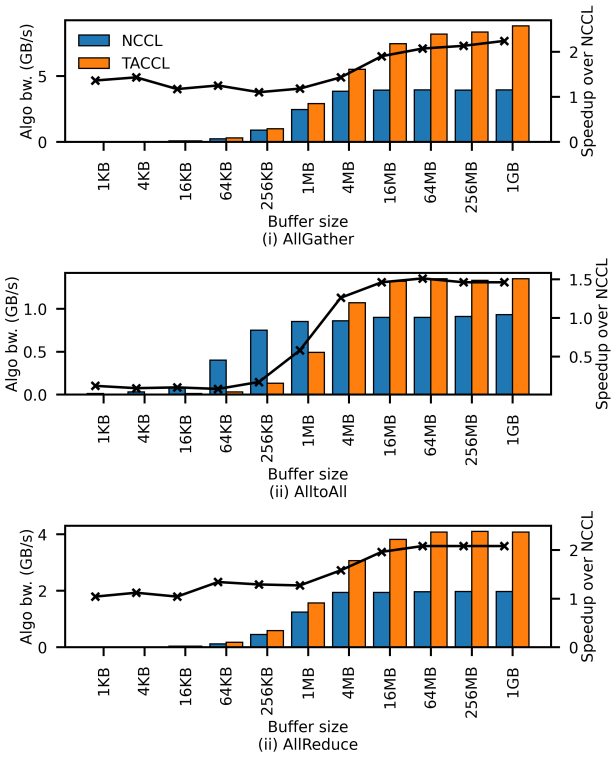


Figure 11: Algorithm bandwidth of TACCL algorithms compared against NCCL (left Y-axis) and their speedup over NCCL (right Y-axis) for ALLGATHER, ALLTOALL, and ALLREDUCE collectives on four NDv2 nodes.

Synthesizing Runtime Programmable Switch Updates

Yiming Qiu Ryan Beckett† Ang Chen

Rice University †*Microsoft*

Abstract

We have witnessed a rapid growth of programmable switch applications, ranging from monitoring to security and offloading. Meanwhile, to safeguard the diverse network behaviors, researchers have developed formal verification techniques for high assurance. As a recent advance, network devices have become runtime programmable, supporting live program changes via partial reconfiguration. However, computing a runtime update plan that provides safety guarantees is a challenging task. FlexPlan is a tool that identifies step-by-step runtime update plans using program synthesis, guaranteeing that each transition state is correct with regard to a user specification and feasible within switch memory constraints. It develops novel, domain-specific techniques for this task, which scale to large, real-world programs with sizable changes.

1 Introduction

Programmable switches accelerate the velocity of change and facilitate innovation [11, 12, 14]. The community's ideal is to develop and deploy new features quickly in the network, without upgrading switch hardware. This has led to two synergistic lines of pursuit. On one hand, researchers have seized this opportunity to develop a slew of *switch applications*, such as monitoring [24, 48], security [31, 54, 55, 58], advanced routing [25, 33, 41], and offloading [27, 28]. On the other hand, to mitigate the potential risks of frequent changes, *formal verification* of switch programs promises to eliminate network bugs and provide high assurance [13, 18, 38, 51]. Combined, the two lines of work pave the way toward a featureful and reliable network infrastructure.

As of late, network programmability is at a new inflection point. Whereas traditional P4 programmable switches [10] require offline procedures for program updates (e.g., draining user traffic, reflashing the data plane with a new program, and undraining traffic), *runtime programmable switches* [4, 19, 52, 53] increase the benefits of programmability even further by supporting live program changes. Upon an update request (e.g., due to tenant requirements or infrastructure upgrades), the switch program is modified online to effect the change, by incrementally adding and removing match/action tables in a running program based on the "delta" [19, 53]. Compared to the approach of recompiling the changed program and reflashing the data plane from scratch, runtime updates enable rapid deployment of new features. Runtime programmability has become available in a variety of targets [19, 52, 53], including commercial off-the-shelf switch ASICs [3, 5, 9].

However, runtime programmability introduces another layer of correctness concerns. It is known that live network updates come with risks—they result in intermediate states with different behaviors from the initial and final networks [29, 44, 45]. Even in a traditional network, updates must be carefully staged to ensure that the transition is free of error [36]. This risk and the challenges it raises are only magnified in programmable switch updates. A step-by-step update to a deployed P4 program, while the switch is in use, could expose partial and potentially unsafe program snapshots to user traffic. For instance, if not careful, an old ACL table may have been removed before the updated ACL is installed, leading to a transient program snapshot without access control. Although we could attempt to make all changes in a single step and avoid intermediate states, in practice, this is only feasible for minuscule updates. The "delta" corresponding to the update needs to be prepared in the switch scratch memory before activated [6, 53]; thus, this leads to a resource utilization peak that may exceed the available memory. A large delta often needs to be broken down into smaller steps, where each step prepares and applies only a fraction of the change [53].

To safeguard runtime network updates, we must again rely on formal reasoning techniques. Existing work in runtime programmability either asks the user to prescribe step-by-step updates [19] or employs algorithms that do not provide semantic guarantees [53]; thus, they do not offer the same level of assurance. Existing verification work, on the other hand, focuses on certifying the correctness of a single P4 program [18, 38, 51], but cannot help *identify a correct-by-construction sequence* for program transition. Thus, these verification techniques cannot guarantee the correctness of intermediate program snapshots during a transition, and bugs could be unwittingly introduced into the network. We believe that customizing *program synthesis* techniques to this new problem domain will bear much fruit. If we can formally synthesize a *safe* (i.e., conforming to a specification) and *feasible* (i.e., within switch resource limits) transition plan, then we can be confident about its effect on the network.

We develop such a tool called FlexPlan. It takes in a P4 program with annotated changes and a user specification, and produces such a transition plan. FlexPlan draws inspirations from a powerful approach to program synthesis—CEGIS, or counterexample-guided inductive synthesis [47]—and develops a variety of domain-specific techniques for our problem at hand. At a high level, the CEGIS algorithm navigates the search space by iterating between a "proposal" phase, which suggests potentially correct programs (e.g., transition plans), and a "verification" phase, which proves or disproves the

proposals. The verification phase also produces counterexamples (i.e., packets) upon failure, which are learned by the next proposal to guide the search. CEGIS has found success in many synthesis problems [17] and more recently, in networking [13, 22, 59]. However, runtime programmable switch updates raise distinct challenges that require novel designs.

The first challenge stems from the fact that FlexPlan needs to synthesize not just one single program but a *sequence* of program snapshots; further, each snapshot must successively modify the previous one safely and stay within the resource constraints. To cast this into a CEGIS framework, we develop two concise encodings that articulate the essence of this synthesis. A *version sketch* is derived from the switch program with annotated changes, adding version variables at change sites to represent the change progress. This building block provides a uniform representation for any intermediate program state that could appear in the transition. Further, a *sequence sketch* concatenates multiple version sketches, while constraining them to modify each other and make progress toward the final program. The synthesis target, therefore, is correct assignments to the version variables across snapshots—these are the program "holes" in the verbiage of the *program sketching* [47] synthesis framework.

The scalability bottlenecks of this synthesis represent the second challenge. Existing verification efforts are already challenged by the large SMT formulas produced by complex switch programs [38, 50, 51], but FlexPlan needs to reason about stacks of these formulas in each proposal/verification phase. To accelerate the synthesis, we develop two domain-specific techniques to shrink the problem sizes whenever possible—to as close as that of single program snapshots. *Snapshot learning* extracts insights about the synthesis from a single snapshot, and generalizes that knowledge to all subsequent snapshots. *Snapshot verification* shatters a proposed transition sequence into individual snapshots for divide-and-conquer. We arrange these techniques carefully to ensure that reasoning about the safety of a sequence from snapshot-based properties still produces a sound analysis.

Finally, we also leverage a unique property in the FlexPlan synthesis problem—its *diagnosability*—to perform introspection into the synthesis process. For a traditional CEGIS problem, the synthesis tool cannot know beforehand whether or not a correct solution exists. FlexPlan, however, can check the initial and final programs against the safety specification to see whether or not some safe transitions exist. If both programs are correct, then a safe transition must exist; a safe transition may not be feasible, however, due to resource constraints. To check feasibility, FlexPlan incrementally grows the transition sequence length—so that each step in the transition sequence makes smaller and smaller changes to approach feasibility—while performing another introspection to decide when to stop trying longer sequences. Finally, when FlexPlan concludes that no safe transition is feasible under the current switch headroom, it introspects on how much resource release is needed for feasibility, as another assistance to the operator.

We prototype FlexPlan [2] and show that it scales to real-world switch programs and sizable changes, and supports a rich set of safety properties including but going beyond those in existing work [53]. With FlexPlan, operaters can synthesize transition plans quickly and automatically (e.g., within a few minutes for sizable changes to switch.p4), while being assured of the correctness of the transition process.

2 Motivation

Analogues of our motivation can be found in existing work on OpenFlow network updates, summarized as follows: Network changes are a constant [23], but they often come with risks [36] due to intermediate states during transition [21, 45]. Rigorous approaches are needed to safeguard against transient disruption [44] to satisfy security requirements and stringent service-level objectives [16]. Ensuring transactional updates at each step [44], and formally guaranteeing the correctness of an update plan [39] is essential. These arguments hold true still, and are further amplified, for P4 programmable networks.

2.1 Runtime programmable switch updates

Programmable switches enable new network features to be quickly developed in-the-field [24, 27, 28, 35, 48, 56]; however, in earlier designs, *deploying* new features to the switch was an intrusive process. To update the switch program (e.g., add, remove, or modify a feature), the traditional approach was to completely recompile the changed program and reflash the data plane [60]. This results in device disruption, so program updates had to be conducted offline—user traffic is drained from the device and diverted elsewhere in the network, after which the switch is re-imaged, and finally, reactivated again.

Recognizing its cumbersomeness and risk for downtime, researchers and practitioners have made a concerted effort toward *runtime programmability* [4, 19, 52, 53, 57]. That is, switch programs are updated using partial reconfiguration without taking down the device for maintenance. Since P4 programs have modular table boundaries, runtime reconfigurations on specific match/action tables and their control flow logic need not disrupt other parts of the program. A feature update can be decomposed into a series of table and branch changes to transform a deployed program to a desired state.

Runtime programmability is not just an academic ideal. In response to the perennial call for both "feature velocity and cloud availability" [16], major switch vendors have embraced this trend with ASIC support. Nvidia's Spectrum switches [53] and Broadcom's Trident [9] and Jericho [3] switches are commercially available off-the-shelf, and academic prototypes [19, 52] are also exploring this design. Use cases of runtime programmable switches include real-time security defense [53], multi-tenancy [52], adaptive telemetry [19], where live switch updates afford higher flexibility not found in earlier programmable switches.

2.2 A motivating example

We will consider a simple example to illustrate the flexibility of runtime switch updates as well as the challenges they raise. The following program snippet uses `@add` and `@del` annotations to demarcate change boundaries in a control block:

```
1  /* Ex1: ipv4_ipv6 */
2  control ingress {
3      apply {
4          if (ipv4.isValid()) {
5              @del acl_v4.apply();
6              @add nat_acl_v4.apply();
7          } else if (ipv6.isValid()) {
8              @del acl_v6.apply();
9              @add nat_acl_v6.apply();
10         }
11         @add stats.apply();
12 }}
```

We remove two older versions of ACL tables for both IPv4 and IPv6 (Lines 5+8), and add two new tables that perform both NAT and ACL (Lines 6+9); we also add a statistics table for monitoring (Line 11). Since P4 is a target-independent language, operators can specify a desired change with such annotations without worrying about hardware details. Indeed, the mechanisms for implementing a live update depend upon the underlying switch platforms [19, 53]:

- FlexCore [53] relies on pointer swaps to achieve transactions, adding and removing a group of tables atomically. In our example above, we can first add `nat_acl_v4`, `nat_acl_v6`, and `stats` to scratch memory, and then use a transaction to make them visible to network traffic, while deleting the two older tables `acl_v4` and `acl_v6`.
- rP4 [19] also adds and removes MA tables leveraging scratch memory, but it relies on temporarily pausing traffic to achieve transactional effects. Before making the update, packets are paused and stored in a front buffer, and the respective tables are modified to effect the change. Buffered packets are then let out into the pipeline.

The careful reader may have noticed that this update appears to have completed within a single transaction, so no intermediate states are exposed. However, this is because we have yet to consider the resource constraints of the switch hardware. Switches have severe memory capacity bottlenecks, and they are easily packed to the brim with large MA tables [32, 53]. Thus, this logically simple update may be physically infeasible if the switch memory has high utilization.

The potentially infeasible operation is preparing the three new tables (`nat_acl_v4`, `nat_acl_v6`, and `stats`) in scratch memory before their old counterparts have been deleted. Assume without loss of generality that every MA table has the same size (say, of U, one unit of table entries), then the net resource increase after the change is only U. However, preparing the transaction causes a resource peak of 5×U, which might exceed the available switch headroom. One workaround is to ensure that the switch always has a low utilization [6], but this is obviously undesirable. Thus, recent work [53] proposes to break a larger update into multiple smaller batches to reduce the resource peak. For instance, if we first add `nat_acl_v4` and delete `acl_v4` in a transaction, it only requires 3×U headroom; the second transaction adds `nat_acl_v6` and deletes `acl_v6`, also within 3×U headroom; a final transaction adds the `stats` table, again within 3×U headroom. As tradeoff, after the first transaction, IPv4 traffic is processed with new tables for NAT and ACL, whereas IPv6 traffic is still processed with the old, and it only encounters new tables after the second transaction; further, the statistics table is only applied after the third transaction completes. Nevertheless, this may still be a reasonable sacrifice in order to achieve a feasible update, as long as the intermediate states are "well-behaved."

2.3 Computing a safe and feasible transition

It is far from clear, however, how to conjure up a transition plan for a desired change. rP4 [19] relies on the user to supply this plan, and FlexCore [53] algorithms only analyze whether changes are "reachable" to each other in the table graph. Neither represents a formal approach that can provide semantic guarantees on the transitional behavior. In general, the notion of correctness is scenario-specific and should be encoded in a user specification in a granular manner, going beyond the three fixed definitions in FlexCore [53], summarized below:

- *Program consistency:* Only one-step updates without intermediate states are allowed.
- *Element consistency:* Intermediate states are acceptable as long as reachable regions (e.g., changes that eventually reach the same table) are changed together atomically.
- *Execution consistency:* Reachable tables can be changed independently as long as no packets will mix them.

This cannot, for instance, capture user intention on "traffic classes" (e.g., IPv4 vs. IPv6); nor can it support more granular correctness definitions (e.g., for any intermediate state, packets must go through an ACL table, or packets must be sent to the same outgoing port). For some cases, the three fixed consistency definitions conflate into the same. For the above change, execution and element consistency will find the same plan as program consistency, because all changes eventually reach the `stats` table, forcing a 5×U peak despite the desire to relax the requirements for a feasible update.

2.4 FlexPlan: A program synthesis perspective

We believe that a principled solution should instead rest upon a firmer foundation, grounded in formal synthesis. Such a solution would satisfy three key goals:

- *Automated:* Beyond expressing a desired property, human reasoning is not required to identify a plan.
- *Completeness:* If a safe and feasible transition exists, we will guarantee to find it.

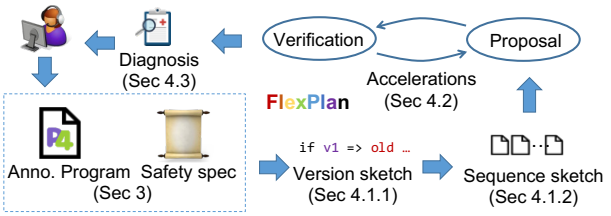


Figure 1: FlexPlan and its key techniques.

- *Soundness:* Once we output a transition, it is guaranteed to be safe and feasible.

The FlexPlan approach. We formulate this problem as follows. Given a P4 program p with a set of change annotations A, a safety specification ϕ, and resource headroom δ, identify a transition sequence $\{p_1 \to p_2 \to \cdots \to p_t\}$, where p_i's are the intermediate program snapshots, and at the end of the sequence, all changes in A have been applied. Further, we require that a) each p_i satisfies ϕ, and b) each transition step $p_i \to p_{i+1}$ makes positive progress and stays within the resource headroom, initialized to δ. Our synthesis relies on a CEGIS approach (cf. [46, 47] for more background): In each iteration, the proposal phase generates a candidate transition sequence that satisfies ϕ and δ on the current set of counterexamples (that is, packets), which is initialized to the empty set. The verification phase will try to extract a new counterexample packet that causes violation, which will be consumed by the proposal phase in the next iteration, and so on. This continues until no counterexample packets can be generated—in which case we have a correct sequence—or until no candidate sequences can be generated—in which case no update plan exists. Figure 1 illustrates the workflow of FlexPlan, with several key milestones in Sections 3 and 4.

3 Specifying Safe Updates

Users provide a P4 program with annotated changes, as well as a specification to constrain intermediate states.

Update annotations. FlexPlan provides three intuitive annotation primitives for users to express a desired update: `@add`, `@del`, and `@mod`. A source P4 program can be annotated with these primitives, where each annotation site captures a set of changes. In Section 2.2, we have already provided an annotated program in this syntax, and here we used another example to describe several other aspects of the annotations.

```
1  /* Ex2: acl_ecmp_flowlet */
2  control ingress {
3      apply {
4          //modify acl into nat_acl
5          if (ipv4.isValid()) {
6              @mod acl.apply(), nat_acl.apply();
7          }
8          //add ECMP, delete flowlet switching
9          @del { if (ipv6.isValid()) flowlet.apply(); }
10         @add ecmp.apply();
11 }}
```

First, the syntax is similar to "P4 annotations" [7] in the P4 language standard. Our annotations target the common intersection of the reconfiguration primitives in existing work [19, 53]—an annotation may specify individual table updates (e.g., Line 10) or a code block update (e.g., tables with their control flow, as in Line 9). The hardware mechanisms are abstracted away from the annotations, but are assumed to provide atomicity for each change annotation, as in recent switches [19, 53]. The `@mod` primitive achieves similar effects as `@del`+`@add`, but `@mod` must complete in a single atomic step whereas the latter could occur in two separate steps. The entire update finishes when all annotations have been applied.

Specification language. Specifying resource constraints is as simple as providing a number δ that denotes the current switch headroom. Thus, our specification language focuses on the *consistency* properties, which constrain the relation between a program snapshot and the initial and final programs. FlexPlan refines the fixed consistency levels in existing work [53] in two ways: (1) a consistency property may refer to specific *traffic classes*, and (2) new consistency levels can be *programmatically* defined. Consider the following examples.

S1: Execution consistency for IPv4 traffic that hits ACL. Any IPv4 packet that hits the `acl` table must not mix old and new code blocks in any intermediate state. However, two IPv4 packets traversing different execution paths in the program do not need to use the same program version—e.g., TCP traffic may be processed by old code blocks, but UDP traffic by the new. We do not constrain the behaviors of other traffic classes.

```
1  specification {
2    // create new ghost variables for the program
3    // these are used for verification only
4    ghost bit<1> sawOld = false;
5    ghost bit<1> sawNew = false;
6    ghost bit<1> acl_hit = false;
7    // update ghost state when tables are applied
8    @old => { sawOld = true; }
9    @new => { sawNew = true; }
10   @hit('acl') => { acl_hit = true; }
11   // define: no path mixes old and new nodes
12   // $cur: the current/transitional program state
13   execution_consistency_ipv4 = {
14     $cur.in.ipv4.isValid() & $cur.eg.acl_hit =>
15       !($cur.eg.sawOld && $cur.eg.sawNew);
16   }
17   assert execution_consistency_ipv4;
18 }
```

Lines 4-6 define "ghost variables" that track meta-level properties of a program execution—specifically, whether a packet has encountered an old code block, a new block, or the ACL table, respectively. Lines 8-10 describe how ghost variables should be updated for an execution: whenever a packet triggers an old code block, a new block, or a table named `acl`, assign the respective ghost variable to be true. Lines 13-17 are the consistency assertion, where `$cur` represents a packet traversing the current program snapshot. If such a packet contains a valid IPv4 header when it arrives at the ingress, and

if it has hit the ACL table before it exits the egress, then we assert that it should not be processed by a mix of code blocks.

S2: Field consistency for egress_spec. We define a new consistency level that only constrains the processing outcomes of specific header fields. The following example specifies that any intermediate states should preserve the processing outcome for the packet's egress port—i.e., a packet should either go to the same port as the old program or as the new one. The `$new` and `$old` variables denote two packets traversing the old and new programs, respectively.

```
specification {
  // preserve processing outcome of egress_spec
  field_consistency_espec = {
    $cur.in == $old.in == $new.in =>
      ($cur.eg.espec == $new.eg.espec ||
       $cur.eg.espec == $old.eg.espec);
  }
  assert field_consistency_espec;
}
```

S3: Program consistency for TCP traffic: We require that TCP packets must not encounter any intermediate state. The `all_old` assertion states that if at the ingress a packet carries a valid TCP header, then at the egress it must only have been processed by old tables. Analogously, the `all_new` assertion states the opposite. Their disjunction implies that TCP traffic will only be processed by one version of the program. The primary difference between this specification and S1 is that execution consistency only constrains the behaviors of each individual packet, whereas program consistency constrains the behaviors across all packets of a certain kind (e.g., TCP).

```
specification {
  // same ghost variables as before
  ghost bit<1> sawOld = false;
  ghost bit<1> sawNew = false;
  @old => { sawOld = true; }
  @new => { sawNew = true; }
  // define whether all packets use the old program
  all_old = {
    $cur.in.tcp.isValid => !$cur.eg.sawNew;
  }
  // define whether all packets use the new program
  all_new = {
    $cur.in.tcp.isValid => !$cur.eg.sawOld;
  }
  // all packets use old program or all use new
  assert all_old || all_new;
}
```

These granular consistency levels require program semantic analysis and cannot be captured by fixed definitions [53]. We describe several more examples in Appendix 9.1 and summarize them in Table 1. Figure 2 presents the grammar of the specification language. Like existing work [18, 50, 51], the specifications are eventually translated into assertions in the source P4 program. Appendix 9.2 shows one such translation.

spec	::=	**specification**{*stmt*∗}	*Specification*
stmt	::=	*gvar*∗	*Ghost vars*
	|	*instr*∗	*Instrumentation*
	|	*property*∗	*Property*
	|	*assert*∗	*Assertion*
gvar	::=	**ghost bit** $< n >$ *gv*	*Ghost vars*
gexpr	::=	**$cur** \| **$old** | **$new**	*Network version*
	|	*gexpr.field*	*Field dereference*
	|	*gexpr* + *gexpr*	*Addition*
	|	...	*Other expr*
instr	::=	*label* => *assignment*∗	*Ghost update*
assignment	::=	*gv* = *gexpr*	*Assignment*
label	::=	**@new** | **@old** | **@hit**	*Annotation*
property	::=	*name* = {*gexpr*∗}	*Consistency*
assert	::=	**assert** *name*	*Assertion*

Figure 2: FlexPlan specification language grammar.

Specifications	LoC
S1. Execution consistency for IPv4 [53]	13
S2. Field consistency for egress_spec	8
S3. Program consistency for TCP [53]	13
S4. Element consistency for ACL [53]	15
S5. Table consistency for ECMP	10
S6. VLAN table access [51]	8
S7. Correct TTL decrement [51]	6

Table 1: FlexPlan supports granular consistency specifications that go beyond existing work [53] (e.g., S1-S5). Although our primary focus is consistency, FlexPlan can also support general program snapshot correctness (e.g., S6-S7) guarantees addressed by existing verification work [18, 51].

4 Update Plan Synthesis

Next, we describe how FlexPlan synthesizes an update plan from the annotated program and specification. We denote these inputs as $\langle p_{[A]}, \phi, \delta \rangle$, where $p_{[A]}$ is an annotated P4 program with a set of change sites $A = \{a_1, \cdots, a_k\}$, and ϕ and δ are the safety and resource constraints, respectively. FlexPlan outputs an update sequence $s = \{p_{old} = p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_t = p_{new}\}$, where two special states p_{old} and p_{new} represent the initial and final programs, respectively. We ask that each intermediate state p_i must be safe (i.e., satisfying ϕ) and each transition from $p_i \rightarrow p_{i+1}$ is feasible (i.e., the resource spike for this transition stays within δ_i, as computed from the initial headroom δ after applying transitions before p_i). If no such s can be found, FlexPlan outputs diagnostic information on whether the safety or feasibility constraints have caused the failure, and in the latter case, it analyzes how much resource release would enable a feasible synthesis.

4.1 Synthesizing a program sequence

Our CEGIS formulation uses *program sketching* [47], a classic framework for synthesis. This formulation views the synthesis task as filling "holes" in an incomplete program (i.e., a "sketch"), and it has been successfully applied to network

```
1  control ingress {
2      apply {
3          /* annotation site 1: acl->nat_acl */
4          if(ipv4.isValid()) {
5              if (! vsk.v1) { // @mod
6                  acl.apply();
7              } else {
8                  nat_acl.apply();
9              }
10         }
11         /* annotation site 2: delete flowlet */
12         if(! vsk.v2) {      // @del
13              if (ipv6.isValid())
14                  flowlet.apply();
15         }
16         /* annotation site 3: add ecmp */
17         if(vsk.v3) {    // @add
18             ecmp.apply();
19         }
20 }}
```

Figure 3: A version sketch that is derived from the `acl_ecmp_flowlet` example in Section 3. Version variables `vsk.v` are the sketch holes. The safety specification will also be instrumented into the version sketch (cf. Appendix 9.2)

programs [13, 22]. However, the objective in FlexPlan differs from existing work as it must identify a correct *sequence* of program snapshots that successively build upon each other toward the final program. To cast this into the CEGIS framework, we develop two novel encodings: a *version sketch* to represent any valid program snapshot during transition, and a *sequence sketch* to string together multiple version sketches and constrain them to make progress toward our final state.

4.1.1 Version sketch: Encoding program snapshots

The version sketch is a uniform and expressive encoding that can capture any transitional program snapshots derived from $p_{[A]}$. As its name suggests, it introduces a "version variable" v_i at the annotation site $a_i \in A$. The annotation is then substituted with two version control branches guarded by v_i. Without loss of generality, consider a_i=`{@mod` $s_i \rightarrow t_i$`}`, a modification from s_i to t_i. It will be transformed into `if (`v_i`) then {`t_i`} else {`s_i`}`, or simply `ite(`v_i, t_i, s_i`)`. That is, a version sketch with v_i turned on encodes a program snapshot where annotation a_i has been applied; on the other hand, a version sketch with v_i turned off represents a snapshot where the change a_i is yet to be applied. The `@add` and `@del` annotations are handled analogously, with one of the branches controlling an empty statement: `ite(`$v_i, t_i, noop$`)` for adding t_i and `ite(`$v_i, noop, s_i$`)` for deleting s_i. Across all annotation sites, by turning version variables on or off, the resulting snapshot seamlessly reflects any combination of applied changes.

Figure 3 shows the version sketch for `acl_ecmp_flowlet`, where version variables are added to the input program as instrumentations. From this instrumented program, FlexPlan derives an SMT formula, where ξ represents the unchanged components without annotations, and the i-th `ite` formula represents annotation a_i. Constraining the version sketch with the safety specification would give an SMT encoding:

$$\texttt{vsk}\,(\xi, \bigwedge_{i=0}^{k} \texttt{ite}(v_i, t_i, s_i)) \;\wedge\; \phi$$

FlexPlan obtains SMT formulas in a similar way as existing work [18]. ϕ is instrumented into the version sketch. Then, FlexPlan converts all statements into static single assignment form and SMT formulas. It then computes the weakest preconditions based on the control flow logic.

The version sketch is expressive enough to encode any intermediate snapshot and constrain its safety with ϕ. However, it cannot capture the resource constraint δ, because it is a *sequence* property defined over a series of snapshots and their relations. As a version sketch only represents an individual snapshot, it cannot easily reason about end-to-end feasibility for a snapshot sequence. By itself, it only enables an awkward workaround—start with an empty version sketch, synthesize a safe snapshot as its immediate next step, and iterate based on the new snapshot. More concretely, one could ask that the next snapshot must fill more holes than the current one, while staying safe with respect to ϕ and within the current headroom δ. This would result in a new snapshot where a subset (but likely not all) of version variables have been turned on. This serves as the new "initial" state for another synthesis, until the final state has been reached. However, this results in a difficult search process, as it can only be guided with some greedy heuristics—e.g., maximizing the progress for each step by filling as many holes as possible, or minimizing the resource spike for each step while ensuring some progress. Either way, the lack of a global view could corner the search into a difficult or infeasible state (e.g., no more headroom), where it must backtrack and probe again in the very large search space—all possible permutations of change annotations, together with all possible combinations of adjacent changes in each permutation. This greedy synthesis also cannot easily conclude that no feasible solution exists.

4.1.2 Zooming in on resource constraints

Thus, resource constraints must be encoded explicitly to enable a cross-snapshot, end-to-end synthesis. Recall that δ denotes the initial switch headroom, which is obtained by subtracting the total table sizes of p_{old} from the overall switch memory. From there, each transition from one snapshot `vsk` to the next `vsk`$'$ adds and removes some tables—thus, we must keep track of the changes to δ and two metrics "spike" and "release" at each transition. Consider the version variable v_1 in Figure 3, which modifies an `acl` table (2Mb) into a `nat_acl` table (3Mb). To achieve atomicity, this is done by first adding `nat_acl` in switch memory, resulting in a transient resource spike of 3Mb, and then deleting `acl` and freeing 2Mb in the same transaction. We record this as $\texttt{add}_1 = 3$ and $\texttt{del}_1 = 2$ for turning on v_1. Thus, across all v_i we define the *spike*:

$$\texttt{vsk}'\texttt{.spike} = \sum_i \texttt{ite}(\texttt{vsk}'.v_i \wedge \neg \texttt{vsk}.v_i,\ \texttt{add}_i,\ 0)$$

That is, if the transition from vsk to vsk′ has turned on v_i, then the transient resource spike increases by add_i; otherwise, if v_i is not changed in this step, it does not contribute to the spike. We can therefore define the resource *release* after the transaction completes and the spike comes down:

$$\texttt{vsk}'\texttt{.rel} = \sum_i \texttt{ite(vsk}'\texttt{.}v_i \wedge \neg \texttt{vsk.}v_i\texttt{, del}_i \texttt{ - add}_i\texttt{, 0)}$$

In our example, the net release is $\texttt{del}_1\texttt{-add}_1 = -1$. This reflects in the overall headroom update to δ after this step:

$$\texttt{vsk}'\texttt{.headroom = vsk.headroom + vsk}'\texttt{.rel}$$

The feasibility constraint can therefore be stated as $\texttt{vsk.headroom} \geq \texttt{vsk}'\texttt{.spike}$ for any two adjacent snapshots, across the entire transition sequence from p_{old} to p_{new}.

4.1.3 Sequence sketch: Encoding a transition plan

Based on the above resource constraint analysis, we develop a sequence sketch encoding that enables an end-to-end CEGIS. A sequence sketch ssk is the conjunction of t version sketches $\{vsk_1, vsk_2, \cdots, vsk_t\}$ as well as their sequential relations to each other. $\texttt{vsk}_1$ and $\texttt{vsk}_t$ encode the initial and final programs, respectively, and other states represent the transitions.

$$\forall i \ \neg \texttt{vsk}_1\texttt{.}v_i \quad \text{; initial program } p_{old}$$
$$\forall i \ \texttt{vsk}_t\texttt{.}v_i \quad \text{; final program } p_{new}$$
$$\forall j \ (\forall i \ \texttt{vsk}_{j-1}\texttt{.}v_i \implies \texttt{vsk}_j\texttt{.}v_i) \quad \text{; progress}$$
$$\forall j \quad \texttt{SpikeAndHeadroom(vsk}_{j-1}\texttt{, vsk}_j\texttt{)} \quad \text{; feasibility}$$

That is, all version variables in the initial sketch are turned off, equating it to p_{old}; all variables in the final sketch are turned on, resulting in p_{new}. A later sketch in the sequence must monotonically advance the version variables to make progress toward the final state. Furthermore, each transition's spike must be feasible within its current headroom (Section 4.1.2).

Sequence synthesis. This encoding enables an end-to-end CEGIS by filling the ssk holes (i.e., all v variables in all version sketches vsk) in a way that satisfies ϕ for all snapshots and δ across all adjacent snapshots. The sketch holes are $\texttt{H} \in \mathbb{B}^{k \times t}$, a two-dimensional matrix of binary variables.

$$\texttt{H}_{k\times t} = \begin{bmatrix} vsk_1.v_1 & vsk_1.v_2 & \cdots & vsk_1.v_k \\ vsk_2.v_1 & vsk_2.v_2 & \cdots & vsk_2.v_k \\ \vdots & \vdots & \vdots & \vdots \\ vsk_t.v_1 & vsk_t.v_2 & \cdots & vsk_t.v_k \end{bmatrix}$$

In this matrix, k is the number of annotations, and t is the number of transitional states; and we will synthesize all holes in an end-to-end CEGIS, as shown in Figure 4. The proposal phase (Lines 7-11) identifies a potentially correct program (i.e., values in H) by solving for H that exhibits correct behaviors on all counterexamples collected so far (Line 9). The verification phase strengthens the check to test against the full specification (Lines 12-17). If no further violations are found, we have obtained a correct solution; otherwise, counterexamples are added to ce_set and we continue with a new proposal. The power of CEGIS lies in the fact that with more counterexamples, SMT solvers learn from violations and eliminate entire classes of proposals in the search. Notice also that at Line 2, we iteratively deepen the search based on the sequence sketch length, so it does not need to reason about a larger problem instance unless absolutely necessary.

```
1:  function SequenceCegis(ssk, ϕ, δ)
2:      for t = 1..k do //Iteratively increase seq length ◁ Opt-Diag
3:          ssk.H ← Rand(𝔹^{k×t}) //Init w/ random H
4:          ce_set ← ∅   //No counterexample so far
5:          // Next, enter main CEGIS loop          ◁ Opt-SnapL
6:          while ¬ Timeout do
7:              // Proposal: Identify candidate H=h
8:              ssk.H ← Symbolic(𝕏^{k×t})
9:              {ssk.H := h} ← SMTSolve(ssk, ϕ, δ, ce_set)
10:             if ssk.H == ∅ then   // No solution exists, t++
11:                 break
12:             // Verification: Verify H=h
13:             ce ← SMTVerify(ssk.H, ϕ, δ)      ◁ Opt-SnapV
14:             if ce ≠ ∅ then // Obtain counterexample
15:                 ce_set ← ce_set ∪ {ce}
16:             else  // Verifies, solution found!
17:                 return ssk.H
```

Figure 4: The end-to-end CEGIS algorithm on the sequence sketch. Later subsections will further develop three optimization techniques, labeled as 'Opt-', to scale this analysis.

In the acl_ecmp_flowlet example (Figure 3), suppose that we require program consistency for IPv4 and that the current headroom is 1Mb. A correct ssk could give a two-step transition denoted by:

$$\texttt{H}_{3\times3} = \begin{bmatrix} \texttt{F} & \texttt{F} & \texttt{F} \\ \texttt{F} & \texttt{T} & \texttt{F} \\ \texttt{T} & \texttt{T} & \texttt{T} \end{bmatrix}$$

The first transition deletes flowlet (3Mb) for IPv6 traffic. This causes a resource spike of 0Mb and a release of 3Mb; and the headroom becomes 4Mb after this transaction. Next, the second transition modifies acl (2Mb) into nat_acl (3Mb) and adds ecmp (1Mb) for IPv4 traffic, which causes a resource spike of 4Mb and a release of -2Mb.

4.2 Accelerating the CEGIS loop

We have now obtained a sequence CEGIS algorithm that is guaranteed to be sound (i.e., a synthesized transition is correct) and complete (i.e., if a correct transition exists, it will be found); it also produces the shortest transition due to the iterative deepening search. (More discussions in Section 4.4) However, in terms of performance, this algorithm has a series of scalability bottlenecks. A traditional CEGIS problem only has to reason about SMT formulas generated from a single program, but FlexPlan produces formulas many times larger as they are derived from program sequences. Thus, we develop

two domain-specific optimizations to accelerate the proposal and verification phases, respectively, based upon a divide-and-conquer approach. Our observation is that, for specific CEGIS steps, we can avoid reasoning about `ssk` directly but instead reason about its comprising `vsk` instances individually. Since SMT algorithms tend to grow exponentially with the formula size, dividing a large instance (i.e., `ssk`) into many smaller ones (i.e., `vsk`) and reasoning about the smaller formulas individually is more efficient than reasoning about the larger instance in a single shot.

Snapshot learning and generalization. We extract insights from a single snapshot before entering the main CEGIS loop (Line 5, 'Opt-SnapL'; Figure 4). This algorithm learns what a "bad" snapshot might look like, and then generalizes the knowledge for the entire sequence. We observe that, if a snapshot `vsk` violates the safety property ϕ, then no matter where this snapshot appears in the transition sequence `ssk`, it still constitute a violation. Thus, there is much to learn from an individual `vsk` before we have to stitch many such snapshots together. Stated in another way, resource constraints δ force us to perform end-to-end reasoning in general, but the safety aspect of the reasoning is still decomposable to individual snapshots.

This is achieved by operating a loop that extracts as many unsat cores as possible (within timeout threshold) from a single snapshot `vsk`, but only asserting safety properties ϕ and ignoring resource concerns δ. Each iteration produces one counterexample that witnesses a specific violation for *any* snapshot in the sequence. For instance, a counterexample might say that {v1(T), v2(T), v3(F)} violates IPv4 execution consistency, so this snapshot should never appear anywhere in the sequence. After producing many counterexamples, we aggregate and feed such knowledge into the main CEGIS loop, so that the proposal phase will not err in the same way on the larger sequence sketches. Further, for efficiency, FlexPlan distills counterexamples into "minimum unsatisfiable cores" (unsat cores) [20], which is a subset of assignments to `vsk.v` as the root cause. In our running example, the unsat core {v1(T), v3(F)} articulates the essence of the violation—the two IPv4 related blocks are not updated together. The main CEGIS loop ingests this condensed knowledge, avoiding the larger formulas from full-blown counterexamples.

Snapshot verification. The 'Opt-SnapV' optimization reduces the task of verifying a proposed `ssk` against ϕ into smaller tasks of verifying each individual `vsk` in it. The intuition still stems from the fact that ϕ can be reasoned per snapshot, whereas δ is a sequential property and needs to be synthesized end-to-end. The proposal phase (Line 9; Figure 4) must already ensure end-to-end feasibility in its proposal; so a subsequent verification may only fail due to violation of ϕ. Thus, when verifying `ssk`, we check individual `vsk` snapshots separately. If any snapshot produces a violation, its counterexample is used in the next round of synthesis.

Figure 5 shows the pseudocode for both optimizations.

```
1:  function SnapshotLearn(vsk, ϕ)
2:      uc_set ← ∅ // Aim to learn unsat cores from vsk
3:      while ¬ Timeout do
4:          // Solve for a new violation by negating ϕ
5:          {vsk.v, pkt} ← Symbolic(B^k, Packet)
6:          {vsk.v := v, pkt := p} ← SMTSolve(vsk, uc_set, ¬ϕ)
7:          if {v, p} == ∅ then // Exhausted all ce's
8:              return uc_set
9:          else // New violation, extract unsat core
10:             uc_set ← uc_set ∪ ExtractUC(v, p, ϕ)
11:     return uc_set
12: function SnapshotVerify(ssk, ϕ) // Verify a proposed ssk.
13:     ce_set ← ∅ // Counterexample set
14:     for vsk ∈ ssk do
15:         ce ← SMTVerify(vsk, ϕ)
16:         if ce ≠ ∅ then
17:             ce_set ← ce_set ∪ {ce}
18:             return false
19:     return true // All snapshots verify!
```

Figure 5: The snapshot learning (Opt-SnapL) and snapshot verification (Opt-SnapV) algorithms. Note that the snapshot learning algorithm does not need to enumerate all counterexamples or unsat cores, as it serves as an optimization for the main CEGIS loop. If the learning times out (Line 3), the collected uc_set is still useful in the main CEGIS.

4.3 Diagnosing the synthesis

Another domain-specific property of our synthesis lies in its *diagnosability*. In traditional synthesis, even if the tool struggles to find a solution, it may not mean that a valid solution does not exist—unless it has exhausted the search space. Thus, the search in the worst-case scenario may spend a significant amount of time only to conclude at the end with a failure. In contrast, we observe that FlexPlan can obtain three types of conclusive proof early in the game. This helps us to determine whether or not a continued search will be fruitful, and enables further optimizations. We call these techniques *introspection*.

Existence? A basic type of introspection is to determine whether or not a safe transition exists at all, regardless of the resource headroom. We observe that FlexPlan can determine this by checking p_{old} and p_{new} against the safety specification ϕ. If both programs are correct, then some safe transition must exist—the degenerate case is to make a one-step transition $\{p_{old} \rightarrow p_{new}\}$, exposing no intermediate state (but potentially causing a very large resource spike and thus may not be feasible). However, if even this check fails, FlexPlan aborts with the conclusion that no solution exists.

Deepening the search? Once we pass this smell test, we are faced with a harder introspection task—what should be the upperbound of t, the length of the sequence sketch? The algorithm in Figure 4 uses a naïve upperbound, where t iteratively deepens from one to k, the total number of change annotations. It first searches through all possible t-step transitions; if no such transition is both feasible and safe, it attempts

a longer transition sequence with $t+1$ steps. Failures are only conclusive at $t = k$, where each transition only applies one change thus no further breakdown is possible. However, this deepening search gets significantly more expensive with every increment to t—at each t, we need to perform a full round of CEGIS with a sequence of t sketches. Thus, it is beneficial to reflect on the usefulness of a larger t before we deepen the search, potentially stopping far earlier than the naïve bound. This introspection relies on the following property:

Introspection theorem. *Assume that the switch has infinite resources. Without resource constraints, if there does not exist a t-step safe transition plan, then there cannot exist any safe transition plan with more than t steps.*

Proof sketch: The intuition is that, if a t-step transition exists that satisfies ϕ but not δ, we can potentially make more granular changes in a longer transition sequence to stay within δ; thus, attempting a $(t+1)$-step transition could be fruitful. On the other hand, if a t-step transition that satisfies ϕ does not exist to begin with, then any t'-step transition where $t' > t$ cannot exist either. We can prove this by contradiction: if a t'-step transition exists, repeatedly combine any adjacent transitions into a larger transition until it becomes a t-step transition. This resulting transition must satisfy ϕ as it exposes strictly a subset of the states in the t'-step transition. □

Thus, when FlexPlan concludes that no t-step transitions exist, before it attempts a $(t+1)$-step CEGIS, it performs the above introspection. The introspection may conclude that a) some safe solution exists but b) no safe solution is feasible under the current resource constraints. This may be disappointing, but all is not lost—runtime programmable switches make it possible to deallocate resources to make extra room (e.g., by deleting certain tables or table entries).

Resource release? Whether to deallocate resources and which tables to delete are up to the network operator, but FlexPlan performs a third introspection to diagnose *how much* resource release would be sufficient for a t-step transition (where the search has stopped). This relies on an SMT optimization primitive `max-smt`, which can maximize an objective function while solving for a solution. Recall that each step causes a resource spike during the transition, and a headroom change after it. We track the the minimum headroom across a t-step transition, and ask for a solution that maximizes it:

$$\texttt{min_headroom} = \min_{\forall j} \texttt{vsk}_j\texttt{.headroom}$$

$$\delta^* = \texttt{max-smt(min_headroom) s.t. ssk} \wedge \phi$$

δ^* will be the smallest headroom possible to maneuver a t-step transition, and $\delta^* - \delta$ is the amount of resource release that is required to achieve a feasible update.

4.4 Remarks

We discuss several properties of the synthesis techniques.

Introspection. An important property of the introspection algorithms is that they work with sequence sketches of the current length of the search (i.e., t steps). Thus, they do not lead to new scalability bottlenecks. Furthermore, the combination of the second and third introspection techniques also enables a synthesis goal of identifying a safe transition plan with minimized resource spikes—first determine the sequence length upperbound for a safe transition (with the second introspection), and then synthesize a transition while minimizing resource spikes at this length (with the third introspection).

Guarantees. The completeness of the synthesis is derived from the fact that the candidate solution space is finite and that CEGIS will eventually finish an exhaustive search [26]. Concretely, the solution space is defined by the two-dimensional matrix $\texttt{H}_{k\times t}$. k is the number of annotations, and therefore finite. t is the sequence length, initially undetermined, but we know that it is upperbounded by k, because applying one annotation per step will result in the longest possible sequence. This is because we ask that each transition step makes positive progress, so no reverts are allowed once a change has been made. The synthesis is also sound, because FlexPlan always verifies the correctness of a proposed candidate plan.

5 Discussions and Limitations

P4 intermediate states. Intermediate states when the data plane is under change have been considered in the P4Runtime standard (cf. DATAPLANEATOMICS) [6]. However, the current standard focuses on the atomicity and intermediate states when adding or removing a batch of *table entries* for existing MA tables. Runtime table additions and removals, as a recent development, have not yet been captured in P4Runtime. Nevertheless, for table entry changes, P4Runtime describes how atomic pointer swaps can be used for transactional changes (when available in the target), and discusses the headroom requirement for preparing the changes in scratch area. This results in a similar range of considerations as recent designs for runtime programmable switches [19, 53]. We hope that FlexPlan will further the research in handling date plane intermediate states and the standardization process in P4Runtime.

Change annotation primitives. In the spirit of target-independence, our change annotations capture the intersection of hardware reconfiguration primitives between FlexCore [53] and rP4 [19]. Reconfiguration primitives that are not yet fully supported across platforms (e.g., parser changes [19] and table swap operations [53]) are considered out of scope for the current paper. These are interesting avenues for future work.

Switch architectures. Recent designs of runtime programmable switches [19, 53] employ disaggregation to split memory from compute. Thus, FlexPlan models memory resources as a global constraint—e.g., when a table is removed, the released resources can be used anywhere. However, future runtime programmable switches might adopt alternative architectures—e.g., RMT switches [12] with fixed stage boundaries would require a different model on memory reusability. Similarly, for SmartNIC targets with software and hardware pipelines [1], atomic transactions may become

Programs	LoC	Tables	Synthesis results		Programs (α)	Specification	Headroom	# Steps	Time(s)	Greedy
			time(s)	c.e.(u.c.)						
flowlet	216	6	5.61	0(0)	switch (20%)	IPv4 exec. consistency	80%	2 (✓)	110.32	132.37
simple_nat	362	6	6.02	1(1)	switch (20%)	IPv4 exec. consistency	50%	3 (✓)	160.25	timeout
ndp	275	7	5.73	1(1)	switch (20%)	IPv4 exec. consistency	20%	5 (×)	318.95	timeout
beamer	448	7	6.79	2(2)	switch (40%)	IPv4 exec. consistency	50%	3 (✓)	197.53	timeout
vpc	272	10	6.48	2(2)	switch (40%)	Espec field consistency	20%	4 (✓)	263.38	timeout
sai_p4	697	14	7.18	2(2)	switch (40%)	L2/L3 field consistency	20%	4 (✓)	436.44	timeout
linear_road	846	24	13.48	4(4)	switch+meter-stat	L2/L3 field consistency	20%	2 (✓)	186.24	552.82
nethcf	822	30	14.71	6(6)	switch+meter-stat	IPv4 exec. consistency	20%	2 (×)	93.90	105.51
netcache	1845	96	37.59	14(14)	switch+ipv4-ipv6	L2/L3 field consistency	20%	2 (✓)	249.99	749.78
switch	5599	120	199.43	27(24)	switch+ipv4-ipv6	IPv4 exec. consistency	20%	2 (✓)	102.35	124.46

Table 2: FlexPlan scales to real-world programs. The left-hand side uses a range of popular programs, ranked by the number of MA tables they contain. It uses an update ratio of 20% with 50% resource headroom. The right-hand side focuses on case studies with switch.p4, including synthetic and realistic changes. For each change, we denote resource peak needed to atomically update the entire program in a single step as S, and set the headroom to $\beta \times S$ ($\beta = 20\%, 50\%, 80\%$). The greedy synthesis times out in 5 out of 10 cases, and it is much slower than FlexPlan for the rest of the cases.

harder due to the need to synchronize across pipelines. Thus, as alternative architectures become available in the future, FlexPlan may need to incorporate a new set of constraints.

Safety properties. The FlexPlan consistency specifications capture safety (but not liveness) properties that are defined over multiple program executions, or k-safety hyperproperties [15, 49]—specifically, 3-safety, as FlexPlan reasons about the old, new, and current snapshots. Among safety properties, it does not analyze stateful packet processing, where program behaviors may mutate based on the input packets [30]. Further, FlexPlan only reasons about a single network device, so extending it for network-wide updates is future work.

Resource utilization. FlexPlan considers switch memory constraints as the main bottleneck resource. In a P4 program, each MA table has a 'size' field that specifies the maximum number of entries it could contain. This provides coarse-grained information as input to FlexPlan. However, the number of entries in a table is not the same as physical memory consumption, which further depends on the match types and their target-specific implementation (e.g., TCAM vs. SRAM). To obtain exact information, FlexPlan would need the compiler to produce such data for the old and new programs. Modeling other types of resource constraints is left to future work.

Synthesis delay. Requiring each runtime update to go through a formal synthesis phase will incur delay to the change. As we will show, the latency is a few minutes across our evaluation. We believe that reliability gains outweigh the resulting delay.

6 Evaluation

Prototype. We have built FlexPlan in ∼5000 lines of code (available at [2]). Our prototype consists of two components: (1) a frontend translator building upon an existing tool [18], which takes in the annotated P4 program and the specification, and converts them into an instrumented sequence sketch and SMT formulas; and (2) the main CEGIS backend which searches for a transition and performs introspection whenever needed. We use Z3 [8] as the SMT solver.

Methodology. Our program corpus is based upon popular programmable switch applications, representing use cases in monitoring, security, offloading—similar as recent P4 verification projects [18, 38, 50]. It contains real-world P4 programs, with sizes ranging from 200+ to 5000+ LoC. Further, FlexPlan uses two methods to generate program changes. Synthetic changes are generated using a similar strategy as existing work [53], which controls the number of changes with a parameter α. If a program has M tables, an update ratio α will generate $M \times \alpha$ table additions, deletions, or modifications. To test realistic changes, we use switch.p4 as the basis to perform manual modifications based on its control block boundaries (e.g., remove or add back the egress statistics control block `process_egress_bd_stats()`), mimicking feature changes in realistic deployments. To analyze headroom, we assume that each table has the same size, denoted as U.

Evaluation objectives. Our evaluation primarily focuses on various dimensions of scalability: (1) How well does FlexPlan scale to real-world programs and sizable changes? (2) How well do the granular consistency levels work? and (3) How effective are the FlexPlan optimization and introspection techniques? We note that existing P4 verification projects [18, 38] analyze the correctness of single program snapshots, so they are not suitable as baseline solutions for comparison. Thus, we have created several FlexPlan variants as baseline solutions, where specific optimization techniques are disabled.

6.1 Macrobenchmarks

We start with the macrobenchmarks summarized in Table 2.

Scalability. The first macrobenchmark (left four columns) tests the scalability of FlexPlan with popular switch programs. We use "L2/L3 field consistency" (i.e., intermediate snapshots should preserve the same L2/L3 processing outcome) with a fixed update ratio of 20%. Further, we set the headroom to 50% of what would be required for the most straightforward plan that updates the entire program in a single step, which in turn would lead to the highest possible resource peak. There are two high-level takeaways. First, as the program size increases from 200+ to 5000+ LoC, synthesis time also grows

significantly, as larger programs produce bigger SMT formulas. However, even for the largest program, FlexPlan was able to finish within 3.4 minutes. Second, the number of learned counterexamples ('ce') also grows with the program size, and FlexPlan effectively learned from a relatively few number of counterexamples (varying from 0 for the smallest programs to 27 for the largest) before finding a correct transition. We further break it down by showing the number of counterexamples learned from a single snapshot (i.e., in the process of enumerating unsat cores with SnapL, labeled as 'uc'). Except for switch.p4, FlexPlan enumerated all unsat cores in the SnapL phase, so the main CEGIS loop identified a correct solution in the first attempt. For switch.p4, FlexPlan performs 3 CEGIS iterations in the main loop to identify the transition.

Synthesized switch.p4 changes. The second macrobenchmark (right-hand side) zooms in on modifications to switch.p4, a datacenter switch implementation. We first test six synthesized changes with different control parameters—resource headrooms (rows 1-3), update ratios (rows 2+4), specification types (rows 5-6)—and examine their influence on the transition sequence lengths and synthesis time. As we can see, more severe headroom constraints lead to longer synthesis time (rows 1-3). FlexPlan had to try longer transition sequences to find a solution or conclude that no solution exists (shown as $\times$). Larger update ratios also lead to longer synthesis time (e.g., 23% increase when changing from $\alpha = 20\%$ to $\alpha = 40\%$ in rows 2 and 4). Moreover, the specification type also has a direct influence on synthesis. Field consistency on `egress_spec` is easier to check compared to L2/L3 header checks. FlexPlan took 1.8-7.3 minutes across all cases.

Hand-crafted switch.p4 changes. Next, we analyze a set of hand-crafted changes to switch.p4, by adding, removing, or modifying well-defined control blocks. We also vary the specifications across data points. The `switch+meter-stat` case removes all statistics tables (5 tables overall) in switch.p4, and adds in meter related tables (4 tables). For L2/L3 field consistency, FlexPlan identifies an update plan with 2 transitions. A closer look at the update plan shows that FlexPlan first removes all statistics tables, then adds meter tables—this is possible because statistics tables do not manipulate L2/L3 packet headers. On the other hand, IPv4 execution consistency fails to generate a update plan because most of the modified tables share some execution paths, which means they must be updated together. The required headroom goes beyond the available resources (20%). The `switch+ipv4-ipv6` update removes IPv6 processing tables (10 tables in total) and adds IPv4 processing tables (8 tables). IPv4 execution consistency, on the other hand, could find a update plan with 20% headroom. This is because IPv4 and IPv6 tables can be updated separately as they do not share execution paths. All experiments with realistic changes finished within 4.2 minutes.

FlexPlan vs. greedy synthesis. We also test the greedy synthesis algorithm (Section 4.1.1), which either maximizes progress ('Greedy MinSeq') or minimizes resource spikes ('Greedy MinSpike') for each step locally. As Table 2 shows, it times out after 30 minutes for five out of ten cases. When it is able to produce a transition, it only finds suboptimal solutions. Figure 6 visualizes a case study on NetCache, where greedily maximizing per-step progress results in a transition in seven steps with 80% headroom, whereas FlexPlan produces a much shorter transition in four steps ('FlexPlan MinSeq'). Similarly, FlexPlan when minimizing resource usage (cf. Section 4.4) leads to much lower peak usage than greedily minimizing resource spikes per step. This demonstrates the benefits of the sequence sketch encoding for end-to-end synthesis.

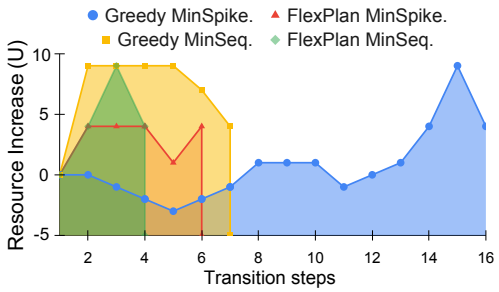
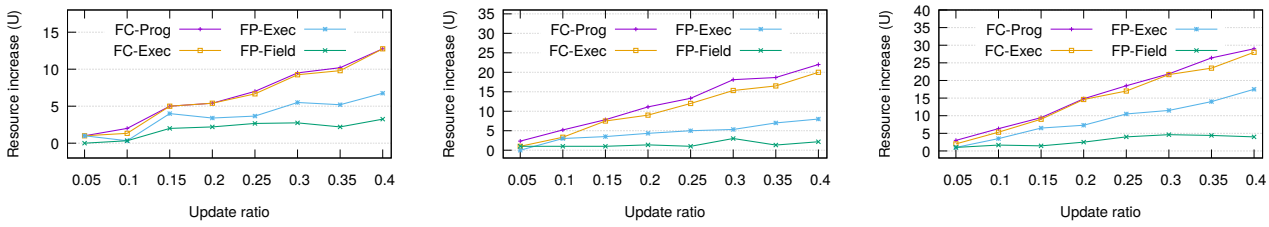


Figure 6: A NetCache case study showing step-by-step transitions and the headroom changes. The greedy synthesis produces suboptimal transitions whtn it is able to finish; it times out under more severe resource constraints.

6.2 Consistency levels vs. headroom

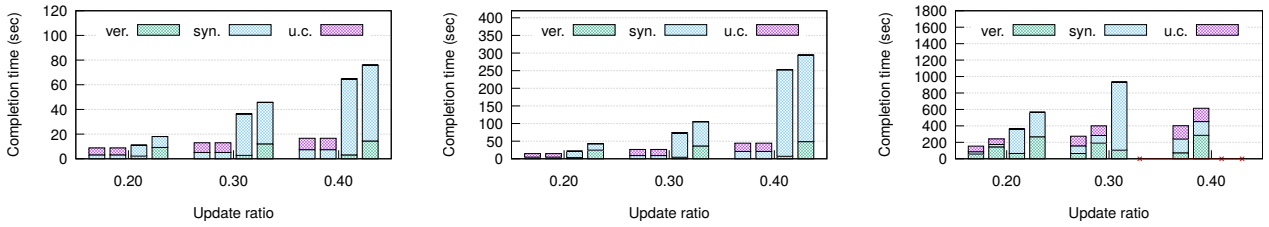
Next, we show that the granular consistency specifications in FlexPlan can lead to lower resource requirements when rolling out an update. We use "program consistency" and "execution consistency" as baselines, which are the strongest and weakest guarantees developed in existing work, respectively [53]. We chose three large switch programs (NetHCF, NetCache, switch.p4), and generated 50 changes with α ranging from 5% to 40%. For each change, we ask FlexPlan to find the transition sequence that minimizes the peak resource usage under each specification. We then averaged across all changes with the same α and show the results in Figure 7.

Our first takeaway is the inflexibility of heuristic-based definitions in FlexCore [53]. Although "execution consistency" is a weaker requirement than "program consistency," it unfortunately does not reduce the resource peak by much and the two corresponding curves are closely coupled together (i.e., 'FC-Prog' vs. 'FC-Exec'). It can reduce the resource peaks for specific cases, but aggregating over all cases the reduction is only 5%. We found that this is highly correlated to the program shapes and where the changes are made. A common root cause can be attributed to the *bottleneck table* problem. If a change modifies some tables that are shared by many or all execution paths, then it forces all changes to be made in the same step, equating "execution consistency" to "program consistency." Since FlexCore [53] only relies on "reachability" information at table level and does not ana-



(a) NetHCF.p4 (b) NetCache.p4 (c) Switch.p4

Figure 7: The granular consistency specifications in FlexPlan can effectively reduce peak resource usage. FC-Prog and FC-Exec represent program consistency and execution consistency properties as defined in FlexCore [53], which only analyzes reachability across tables without considering program semantics. FP-Exec refines execution consistency to consider specific traffic classes in FlexPlan. FP-Field is a new consistency definition in FlexPlan that constrains the processing outcomes of specific header fields.



(a) NetHCF.p4 (b) NetCache.p4 (c) Switch.p4

Figure 8: The snapshot-based optimizations are effective. Each group of bars plots the turnaround times, from left to right, for FlexPlan, NoSnapV (no snapshot verification), NoSnapL (no shapshot learning), and NoSnapVL (disabling both optimizations), respectively. On switch.p4, both techniques are necessary for sizable changes; otherwise the analysis will time out after 30 minutes. We further break down each bar by the time the solution spends in verification, synthesis, and unsat core extraction.

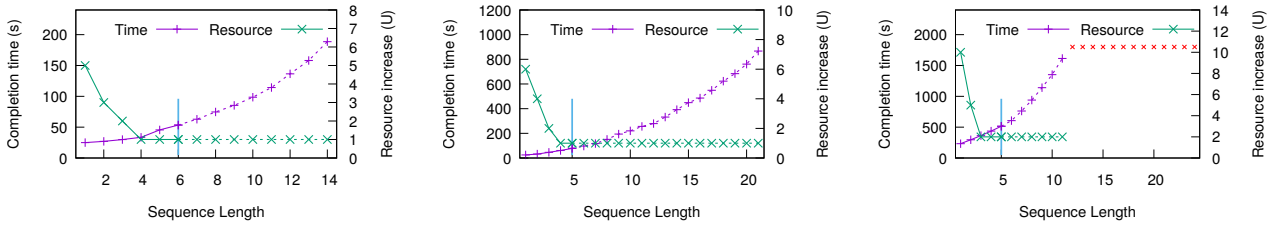
lyze program semantics, it cannot distinguish whether or not a change actually affects a particular traffic class, conflating the two guarantees whenever bottleneck tables are modified.

In contrast, the granular consistency levels in FlexPlan are effective in reducing the peak resource usage by capturing program semantics. First, FlexPlan refines "execution consistency" even further by defining it over traffic classes of interest. Specifically, we have tested three variants of "execution consistency" defined over traffic classes (shown as 'FP-Exec'). For switch.p4, we specify it only for IPv4 traffic without tunneling; for NetCache, only for read requests to the cache data structure; and for NetHCF, only packets that establish TCP sessions. Thus, if a table change does not affect how these traffic classes are processed—even at a bottleneck table—FlexPlan is still able to produce granular transitions with lower peak resource usage. The reductions for NetHCF, NetCache, and switch.p4 are 47%, 64%, and 45%, respectively, compared to the fixed execution consistency in FlexCore. Further, we have also tested "field consistency," to showcase FlexPlan's ability to define new consistency levels beyond tuning traffic classes (shown as 'FP-Field'). This states that L2/L3 headers and standard metadata must be processed with the same outcome during transition. This leads to even lower peak usage: the reduction is 46%, 67%, and 68%, respectively, compared to the granular execution consistency levels above. This is because it allows a mix of old and new tables to co-exist on execution paths, as long as this does not change the processing behaviors for specific header fields.

6.3 Snapshot learning and verification

Next, we evaluate the effectiveness of the snapshot learning (SnapL) and verification (SnapV) optimizations. We create three baseline solutions from FlexPlan where one or both optimizations are turned off: NoSnapV, NoSnapL, and NoSnapVL. Figure 8 compares the four solutions with different update ratios and programs. Across all data points, FlexPlan outperforms NoSnapVL in terms of completion time by 70% on average, demonstrating the effectiveness of the two snapshot-based optimizations. On switch.p4, the NoSnapVL baseline times out when $\alpha > 20\%$. Further decomposition shows that the SnapL and SnapV optimizations lead to 58% and 37% improvements on average, respectively.

Both SnapL and SnapV are more effective with larger programs (which lead to larger SMT formulas per snapshot) and higher update ratios (which lead to a larger update plan search space). For instance, at $\alpha = 40\%$, SnapL reduces the completion time by up to 81% for NetCache. With smaller update ratios, the number of possible update sequences is already small, so the time spent in learning unsat cores with SnapL does not afford as much improvement. The trend for SnapV is similar. For smaller formulas (e.g., NetHCF and NetCache), it is possible for FlexPlan to iterate through all unsat cores,



(a) NetHCF.p4 (b) NetCache.p4 (c) Switch.P4

Figure 9: The FlexPlan introspection technique is effective in determining whether the synthesis should continue to try longer sequences. The vertical lines denote the stopping points for FlexPlan, which significantly outperforms a solution with introspection turned off, which naïvely increases the sequence length beyond the vertical lines until the maximal upper bound or timeout.

so the main CEGIS phase could bypass the verification completely since the first proposal attempt will identify a correct plan. Its impact is stronger on larger programs like switch.p4.

6.4 Introspection and diagnosis

We now evaluate the three CEGIS introspection techniques.

Existence. We first generate a set of program modifications that are guaranteed to violate the safety specification. For instance, if the specification requires that egress_spec processing outcomes should be preserved, we ensure that it is modified in a different way. For all cases, FlexPlan correctly rejects the annotated change as unsafe within 90 seconds.

Sequence length. Figure 9 evaluates the FlexPlan introspection for determining whether to attempt a longer sequence. We create cases where the resource constraints will guarantee an eventual failure, and test how soon FlexPlan can detect this inevitability. We also compare against a version of FlexPlan without this introspection. For NetHCF (α=40%), NetCache (α=20%), and switch.p4 (α=20%), FlexPlan concludes that the synthesis will fail when the sequence lengths are six, five, and five, respectively, within several minutes. However, the solution without introspection will keep increasing the sequence lengths (and running time) until the maximal upper bound (i.e., the total number of changes) or timeout, without being able to decrease resource usage further. It took 3$\times$ and 12$\times$ more time to conclude that the resources are insufficient for NetHCF and NetCache, respectively; for switch.p4, it times out before producing any useful results. Thus, the introspection technique helps determine failures efficiently.

Resource release. After each failure, we further ask FlexPlan to produce diagnostic results on the least amount of resource release that will enable a safe and feasible transition. This is achieved by introspecting the sequence upper bound for a safe transition, and then solving for a transition with minimized resource peak. This diagnosis took less than ten minutes across all update ratios and programs—the longer completion time is due to the need for trying the longest possible safe transition. We then emulated the release by increasing the switch headroom by the suggested amount, and re-attempted another synthesis and verified that it succeeded.

7 Related Work

Runtime programmability. Network switches have become programmable at runtime [3, 4, 9, 19, 52, 53], where switch programs can be modified with partial reconfiguration without downtime. Runtime programmability has also been studied in host networking [40,42]. FlexPlan develops a formal approach to synthesizing runtime programmable switch updates.

Safe network updates. Ensuring the safety of network updates [34, 43] is a key goal in cloud datacenters [37]. In the context of OpenFlow-based SDN, consistent update algorithms have been extensively studied [21, 44, 45]. FlexPlan considers an analogous problem for programmable data planes, with new definitions on correctness and intermediate states, and uses program synthesis to achieve this goal.

Synthesis and verification. Program synthesis has found many applications in the networking domain [13, 22, 59], including for identifying safe configuration updates in OpenFlow SDN [39]. For programmable switches, existing projects have developed many formal verification techniques for P4 programs [18,38,50,51,61]. Compared to these lines of work, FlexPlan aims at synthesizing a correct-by-construction update sequence, which in turn requires new techniques.

8 Conclusion

Programmable networks enable the *development* of new features "in the field," without relying on slow-paced vendors. To safeguard network behaviors, formal verification has proven essential. Runtime programmable networks [57], in contrast, emphasize that the *deployment* of these features must also be seamless and "in the field"—without requiring slow-paced maintenance operations. However, live program modifications necessitate new techniques for providing formal assurance. FlexPlan is a synthesis tool that can identify a safe and feasible program transition sequence automatically. It introduces domain-specific techniques for synthesizing switch program updates. With comprehensive evaluation, we demonstrate the scalability of FlexPlan on real-world programs.

Acknowledgments: We thank our shepherd Muhammad Shahbaz and all reviewers, as well as Jiarong Xing and Kuo-Feng Hsu for their insightful comments and suggestions. This work was supported by NSF in part by CNS-2214272.

References

[1] BlueField SmartNIC Ethernet. https://www.mellanox.com/products/BlueField-SmartNIC-Ethernet.

[2] FlexPlan code repository. https://github.com/824728350/FlexPlan.

[3] Jericho2. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88850.

[4] The NPL network programming language. https://github.com/nplang.

[5] Nvidia/Mellanox Spectrum Ethernet Switches. https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/.

[6] P4 Runtime Specification: Atomicity, Batch and Ordering of Updates: DataPlaneAtomic. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html#sec-batching-and-ordering-of-updates.

[7] P4 Specification: Annotations. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html#sec-annotations.

[8] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[9] Trident4 boosts enterprise switch capacity to 12.8 terabit. http://www.gazettabyte.com/home/2019/7/11/trident-4-boosts-enterprise-switch-capacity-to-128-terabit.html.

[10] Wedge 100bf-32x 100gbe data center switch. https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[13] E. H. Campbell, W. T. Hallahan, P. Srikumar, C. Cascone, J. Liu, V. Ramamurthy, H. Hojjat, R. Piskac, R. Soulé, and N. Foster. Avenir: Managing data plane diversity with control plane synthesis. In *Proc. NSDI*, 2021.

[14] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, et al. dRMT: Disaggregated programmable switching. In *Proc. SIGCOMM*, 2017.

[15] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Cornell University Tech. Report*, 2009. https://hdl.handle.net/1813/11660.

[16] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. NSDI*, 2018.

[17] C. David and D. Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions A: Mathematical, Physical and Engineering Sciences*, 2017.

[18] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu. bf4: Towards bug-free P4 programs. In *Proc. SIGCOMM*, 2020.

[19] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *Proc. NSDI*, 2022.

[20] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *Proc. PLDI*, 2018.

[21] K.-T. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent software-defined network updates. *IEEE Communications Surveys Tutorials*, 21(2):1435–1461, 2019.

[22] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta. Switch code generation using program synthesis. In *Proc. SIGCOMM*, 2020.

[23] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from Google's network infrastructure. In *Proc. SIGCOMM*, 2016.

[24] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[25] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[26] S. Jha and S. A. Seshia. Are there good mistakes? A theoretical analysis of CEGIS. In *Proc. SYNT*, 2014.

[27] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soule, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. NSDI*, 2018.

[28] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[29] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *Proc. SIGCOMM*, 2014.

[30] Q. Kang, J. Xing, Y. Qiu, and A. Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *Proc. ASPLOS*, 2021.

[31] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[32] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. SOSR*, 2016.

[33] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[34] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *Proc. HotNets*, 2013.

[35] G. Li, M. Zhang, C. Liu, X. Kong, A. Chen, G. Gu, and H. Duan. NetHCF: Enabling line-rate and adaptive spoofed IP traffic filtering. In *Proc. ICNP*, 2019.

[36] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. In *Proc. SIGCOMM*, 2013.

[37] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: updating data center networks with zero loss. In *Proc. SIGCOMM*, 2013.

[38] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, C. C. Robert Soulé, Han Wang, N. McKeown, and N. Foster. p4v: Practical verification for programmable data planes. In *Proc. SIGCOMM*, 2018.

[39] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *Proc. PLDI*, 2015.

[40] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi. Domain specific run time optimization for software data planes. In *Proc. ASPLOS*, 2022.

[41] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[42] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári. Dataplane specialization for high-performance OpenFlow software switching. In *Proc. SIGCOMM*, 2016.

[43] T. D. Nguyen, M. Chiesa, and M. Canini. Decentralized consistent updates in SDN. In *Proc. SOSR*, 2017.

[44] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[45] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. HotNets*, 2011.

[46] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[47] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *Proc. ASPLOS*, 2006.

[48] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. ATC*, 2018.

[49] M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proc. PLDI*, 2016.

[50] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging P4 programs with Vera. In *Proc. SIGCOMM*, 2018.

[51] B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang, J. Lu, X. Wei, H. H. Liu, M. Zhang, C. Tian, and M. Yu. Aquila: A practical usable verification system for production-scale programmable data planes. In *Proc. SIGCOMM*, 2021.

[52] T. Wang, X. Yang, G. Antichi, A. Sivaraman, and A. Panda. Isolation mechanisms for high-speed packet-processing pipelines. In *Proc. NSDI*, 2022.

[53] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen. Runtime programmable switches. In *Proc. NSDI*, 2022.

[54] J. Xing, K.-F. Hsu, Y. Qiu, Z. Yang, H. Liu, and A. Chen. Bedrock: Programmable network support for secure RDMA systems. In *Proc. USENIX Security*, 2022.

[55] J. Xing, Q. Kang, and A. Chen. Netwarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[56] J. Xing, A. Morrison, and A. Chen. NetWarden: Mitigating network covert channels without performance loss. In *Proc. HotCloud*, 2019.

[57] J. Xing, Y. Qiu, K.-F. Hsu, H. Liu, M. Kadosh, A. Lo, A. Akella, T. Anderson, A. Krishnamurthy, T. S. E. Ng, and A. Chen. A vision for runtime programmable networks. In *Proc. HotNets*, 2021.

[58] J. Xing, W. Wu, and A. Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *Proc. USENIX Security*, 2021.

[59] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proc. SIGCOMM*, 2021.

[60] L. Yu, J. Sonchack, and V. Liu. Mantis: Reactive programmable switches. In *Proc. SIGCOMM*, 2020.

[61] N. Zheng, M. Liu, E. Zhai, H. H. Liu, Y. Li, K. Yang, X. Liu, and X. Jin. Meissa: scalable network testing for programmable data planes. In *Proc. SIGCOMM*, 2022.

9 Appendix

This appendix includes more details on the safety specifications as summarized in Table 1 in the main paper.

9.1 Specifications

S4: Element consistency for ACL. This specification is similar as program consistency, but only constrains traffic that goes through a particular ACL table. For all packets that have been processed by the ACL table, their execution paths must be of the same version (i.e., either old or new).

```
specification {
  ghost bit<1> sawOld = false;
  ghost bit<1> sawNew = false;
  ghost bit<1> acl_hit = false;
  @old => { sawOld = true; }
  @new => { sawNew = true; }
  @hit('acl') => { acl_hit = true; }
  all_old = {
    $cur.in.acl_hit => !$cur.eg.sawNew;
  }
  all_new = {
    $cur.in.acl_hit => !$cur.eg.sawOld;
  }
  assert all_old || all_new;
}
```

S5: Table consistency for ECMP. We introduce a new consistency definition that is not available from existing

work [53], which we call "table consistency." It states that for each packet, the table (in this case ECMP) hit/miss behavior in the intermediate state should be either the same with the old or the new processing logic.

```
specification {
  ghost bit<1> ecmp_hit = false;
  @hit('ecmp') => { ecmp_hit = true; }
  // preserve processing behavior across states
  table_consistency_ecmp = {
    $cur.in == $old.in == $new.in =>
      ($cur.eg.ecmp_hit == $new.eg.ecmp_hit ||
       $cur.eg.ecmp_hit == $old.eg.ecmp_hit);
  }
  assert table_consistency_ecmp;
}
```

Although FlexPlan's primary focus is consistency guarantees during the program transition, its specification language can naturally support general program correctness properties that are used in P4 program verification [18, 38]. We showcase two of them below. At a high level, general program correctness properties are expressed by constraining a single snapshot using `$cur`, without referring to `$old` or `$new`.

S6: VLAN table access. Packets should go through VLAN table during any intermediate state.

```
specification {
  ghost bit<1> vlan_hit = false;
  @hit('vlan') => { vlan_hit = true; }
  table_access_vlan = {
    $cur.eg.vlan_hit == true;
  }
  assert table_access_vlan;
}
```

S7: Modification on ipv4.ttl. Any intermediate program snapshot should always decrement the ipv4.ttl header by one when the packet leaves the egress.

```
specification {
  decrement_ipv4_ttl = {
    $cur.eg.ipv4.ttl == $cur.in.ipv4.ttl - 1;
  }
  assert decrement_ipv4_ttl;
}
```

9.2 Instrumentations

Figure 10 includes an illustrative example that shows how a specification is translated into instrumentations in the input P4 program. This uses our running example `acl_flowlet_ecmp`, building upon the version sketch in Figure 3 and adding the instrumentations from the specification S1.

As we can see, the `sawOld`, `sawNew` and `acl_hit` variables are directly inserted into source program. We then add instrumentation that checks whether there exists a packet that violates the execution consistency after going through execution path. We then compute the weakest preconditions for

```
control ingress {
    apply {
        ghost_ipv4_valid = ipv4.isValid();
        /* annotation site 1: acl->nat_acl */
        if(ipv4.isValid()) {
            if (! vsk.v1) { // @mod
                acl.apply(); ghost_saw_old = 1;
                ghost_acl_hit = 1;
            } else {
                nat_acl.apply(); ghost_saw_new = 1;
                ghost_acl_hit = 1;
            }
        }
        /* annotation site 2: delete flowlet */
        if(! vsk.v2) {      // @del
            if (ipv6.isValid())
                flowlet.apply(); ghost_saw_old = 1;
        }
        /* annotation site 3: add ecmp */
        if(vsk.v3) {    // @add
            ecmp.apply(); ghost_saw_new = 1;
        }
        if (ghost_ipv4_valid && ghost_acl_hit) {
            if (ghost_saw_old && ghost_saw_new){
                violation();
            }
        }
    }
}
```

Figure 10: Instrumenting the version sketch in Figure 3 with safety specification, and also adding statements that check the safety properties.

the reachability of the violation nodes. This is achieved by iterating through CFG nodes and propagating stronger conditions to all their neighbors based on the transition relation. We then check whether the predicate is valid using the Z3 theorem prover. In the example, the combined program and safety formula to check would be derived as:

$$
\begin{aligned}
gso &= (\text{ipv4.isValid} \wedge \neg \text{vsk.v1}) \vee (\neg \text{vsk.v2} \wedge \text{ipv6.isValid}) \\
gsn &= (\text{ipv4.isValid} \wedge \text{vsk.v1}) \vee \text{vsk.v3} \\
gah &= \text{ipv4.isValid} \\
check &= \neg(\text{ipv4.isValid} \wedge gah \wedge gso \wedge gsn)
\end{aligned}
$$

where *gso* represents the logical formula for when `ghost_saw_old` is assigned true, and similarly for *gsn* as `ghost_saw_new` and *gah* for `ghost_acl_hit`. The logical variable *gso*, for example, captures the path conditions required to set the ghost variable to true as well as any intermediate assignments on the path to variables that might affect these path conditions/branches.

Practical Intent-driven Routing Configuration Synthesis

Sivaramakrishnan Ramanathan
Meta

Ying Zhang
Meta

Mohab Gawish
Meta

Yogesh Mundada
Meta

Zhaodong Wang
Meta

Sangki Yun
Meta

Eric Lippert
Meta

Walid Taha
Meta

Minlan Yu
Harvard University

Jelena Mirkovic
USC/ISI

Abstract

Configuration of production datacenters is challenging due to their scale (many switches), complexity (specific policy requirements), and dynamism (need for many configuration changes). This paper introduces Aura, a production-level synthesis system for datacenter routing policies. It consists of a high-level language, called RPL, that expresses the desired behavior and a compiler that automatically generates switch configurations. Unlike existing approaches, which generate full network configuration for a static policy, Aura is built to support frequent policy and network changes. It generates and deploys multiple parallel policy collections, in a way that supports smooth transitions between them without disrupting live production traffic. Aura has been deployed for over two years in Meta datacenters and has greatly improved our management efficiency. We also share our operational requirements and experiences, which can potentially inspire future research.

1 Introduction

Stable and efficient routing in large data centers is crucial for many online service providers. Routing misconfiguration can lead to packet drops, traffic black holes, performance degradation, and service downtime [2, 3, 9, 22]. Traditionally at Meta datacenters, routing configuration relied on human operator expertise to manually translate high-level routing policies into low-level switch configurations. This approach has two key problems.

First, manual generation of policies is often error-prone, especially with the enormous increase in scale (thousands of switches across multiple data centers), complexity (policies that describe specifications unique to a network), and dynamism (configuration changes to accommodate failures or maintenance of switches) of modern datacenters [10, 14].

Second, manually crafting configurations for datacenters is a time-consuming process. Earlier data centers at Meta were uniform and the similar configurations could be used to provision new data centers. However, there is a need to support diverse topologies required for various AI applications or existing topologies that are modified to accommodate resource shortages caused by supply chain bottlenecks. With the need to support diverse topologies, existing configurations can no longer be reused and each new topology implies a long process of manual configuration.

Recently, researchers have proposed configuration synthesis [7, 8, 12], which automatically generates switch configurations based on high-level policies. These systems usually provide a declarative language for operators to define the intended routing policies and then automatically synthesize low-level routing configurations, which implement these policies. While these solutions work well in principle, there are still a few challenges that are not handled by one-shot automated configuration synthesis.

Challenge 1: Handling dynamic configurations. Configuration changes are frequent in networks, because of many dynamic events. The dynamism is driven by different business objectives (shifting services from one data center to another), making network operations more efficient (e.g., smarter load balancing or more redundancy to failure), safely testing new protocols, or even performing regular routine maintenance of switches. Configuration synthesis should be able to generate configurations that natively handle dynamism.

Challenge 2: Expressing conditional policies. Current declarative languages express routing policies in a way that is not aligned with the realities of a production network. First, they treat each switch as live and ready to serve traffic. Yet, in large-scale data centers, switches can be in different operational states at the same time, and thus we need to be able to express routing policies that depend on these states. Second, current declarative languages specify all switches at a fixed granularity (e.g., one specific switch [7] or all switches in the specific role [8]). However, operational needs require specifications at a flexible granularity. Some intents in high-level policies may require specific switch or set of switches in one location, while others may require all switches in a given role.

Challenge 3: Reconfigurations at scale. Existing brownfield migration systems, plan out the configuration change

without disrupting production traffic by creating intermediate configurations that would help transitioning the network from old configuration to a new one [18, 23]. Although such techniques provide a safe, non-disruptive mechanism to change configurations, they can be expensive to carry out migrations. Our experiences show that deploying a new configuration in switches often takes a much longer time (minutes to hours) than computing the configuration (e.g., seconds), primarily because of the scale of the network. Given this constraint, migrating configurations via transitioning would take a very long time, hindering network operations.

In this paper, we introduce our configuration synthesis system Aura[1], which supports flexible granularity of policy intents, conditional intents and scalable synthesis to BGP configurations, to smoothly aid network dynamics. Our contributions are as follows:

- We propose a novel configuration synthesis approach, which pre-compiles a set of possible paths for the datacenter, called "base paths", given a set of high-level policies. Our insight is that given datacenter regularity and symmetry, any path can be expressed as one of a small set of base paths. Network operators can use these base paths to support dynamic configuration.
- We define a new declarative language called RPL (routing policy language), which allows operators to define policy intents with flexible switch granularities and activation conditions, based on switch states.
- Aura leverages configuration staging and uses labels to activate them. When there is a need to change configurations, routes are announced with appropriate labels (e.g., BGP community tags). Switches on receiving the labels, check the appropriate configuration that match the conditions and activate the configuration. This minimizes the need to reconfigure the network every time there is a need to change the configuration.

In addition to our contributions, Aura has been partially deployed in Meta datacenters for over two years, compiling all policies on hundreds of thousands of switches daily. To the best of our knowledge, we are the first to share operational experiences in building and deploying a configuration synthesis system. Our work unveils a unique set of challenges to academia, which we hope may inspire future research.

2 Background and Challenges

As discussed earlier, large-scale datacenters need automated configuration synthesis. Even with fully automated configuration synthesis, carrying out a data center-wide policy change still requires careful planning at every step, to ensure that the network remains operational throughout the change. We need to gradually roll out a policy change, with minimum

[1] We chose the name Aura, because it represents the essence (of an individual).

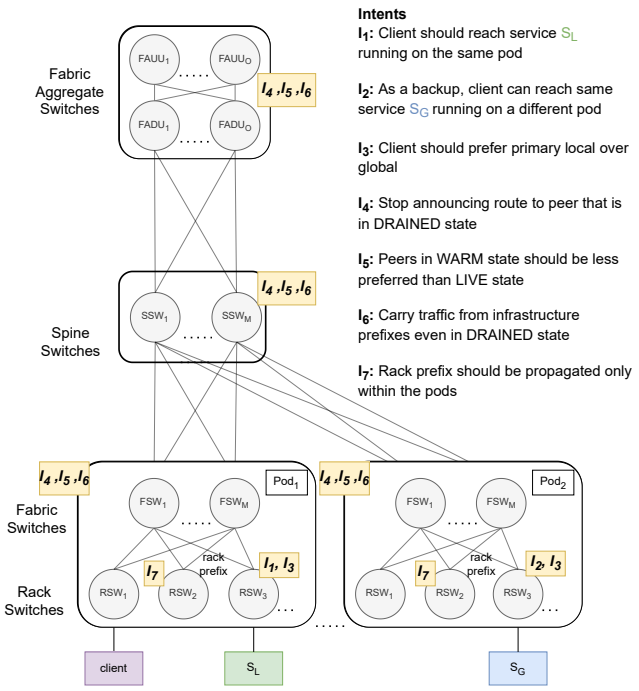


Figure 1: Data center topology with sample intents.

disruptions and safe fallback mechanisms. And we need to support such rollout across hundreds of thousands of switches, in different states of readiness. In this work, we stress one overlooked goal of configuration synthesis: producing configuration changes that lead to the shortest time to complete the reconfiguration, and making the process automated, non-disruptive to production traffic and with minimal operator burden. In this section, we start by describing our data center topology and routing intents. We then discuss the operational challenges of configuration synthesis in large data centers.

2.1 Background

We use Figure 1 to illustrate our production network topology [6, 29], a few sample intents (I_1 to I_7) used in our network, and the remaining open challenges, which our work addresses.

Data Center Topology: It consists of a hierarchy of four layers with thousands of switches at each layer. Switches at the same layer share the same *switch role*. The servers are connected to leaf rack switches (RSW). RSWs are connected to fabric switches (FSW). RSWs and FSWs are grouped into pods. Spine switches (SSW) connect the pods and provide several disjoint paths between pods. Data centers are distributed over multiple buildings and are interconnected via the Fabric Aggregation (FA) layer. The FA consists of two layers: FAUU (uplink) and FADU (downlink). FAUU connects to data-center-external networks (i.e., the backbone planes), while FADU aggregates downstream-data-center networks by connecting to SSWs. In this work, we limit our scope of configuration synthesis till the FAUU layer, and leave the con-

figuration synthesis of external switches for future work. This topology enables several disjoint end-to-end paths between any two server racks for failure resilience. Other topologies in Meta datacenters share the same properties with different numbers of layers and switch roles. Other large production data centers exhibit similar symmetry of roles and hierarchy of switch roles [24], and likely face similar challenges to configuration synthesis.

Routing: We use the Border Gateway Protocol (BGP) to disseminate routing information through the network and provide connectivity to end servers [5]. BGP is a highly scalable routing protocol that can support large topologies with many prefixes. BGP is also commonly supported many network switch vendors, and network operators are typically familiar with BGP operation. We configure BGP policies to manage how routing information is shared across the network and to control traffic flow objectives, such as traffic load-balancing, redundancy, and path preference. At Meta, BGP configurations within network tiers are homogeneous. We configure BGP to use Equal Cost Multipath (ECMP), where each switch forwards traffic equally to its neighboring peers based on the routing policy. Meta data centers are also experimenting with our own routing protocols such as OpenR [15] that is being rolled out into parts of the data center. OpenR policies can coexist with BGP policies. Currently, we use OpenR policies for infrastructure prefixes (those prefixes belonging to the management plane) and BGP for traffic prefixes.

Intents, policies, and configurations: We define an *intent* as a high-level description of a routing goal, e.g., all rack prefixes should be propagated within pods. Intents are closely tied to our operational needs of service reachability, maintainability and reliability. [5]. Figure 1 shows seven intent examples to achieve our needs in traffic control. I_1, I_2 and I_3 define reachability goals for a client to a service, that is, a service should be reachable even when an instance of the service is added, removed or migrated. Intents I_4 and I_5 help in managing networks during such events without disrupting production traffic, as switches often undergo maintenance when they fail, reboot, or even crash. I_6 is an exception to I_4 that aims to provide reliability under failure, and I_7 confines the propagation scope of a route. We will elaborate more about these intents from §2.2 to §2.4. We define a *policy* to be a collection of intents, e.g., a combination of intents I_1 to I_7. At Meta, there are many data centers, and we define a collection of policies known as *configurations* for every data center. For example, the datacenter in Utah (EAG) has 53 policies with each policy containing upto 5 intents. The synthesis process starts with a policy specified by a domain-specific language and produces a set of switch configurations.

2.2 Handling Dynamic Configurations

Configuration changes are frequent in production and often impose a high operational burden to ensure live production traffic is not affected. Previous synthesis systems focus on generating one snapshot of the entire network's configurations [7, 8]. When changes happen, they have to rerun the entire synthesis and generate another snapshot. There are three common scenarios that necessitate configuration changes:

Intent changes: Intents describe high-level objectives of reachability, aggregation, and route propagation (see [5] for routing objectives). They can change due to various operational needs – a service migrating from one data center to another, a better load balancing strategy (e.g., adding new paths to a destination in the ECMP pool), a more resilient failure recovery (e.g., adding a new backup path) or changing preference strategy (e.g., when we would move from location-based path selection to client-based preference, where the client can themselves set a preference of which server they would like to reach). The underlying routing configuration should reflect these intent changes. To support changes between existing intent collections, we need a synthesis approach that can generate multiple configurations in a switch and control when each would activate. The same approach can add new policies by enriching existing switches with new, inactive configurations. New configurations can then be activated gradually throughout the production network.

Policy implementation changes: We can implement intents expressed by operators in different ways, by using different protocols, or different mechanisms in one protocol. For example, in BGP, one can carefully set MED values, IGP values, or local preference values to achieve the same intents. In our data centers, we are exploring alternative routing protocols for better scalability, e.g. our home-grown intra-domain routing OpenR [15]. Thus, the same network-wide intents can be supported in OpenR with a completely different set of configurations compared to BGP. To facilitate testing, we need to gracefully roll out OpenR configurations and replace old BGP configurations. Such evolution requires a large amount of changes to configurations on all switches. To minimize impact to business, we need a way to gracefully migrate between configurations, and even roll back changes should they prove inefficient.

Switch state changes: Switches constantly undergo changes due to failures, new builds, and regular maintenance. In all of these scenarios, we need to gracefully remove the impacted switches from serving traffic, to minimize disruptions to services. Switch state changes are common in our production network. In the month of August 2022, there were about 745K drain events with about 8K drain events on average per day.

2.3 Expressing Conditional Policies

Conditional policies require different configurations for a given switch, depending on network conditions. There are three common classes of network conditions:

Intents that apply to a subset of switches: Operators need

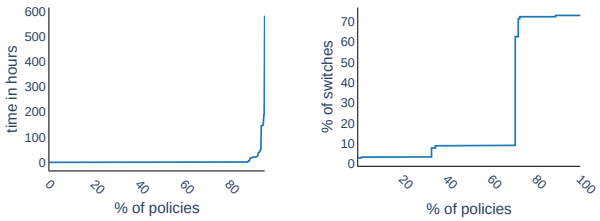
the flexibility to specify the groups of switches *where* a policy intent is applied. Some intents should be applied globally, while others may be relevant only for a specific service in a particular region to have customized routing solution. It is often needed during migration or deployment to accomodate the changing capacity. For example, intents I_1 to I_3 specify preference for local service S_L over global service S_G only for RSWs in Pod_1 and Pod_2, whereas intent I_7 applies to all RSWs and restricts rack prefixes to pods. This means our intent specification and synthesis should support conditions that define groups of switches at different topological granularity.

Intents that apply to switches in a certain state: Data center topologies support multiple paths between any pair of server racks. Path selection depends on switch states. We define three operational states for each switch: LIVE, DRAINED, and WARM. A LIVE switch is in operation; it allows all traffic and announcements to go through. Conversely, a DRAINED switch is brought down for maintenance; it should not carry any live production traffic. A switch being drained goes through an intermediate WARM state when prefixes are gradually removed from its announcements. In this state, the switch could carry traffic for any prefix if the prefix does not have other paths involving only LIVE switches. Switch configurations should suppress announcements from DRAINED switches (reflected in intent I_4), and favor announcements from LIVE switches over those from WARM switches (reflected in intent I_5). To keep the production network operational with these requirements, one method is to synthesize, which generates multiple configurations, for different switch states. Only one configuration is active at the time at a given switch, depending on the current state of the switch. For instance, intent I_4 specifies to stop announcing a route to peers in DRAINED states and intent I_5 specifies that we prefer peers in LIVE states than those in WARM states.

Exceptions to policies: Network operators need to specify exceptions to their policies for failure resilience. For instance, as per intent I_4, DRAINED switches do not carry any live traffic. However, an exception to this intent is I_6, which requires DRAINED switches to carry traffic towards infrastructure prefixes. Intent specification language needs to support exceptions for critical prefixes (e.g., infrastructure prefixes), and our synthesis process must generate corresponding configurations that treat those specific prefixes differently from others.

2.4 Reconfigurations At Scale

It is challenging to support dynamic policies and conditional policies at production scale. Switch reconfiguration in a live production network is expensive, because it typically requires transferring away all the services that use the network and draining all routes from the switch. After these actions, we can bring down the switch, change the configurations, and then bring up the switch again. Finally, the switch would be tested before reintroducing the routes that it carried before



(a) Configuration update times. (b) Reconfigured switches.

Figure 2: Switch reconfiguration metrics.

reconfiguration. Typically, not all switches in the data center are reconfigured at the same time. Instead, reconfiguration is achieved in a phased manner, where only a portion of the network is reconfigured at a time. This process could take many weeks to update all switches, given the scale of a production network, as well as the complexity of the phased deployment [10]. Figure 2(a), shows the configuration update times for policies that were changed at Meta in the last three months of 2021. On average, switches can take upto 4.5 hours to update. Most of the switches are updated immediately, but some switches take very long to configure. The 95th percentile of configuration update can take as much as 20 hours. This is mainly caused due to switch failures that require additional time to make them operational again.

Changing policies (e.g., switching between collections of intents) can also have a large footprint, that is, it involves the configuration of many switches. For instance, configuring backbone policies typically involves changing configurations for a few FA switches. On the other hand, introducing a new technique to handle infrastructure prefixes (such as I_6), would involve a lot more switches, as they can go into a DRAIN state. Figure 2(b) shows the percentage of switches in our network that required configuration update for different policy changes. An average policy change requires configuration of at least 25.6% of switches. Reconfigurations are only going to get longer as we typically double the number of switches every five years [10]. Moreover, the time and complexity taken for configuring switches differ depending on a switch's role. On one hand, FSW switches carry only a small portion of traffic in our network. Given the redundancies of our network, a failure in a FSW switch usually does not cause issues in the network. On the other hand, aggregate switches, such as fabric aggregators, carry a larger portion of traffic, so their changes should be planned more carefully. Failures during reconfiguration of these aggregator switches can be of a higher consequence as they can disconnect buildings within datacenters.

3 Aura Design

To address the outlined challenges, we present Aura in Figure 3. The goal of Aura is to allow network operators to

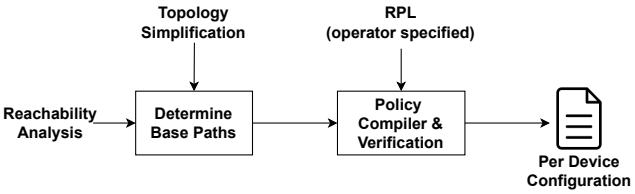


Figure 3: Aura architecture.

express flexible policies that are capable of handling various network reconfiguration scenarios at scale while minimizing disruption to production traffic. Aura handles the challenges outlined in §2 as follows:

- To support multiple configurations in a switch (as seen in § 2.2), Aura uses a set of base paths to pre-configure the network. Base paths are a collection of paths across different types of switches, which have the property that any propagation path in the network can be expressed as the base path or a sub-path of the base path. In §3.1, we explain how Aura uses topological features such as symmetry and hierarchy, along with reachability requirements to determine the set of base paths.
- To handle dynamic policies (as seen in § 2.2), Aura uses a labeling mechanism. If we are synthesizing into BGP configurations, Aura generates configurations that match on dedicated community tags and maps them to a corresponding policy. This results in multiple configurations defined in a single configuration file. In §3.2, Aura uses dedicated community tags that are attached by every switch to indicate its current state. Together with staging, this behaves as if there is a change between an old and a new configuration without the need for switch reconfiguration. Minimizing reconfigurations helps in supporting changes in policies at scale (as seen in §2.4).
- To express conditional policies (as seen in § 2.3), we introduce conditions that depend on switch state.
- We design a routing policy language (RPL) for Aura (§4), which uses base paths, switch conditions, route propagation conditions, and route preferences to express a set of policies.
- Finally, a compiler generates BGP configurations from RPL and tests configurations in the network (§5).

3.1 Base Paths: Minimizing Reconfiguration

Aura minimizes reconfiguration of large production networks by pre-compiling the network with a set of paths known as the *base paths*. Base paths are similar to pathlets, which are fragments of paths representing nodes that are willing to route [13]. The base path concept stems from the following two insights.

- *Policies can be defined on abstract paths:* Modern data center topologies are usually hierarchical, symmetric, and uniform. To be resilient to failures and ease the operation, most data centers classify switches into different roles according to their layers in the FAT-tree like topology [24]. For example, every switch in the lowest layer (RSW in our topology) has a south bound connection to servers in a rack, and a north bound interface up to the aggregate switches (FSW in our topology). Switches of the same role are functionally equivalent. Thus, base paths can be defined as abstract paths using these abstract switch roles. In turn, policy intents can be expressed by using the base path or a sub-path of the base path.

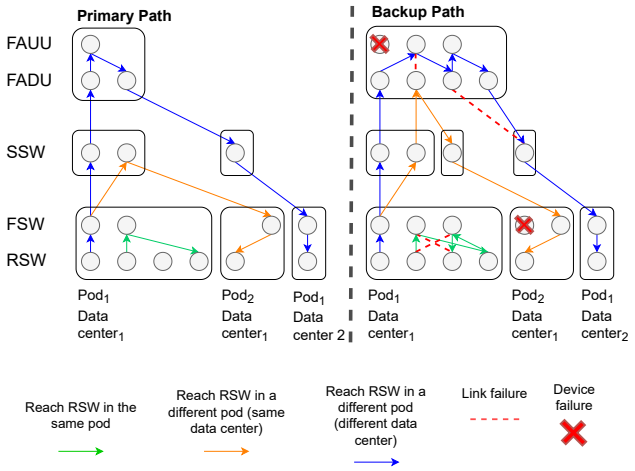


Figure 4: Extracting base paths.

- *Staging policies for dynamic scenarios:* Our base path set contains not only the preferred paths under a regular, static scenario, but also alternative paths under many dynamic scenarios, such as failures, migration, or maintenance as discussed in §2.2. When these changes occur, we simply need to select configurations that correspond to a different subset of base paths. Aura's synthesis has already deployed all configurations in the individual switches, and we just need to deactivate one configuration and activate the other.

We identify base paths by first *simplifying datacenter topology* and then *performing comprehensive reachability analysis* on that topology.

Topology simplification. We simplify a datacenter topology by abstracting multiple switches that share some given characteristic as a single switch. We can make abstractions at different granularities. For example, we can abstract all RSWs into a single node, or abstract all RSWs *in the same pod* into a single node. It is important to find the right granularity to maintain reachability without jeopardizing scale. Abstractions at too fine granularity create many paths and jeopardize synthesis scalability and operational maintainability. Abstractions at too coarse granularity (e.g., single role – single switch [7]) do not allow us to stage paths at different scenarios, such as intents I_1 and I_2, which requires traversing through specific switches in the network. Moreover, role-based abstraction can

express these policies, but configuring switches to support them is a challenge. For example, intent I_2 requires visiting the same switch role multiple times. Aura's compiler needs a way to keep track of propagating the announcement through such paths (see §5).

Reachability analysis. We find a balanced abstraction by performing a primary reachability analysis between RSW switches on the minimal topology. We then extend these paths to support alternative paths, to accommodate cases when nodes on a primary path fail. Any duplicate paths will be collapsed into a single path. Figure 4 illustrates this process. We start with three primary paths: one between two RSWs in the same pod, the second between two RSWs in different pods, and the third between two RSWs in different datacenters. Then we consider the failure of the direct FSW, direct SSW, or direct FA switch (or links connecting them) on the primary paths. For each failure we add nodes to express the backup path. Note that we first consider single switch or link failures on each primary path. Further, we consider larger failure scenarios such as regional failures and disasters. For example, we consider backbone failure and craft the paths to provide intra-region paths by traversing the FA layer. Currently, we generate base paths according to our reachability objective, which is, providing reachability of intents in the presence of at most two failures (either link or device or both)[2]. Note that the process of generating base paths is not unique to Meta's network. Any datacenter network can exploit its symmetry and hierarchical nature to derive its base paths.

3.2 Staging and Labeling: Supporting Dynamic Configurations

We support network changes, while minimizing reconfiguration via staging and labeling. These mechanisms help us support parallel configurations in switches (with only one configuration being active) and to support conditional intents.

- *Staging configuration snippets and activating them with labels* are the two key techniques to allow coexistence of multiple policies. Each policy is implemented as a set of configurations and loaded onto the switches. A policy may have a condition expressed as a combination of labels. Only when the conditions are satisfied, the policy will be activated.
- *Propagating label with routes* is the way we achieve conditional policies. The labels are translated into BGP community tags in our synthesis process, and are attached to routes and propagated through switches. Each switch can match the conditions based on the current switch state, attach its own state as community tags, and announce it to downstream switches.

[2]We determined any failures beyond that would degrade capacity, thus this would no longer be a reachability problem. We deploy other measures to handle such large-scale failures, which are beyond the scope of this work.

We provide two examples to show how staging and labeling support dynamic scenarios.

Supporting OpenR deployment. In our data centers, we are developing an alternative routing protocol known as OpenR [15] for better scalability. One challenge of applying Aura to production is how to gracefully migrate the switch configurations from an old to a new set of configurations. BGP and OpenR configurations are completely different and switching between them requires drastic changes to the fleet. We use Aura to support this process with no disruptions, as illustrated in Figure 5.

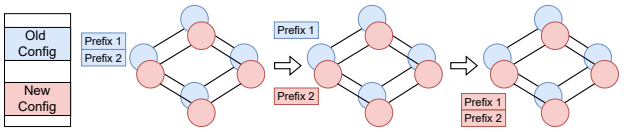


Figure 5: Staging and subscription used for migration.

Both versions of configurations are staged on all switches[3], each snippet starts with an activating condition, mapped into different BGP community tags. In Figure 5, we illustrate this as red and blue tags. Network operators can then attach tags to prefixes (prefix 1 or 2) to implement the policy for that particular prefix. By this, Aura simultaneously supports both the old and new policy. This technique also gives the network operator the flexibility to shift traffic in any order they like, and it also offers the fallback opportunity, should the new policy have some unforeseen effects on traffic when it is deployed.

Initially, all prefixes (prefix 1 and 2) use the old configuration (blue). We start with prefixes of less critical services first to avoid business disruptions. At the origin, $Prefix_2$ is announced with the red tag, so when the announcement arrives at intermediate switches, the corresponding red configuration is activated and the blue configuration becomes inactive. Other prefixes are gradually on-boarded to the new configuration by switching their tags in announcements. If the new configuration has any issue, we can safely switch back to the old version by controlling community tags at the origin.

Handling switch maintenance. As explained in §2.2, routes from LIVE switches are preferred over those from WARM switches (intent I_5), and routes from DRAINED switches should not be used (intent I_4). To reflect this policy, we stage parallel configurations on each switch[4] Each configuration describes actions for a given state of this switch and a given

[3]FBOSS (Facebook Open Switching System [10]) supports concurrent deployment of BGP and OpenR configurations.

[4]FBOSS supports parallel configurations for each device state via instances. Each FBOSS instance contains a local control place and communicates with the central network management system via thrift based service. Although there can be several instances, only the active instance uses the switch's hardware resources [10]. We believe any other switch could also achieve this by pushing necessary configurations from the control plane.

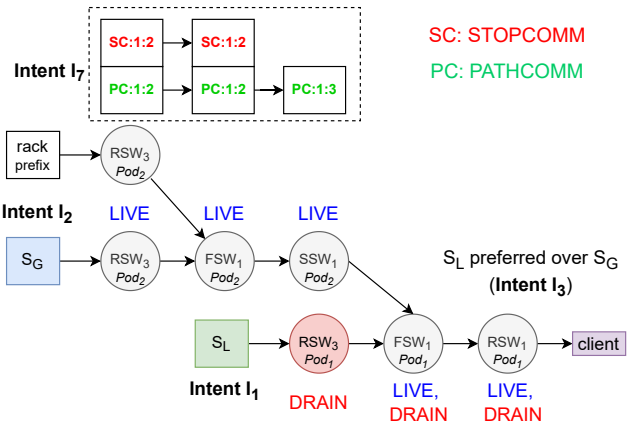


Figure 6: Supporting conditional policies.

state of the neighboring switch sending announcements to this switch. The actions could be adding/removing community tags, setting preferences, and accepting/rejecting routes. Taking intent I_1 and I_2 as an example, as shown in Figure 6, there are two paths to the client to reach the local service S_L and the global service S_G. Intent I_3 specifies that I_1 (primary) is preferred over I_2 (backup). The RSW switch along the primary path is going through a DRAINED state, therefore it attaches the DRAINED tag to its announcement. On the other hand, the backup path contains all live switches and the announcement only contains the LIVE tag. The downstream switches on receiving both the announcements would prefer the backup path to global prefix over the primary path to the local prefix. To support this, switches in both primary and backup paths should be configured to match the DRAINED tag and allocate a lower preference to the primary path. In §5, we show how to implement it with concrete BGP attributes.

4 RPL: Expressing Conditional Policies

Our routing policy language – RPL – allows a network operator to express high-level routing intents, by defining paths and path preferences. We had two choices while designing RPL – to make it a domain specific language (DSL), that is, its syntax is designed from scratch or to make it an embedded language based on Python. Embedded languages typically benefit from existing IDE, debugger, type ahead assistant and error messages, which allows for quick adoption by network operators. However, embedded languages are hard to verify. For instance, in Python, users can override basic operators making it hard to verify a program without running it. On the other hand, with DSL, the language itself can be defined in a way that it can reject wrong programs and enforce invariants we care about. We could also apply any number of static analysis techniques to determine the effect without running the program. Therefore, we traded-off the flexibility provided by embedded languages for correctness and verifiablity of DSL.

Syntax

name	::=	*string*	
rx	::=	*regular exp*	
lit	::=	*name* | ∼*name*	*names*
comp	::=	<| >| =	*comp*
bp	::=	*name {hops lit (->lit)*}*	*base path*
B_{bp}	::=	*base-paths {bp+}*	*base path block*
loc	::=	*name {regex-def rx (. rx)*}*	*location*
B_{loc}	::=	*locations {loc+}*	*location block*
tag	::=	*name rx*	*tag*
B_{tag}	::=	*tags {tag+}*	*tags block*
B_{top}	::=	*topology { name { B_{loc} | B_{tag} | B_{bp} } }*	*topology block*
B_{rout}	::=	*routing {topology lit}*	*routing block*
o	::=	*origins {location lit}*	*origin*
pc	::=	*propagate-condition { lit (,lit)* }*	*propagation condition*
p	::=	*name {hops lit (->lit)*}*	*path*
B_{path}	::=	*paths {p+}*	*path block*
B_{prop}	::=	*propogation {pc B_{path}}*	*propagation block*
pref	::=	*preference {lit (comp lit)+}*	*preference*
pol	::=	*policies { name { B_{rout} | *o* | B_{prop} | *pref* } }*	*policy*

Table 1: RPL block and leaf statements.

RPL's syntax is designed from scratch and is based on ANTLR [1] – an engine that provides basic syntax parsing. ANTLR also allows us to easily extend the support of RPL as and when our operational needs change. Table 1 shows the collection of statements supported by RPL. Statements are used to identify base paths, location of switches, describe tags and define policies with their preferences and conditions. Groups of these statements, known as "blocks," are used to describe different components of the policy. For example, the topology block (B_{top}), describes the topology containing the locations of switches, tags and base paths. The RPL program required to support all policies discussed in Figure 1 is shown in Figure 7. To handle the challenges of expressing flexible policies (§2.3), RPL supports the following features.

Minimizing reconfigurations: RPL allows network operators to specify topology block (shown in lines 1–23) that helps in pre-compiling the network. Specifically, in the topology block, the operator can specify scopes of devices (§3.2), tags (§3.2) and base paths (§3.1) that could be used by policies. For example, to support intents I_1 to I_7, topology block `f16` is sufficient. As described in §3.1, network operators are free

to specify any number of intents that could be supported by this topology block. If operators want to support other intents, they could always extend the topology block, but would require reconfiguration. From our operational experience, we show in §6 that these changes configure much lesser number of switches than competing approaches.

Supporting dynamic configurations: Network operators can specify different policy in the `policies` block of an RPL configuration file. This allows them to dynamically change policy over time. Each policy block, contains the `name` of the policy (e.g., RSW_REACHABILITY), `routing` (topology block used by the policy), `origin` (origination location of routes), `propagation` (set of paths used by the policy) and `preference` (preference among paths or tags). In Figure 7, policy RSW_REACHABILITY implements the intents I_1 to I_5, policy ALLOW_INFRA implements intent I_6 and policy LIMIT_RACK_PREFIX implements intent I_7. A network operator may choose to apply any one of these policies for appropriate prefixes. For instance, as described in an example in §3.2, network operators may apply RSW_REACHABILITY for services S_L and S_G and apply ALLOW_INFRA for infrastructure prefixes.

Expressing conditions with scopes: To support flexible granularity of intents, RPL introduces the notion of a *locations*, i.e., one can define a switch in RPL at different levels of granularity, such as switch role, switch ID, switch plane, fabric, region, and the datacenter. A location can be defined in the topology block (as shown in lines 3–12 in Figure 7). Location definition consists of the switch role, followed by switch number, pod number, fabric number, and datacenter name. Some elements can also be replaced by wildcards. The naming convention for our production network leverages the symmetry of the network to keep it simple and uniform. For example, the first spine plane in every data center would have the same identification number [5]. Network operators can leverage these naming conventions to define a scope. For example, I_1 requires the route to propagate through the first RSW present in Pod_1, present in the first fabric in the Altoona (ATN) datacenter, scope RSW.3.2.1.1.ATN can be used.

Expressing different switch states: RPL introduces the `tags` block to indicate tags that will be used to identify switches in a given state. An example of the tags block is shown from lines 13–17 in Figure 7, which defines tags LIVE, DRAINED and WARM. These tags will eventually be mapped in the synthesis process into BGP community tags, which would be attached from every switch to communicate the state of the switch to its neighbors (see §5 for implementation details). RPL also allows network operators to limit the propagation of a route announcement by using `prop-condition` statement. In Figure 7, we limit the propagation of announcements based on the switch state to only LIVE and WARM switches (line 29).

Handling exceptions: As per I_6, DRAINED state switches are required to carry traffic for infrastructure prefixes. Network operators can define a new policy ALLOW_INFRA, that allows DRAINED switches to propagate routes. Network operators can use the ALLOW_INFRA policy for applicable prefixes (as described in §3.2).

```
1  topology{
2      f16{
3          locations{
4              R1P1 { regex-def: RSW.1.1.1.1.ATN }
5              R3P1 { regex-def: RSW.3.1.1.1.ATN }
6              R3P2 { regex-def: RSW.3.2.1.1.ATN }
7              F1P1 { regex-def: FSW.1.1.1.1.ATN }
8              F1P2 { regex-def: FSW.1.2.1.1.ATN }
9              S1PL2 { regex-def: SSW.1.2.1.1.ATN }
10             FSW { regex-def:: FSW* }
11             RSW { regex-def:: RSW* }
12         }
13         tags{
14             LIVE L
15             WARM W
16             DRAIN D
17         }
18         base-paths{
19             B1 {hops RSW → FSW → RSW}
20             B2 {hops RSW → FSW → SSW → FSW → RSW}
21         }
22     }
23 }
24 policies{
25    RSW_REACHABILITY{
26         routing{topology f16}
27         origin{location RSW}
28         propagation{
29             prop-condition (L or W)← I4
30             paths{
31                 path P1 R1P1 → F2P1 → R3P1 ← I1
32                 path P2 R1P1 → F2P1 → S1PL2 → F2P2 → R3P2 ← I2
33             }
34         }
35         preference{
36             P1 > P2 ← I3
37             L > W ← I5
38         }
39     }
40     ALLOW_INFRA{
41         # Same routing, origin, paths as RSW_REACHABILITY
42         propagation{
43             prop-condition (L or W or D)← I6
44         }
45         preference{
46             L > W > D← I6
47         }
48     }
49     LIMIT_RACK_PREFIX{ ... } ← I7
50 }
```

Figure 7: RPL configuration describing policies.

5 Compiler Implementation

In this section, we discuss how Aura takes RPL specification and synthesizes BGP configurations for various switches. Aura uses properties of BGP to flexibly map the RPL specification to switch configurations.

5.1 Supporting Base Paths

The production network would need to be pre-compiled to support the base paths (specified in the topology block in RPL). To achieve this, Aura utilizes reserved community tags and generates rules to match these community tags. The key idea of using community tags is similar to that of source routing [26]. To implement a policy for a prefix, appropriate community tags are attached to the prefix while announcing it. On receiving the announcement, switches match on the community tags and take appropriate action of allowing or denying the route or modifying the tags. Although the idea seems straightforward, the challenge is how to come up with *systematic community assignment that can be interpretable and maintainable at scale*. Our key idea is an easy to interpret and debug encoding scheme that translates the base paths and other intents directly into different bits in the community attribute.

Encoding paths: Aura encodes the base paths into a structure called *path community tags* (PathComm for short). Depending on the number of base paths, Aura allocates a unique PathComm with a format of *PATHCOMM:P(1-10):H(1-6)*, where numbers in parentheses denote the bit sizes of the fields. The 10 most significant bits denote the unique base path ID P and the least significant 6 bits denote the hop number H. Some paths have switch roles that occur multiple times, e.g., intent I_2, where switch roles RSW and FSW occur twice. For such paths, the hop count field is used to keep track of where the announcement is on the path. Aura configures the switches to support intent I_2 as follows. At RSW on the first hop, the switch matches announcements that contain the PATHCOMM:1:1, modifies the community tag to PATHCOMM:1:2, (i.e., increments the hop count), and allows the announcement to be advertised to FSW. Similarly, Aura configures FSW to match, modify and advertise the announcement to the corresponding next-hop, that is RSW. When the announcement reaches RSW again at the last hop, the switch matches PATHCOMM:1:3, it stops announcing the announcement further. Therefore at RSW, there are two sets of rules, one that matches the announcement on the first hop (PATHCOMM:1:1) and the other that matches the announcement on the third hop (PATHCOMM:1:3). The use of the community tag ensures the abstraction of the switch role is maintained without needing to split roles of switches further to support multiple hops over the same switch type on the same path.

Containing announcements: Some intents require announcements to be contained to a specific location. For instance, intent I_7 limits rack prefix to pods (Figure 6). To support this, Aura uses a dedicated community tag known as *stop community* (StopComm for short). Similar to PathComm, StopComm follows the format of *STOPCOMM:P(1-10):H(1-6)*, where P and H are used to denote the path ID and the hop number to stop announcement respectively. For intent I_7, Aura appropriately configures RSW to stop propagating

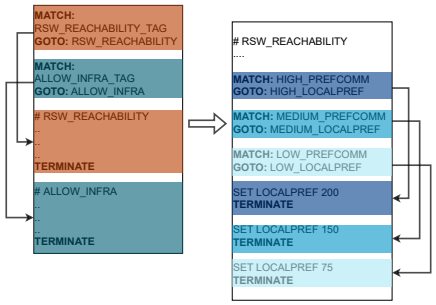


Figure 8: Aura generated configuration.

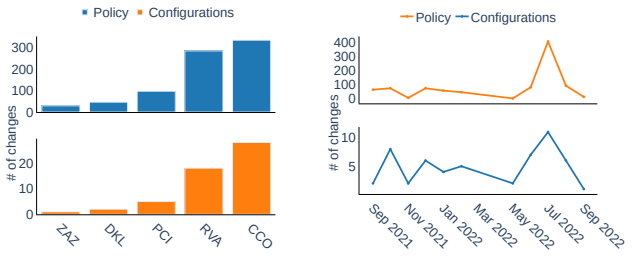
the announcement when it receives STOPCOMM:1:2 and restricts the announcement to pods.

During configuration synthesis, Aura generates appropriate match action rules to match on the corresponding communities and take appropriate action. That is, for PATHCOMM, the action would be to modify the PATHCOMM to reflect the hop changes and for STOPCOMM, it would remove the community tags and prevent the forwarding of the announcement.

5.2 Supporting Dynamic Network

Staging: Aura has reserved community tags to accommodate different policies. Aura's compiler incrementally allocates the community tags based on the order in which they are specified in RPL. For instance, from Figure 7, a dedicated community tag known as policy community (PolComm) will be allocated for RSW_REACHABILITY and ALLOW_INFRA policies. Similar to PathComm and StopComm, PolComm follows the format of POLCOMM:POL(1-16), where POL denotes the policy ID. Figure 8, illustrates how the staged community tags are used as a pointer to different segments of the BGP configuration file. BGP configurations are processed sequentially by switches. At the beginning of the configuration are match statements, matching the staged community tags. The corresponding action on a match is a GOTO statement pointing to the section of the configuration implementing the policy. At the end of the section is the TERMINATE command that terminates the processing of the configuration. We discuss consistency guarantee from a practical perspective in §7.

Supporting preferences among paths: Within each policy, network operators may specify preferences among intents (see §4). To accommodate this, Aura uses preference community (PrefComm) with a format of *PREFCOMM:X*, where X denotes the preference value. For every X value there is a one-to-one mapping to a local preference value. Currently, at every switch, there are twelve preferences matching twelve local preference values. If a switch receives an announcement with PrefComm, the configuration generated by the compiler contains a rule that matches on X and the action is a GOTO statement that implements the appropriate localpref. This is



(a) Region changes. (b) Changes over time.

Figure 9: Intents/policies generated by Aura.

also illustrated in Figure 8, where there are match statements for setting high, medium, and low local preference values.

5.3 Validating Configuration

To verify the correctness of the BGP compiler output, Aura uses an emulation-based, routing policy validation framework. We run a container-based high-fidelity emulation of FBOSS switch [10], which constructs an overlay network among containers emulating a small-scale data center without relying on hardware switches. The topology contains multiple switches of the same role to mimic cross-POD and cross-DC route propagation. We load the configurations generated by Aura to the BGP agent on the emulation switches. The switches exchange routes according to the rules in the BGP policy and broadcast End-of-Rib (EoR) messages on convergence. On receiving all EoR messages, the verification framework collects Routing Information Base (RIB) from all switches for validation. The validation algorithm verifies whether the routing status is identical to the RPL policy. Here, the basic idea is to traverse the switch network graph as specified in the RPL propagation path, and check whether all routing paths are correctly present as specified in RPL (Algorithm in Appendix §A). First, the algorithm checks whether the prefixes are originated as intended. It mines the originated prefixes from the first hop switches of the RPL propagation path and inserts the found switch and prefix information into a queue for Breadth-First Search (BFS) traversal. Then, it pops data from the queue, looks up the given switch and prefix and collects the RIBs from the switches. From the next switches' RIBs, the algorithm searches for the matching prefix and community tag and checks if the next hop of the path is the current switch. If so, it marks the routing path as *visited* and pushes the next path information into the queue to continue traversal. This procedure is repeated until the algorithm traverses all propagation paths in RPL. We also perform additional verification on the RPL (described in Appendix C).

6 Evaluation

We first measure the configuration changes made in our production data center, and use these measurements to evaluate Aura by showing how it minimizes switch re-configurations (§5.1), has a flexible language to express policies (§4) and overall reduces operator burden. Aura's compiler is implemented in Python, and has around 13.8 K lines of code. The routing policy verification is developed in Python with around 1,200 lines of code.

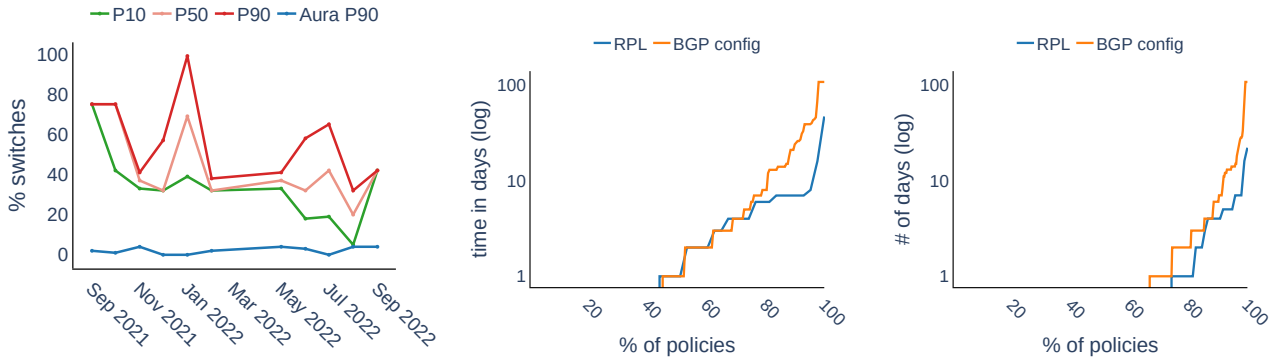
We capture the intent changes made by network operators for a year from Sept, 2021 to Sept, 2022. During this time, Aura supported five different data center regions, where each region has a separate configuration. Figure 9(a) shows the number of configuration and policy versions generated by network operators across all Aura-supported regions. We introduced Aura to CCO and RVA datacenter regions in September 2021 and thus these two regions have the highest number of changes. We eventually rolled it out to the remaining regions this year, with ZAZ being the most recent data center running Aura. Over time, across these data center regions, there were 54 different versions of configurations and 840K different policies. On average, there were 10.8 versions of the configurations and 168 versions of policies per region. There are fewer configuration changes than policy changes as not all policy change would require reconfiguration. Many policies would have already been pre-compiled by Aura in the network (as seen in §3.1 and §3.2).

Figure 9(b) shows a timeline of the policy and configuration changes that occurred across all Aura-supported data center regions every month. Every month on average, we changed 5 configurations and 87.3 policies. The largest frequency of updates was observed in July 2022, where we made several updates to introduce a new propagation path from the backbone network to our data centers. We discuss our experience of this roll-out in Section 7.

6.1 Minimize Switch Changes

To show the benefits of pre-compiling the network, we compare Aura with Propane [7]. Aura creates snapshots of policies and configures the network to support multiple common policies in parallel, as opposed to Propane, which compiles the configuration as and when there is a policy that needs to be supported in the network. A key drawback of Propane is that a change in policy may lead to reconfiguration of a large number of switches. We use simulation to quantify the benefit of our base path design in practice. We simulate a scenario where we replay all the policy changes made by network operators as shown in Figure 9(b). That is, whenever a policy is changed, we determine the switches that require re-configuring in case when Aura is run versus the case when Propane is run, to synthesize switch configurations.

Propane involves too many switch re-configurations: Fig-



(a) Percentage of switches reconfigured.

(b) Time taken to review.

(c) Number of revisions.

Figure 10: Aura performance.

ure 10(a), shows the number of switches in all datacenter regions supported by Aura that are reconfigured by Propane. For every month, we show the 10th, 50th and 90th percentile of switch re-configurations that are needed. As described in §2.4, this is time-consuming and disruptive to production traffic. Even the median number of policy changes on average can reconfigure 46.8% of all switches. Given the size of our networks, this could be in order of tens of thousands of switches. On the other hand, Aura reconfigures the network only when new policies cannot be supported by the pre-compiled configurations. Even for the 90th percentile of cases, Aura only reconfigures 2.1% of the switches on average.

6.2 Aura Reduces Operational Burden

During policy generation, Meta adopts a peer review process, where network operators generate a configuration and ask other operators to review the changes. In case of non-Aura-supported regions, network operators send the new BGP configuration for review. On the other hand, for Aura-supported regions, network operators send the new RPL configuration for review.

Non-Aura regions take longer to evaluate: Figure 10(b) shows the time (log scale) taken in peer review for Aura and non-Aura regions. On average, we find that non-Aura regions take 8 days per policy and Aura-regions take 3.6 days per policy. Some policies can take much longer to review than others. For instance, the most amount of review time taken for a policy by non-Aura-region was about 108 days. This policy introduced a new fairness goal in the fabric aggregation layer, which was carried out in phases where multiple policies in non-Aura-regions had to be implemented before the given policy. On the other hand, for Aura-generated policies, the maximum review time was about 47 days. Similar to the non-Aura case, this specific policy involved introducing new backup paths and had to be extensively tested along with other Aura-generated policies.

Non-Aura regions generate many more code revisions: One key factor for reviewing configurations from non-Aura-supported regions are code revisions. Once the reviewer gives feedback via code review, the author addresses the comments and gets back to the reviewer. This process goes on until the author has addressed all the comments and the reviewer has no other feedback. Since the raw BGP configurations are much harder to understand than RPL, revision process tends to be error prone. Figure 10(c) shows the number of days taken to generate a revision for Aura and non-Aura regions. On average, Aura-supported regions take 1.1 day per revision and non-Aura-supported regions have 3.2 days per revision. This difference is pronounced at the tail, where the maximum time taken for a revision was 22 days for Aura-supported region. However, for non-Aura-supported region, the maximum number of revisions was 107 days.

6.3 Aura Configuration Properties

Breakdown of RPL policies. All policy versions were encoded in 83.2 K lines of RPL by network operators. The most common statements used to define the policies include the propagation condition, propagation policy, origin and routing – this constitutes 1.9K–2.5K lines of code. There are also 382 preference statements. A large number of origin (2.5K), routing (2.5K) and path (2.3K) statements reflect the support for multiple backup paths in the network. Similarly, a large number of prop-condition statements, reflects the frequent need to restrict the scope of propagation in the network. Other statements including location, signature, base path, routing protocols and device states are used only once per configuration, and there were only 54 such statements.

Aura's Performance. Aura can be broken down into three stages: time taken to process and validate RPL, compiler execution time and the per-switch configuration generation time. On average, RPL processing and validation takes 4.3 seconds and the compiler contributes 2.9 seconds, both of

which are negligible in the entire process. The majority of the time is taken by BGP configuration generation, which takes about 88 seconds on average. This is expected, as the complexity of Aura mainly lies in the BGP compiler, which must synthesize BGP configurations for every switch in the network. Before Aura, network operators manually encoded intents into switch configurations. Based on our interviews with network operators, these configurations took about 26 weeks to generate, validate and safely deploy. Aura also scales when the number of policies increases, which typically occurs as the network grows. We artificially increase the number of policies to test the configuration synthesis times. We find that the synthesis times are about the same (about 90 seconds) for 50, 100 and 150 policies.

7 Operational Experience

RPL supporting verification for non-Aura regions. During Aura deployment, for the regions that are not Aura ready, i.e., those using legacy BGP configurations, operators still craft the intent in RPL. In this case, although RPL-based intent is not used to generate BGP policies, it is used for verification. The existing verification tool uses RPL as the source of truth and verifies it against manually generated BGP configuration. From this experience, RPL enables global verification before and during Aura deployment, which guarantees the correctness of the policy migration. There are other tools which uses RPL as an input. For instance, after the policy has been deployed, we constantly verify the routes in the produciton network and alert when unintended behaviors are observed. This tool uses RPL as the source of truth for verifying the intents.

Identifying latent failures. The verification approach helped identifying several errors during the Aura deployment, such as intent specification errors, faulty compiler logic, BGP agent bugs, etc. We summarize the identified issues in Table 2 in Appendix B. We now look into two issues. First, when we configured multiple backup paths to reach RSW within the same pod – a primary path ($RSW \rightarrow FSW \rightarrow RSW$), a backup path via a different FSW, and a second longer backup path ($RSW \rightarrow FSW \rightarrow RSW \rightarrow FSW \rightarrow RSW$). When the primary path fails, the wrong configuration was used, resulting in using the longer backup path, instead of the shorter one, causing performance degradation. The second issue was a drained FSW switch, which did not announce the infrastructure prefixes to the connected RSWs. When another FSW switch failed, the external world lost connectivity to the RSW's management plane, which should have existed through the drained FSW switch. Such latent failures are hard to detect in production as they only manifest themselves during certain scenarios and the verification helped fix these errors before deployment.

The verification approach is not perfect and has failed to prevent some issues. In one instance, a operator created a policy to inject a route to server using virtual IPs. The verification framework verified the propagation of the path to the server, but did not check the IP address usage. It turned out that the operator specified an incorrect IP address which actually belonged to a switch. This resulted in traffic destined towards the switch to be wrongly routed to the server. To prevent this, the verification approach is now planning to validate the IP address usage along before validating route propagation.

Consistency guarantee in practice: One concern for policy update is the consistency during convergence. Aura tackles the problem in a practical manner with three steps. First, we always drain a switch by moving the live traffic away, which is also called *disruptive config update* [10]. This prevents packet loss during transient state. Second, after the new path is enabled, we wait for a time period that is long enough for BGP convergence at a DC level. We conducted experiments to evaluate the convergence time scaling with the size of the network and chose the window to be tens of minutes. Finally, we use BMP monitoring to guarantee that the new routes are propagated in place before removing the drain configuration. The convergence time depends on the amount of traffic a switch carries. On one hand, switches that carry less traffic such as FSWs can take upto 6 seconds, but on the other hand, switches that carry more traffic such as FAUU can take 92 seconds.

Ease of operation: After implementing Aura in some datacenter regions, we see significant improvement in experiences of network operators. For instance, after the last year's outage [3], we wanted to provide additional backup paths which included allowing routes over DRAINED switches with a lower preference. In Aura, we can support it with only 4 lines of RPL code change (details shown in Appendix D, Figure 11). This change took less than an hour to implement. On the other hand, we still had to support this change for non-Aura-supported regions, and it took three experienced engineers 30 days to make manual changes. During this time, the policy went through 6 rounds of review with over 40 comments for changes in the peer review process.

Supporting new networks: Our data center topology is constantly evolving to react to various deployment constraints and to new business requirements. For example, the recent global supply chain shortage pressed us to have a more condensed topology to reuse available ports. To accommodate the growing AI workload, we are developing a new AI backend topology. Adapting routing policy to a new topology is not trivial. Before Aura's deployment, it took network engineers up to 6 months to support a new topology. As an anecdotal example, in a recent deployment topology with Aura in June 2022 (ZAZ) took only 3 weeks by a single engineer.

Unsupported policies: Aura currently does not support UCMP (Unequal Cost Multi-Paths). Typically, UCMP is used in traffic migration scenarios, where we add additional planes to the backbone layer. Under regular operations, the FAUU uses ECMP to equally distribute traffic to available backbone

planes. During migration, we would like to test the newly introduced backbone planes, before running them in full capacity. In such cases, we make use of UCMP to manipulate the amount of traffic going towards the backbone planes. This is an area of potential future work for Aura to support UCMP.

Coordination with non-routing policies: Aura focuses on specification and validation of routing policies. However, there are other policies that impact forwarding decisions. First, there are policies that are specific to services. Services have their own policies about the use of network resources (e.g., load balancing, replication) and can create varying traffic patterns that could lead to a routing policy adjustment. Future research is needed to streamline and coordinate intent changes between service and network layers. Second, there are the access control lists (ACLs) that restrict the flow of traffic across different network domains at Meta, which in turn affects routing policies. These ACLs also have a policy specification that is maintained by a different team. Detecting conflict and maintaining consistency across different intent management systems is an unsolved problem. Finally, while Aura manages all the data center routing policies, there are outside domains such as backbone and edge networks at Meta. These networks run different routing protocols and have their own policy intents. One of the extension of Aura is to support backbone routing intent using RPL, so that we can perform end-to-end verification. In the future, we plan to explore how to adapt Aura to other non-routing policies.

8 Related Works

Configuration synthesis: Propane [7], Propane/AT [8] and SyNet [11] use their own DSL or existing techniques to express intents. Then the intents are synthesized into low level configurations. Alternatively, rather than specifying intents, operators can partially specify the BGP configuration, and the synthesis approach can fill in the holes. Although these techniques help reduce operational burden, they cannot handle dynamic changes at scale without requiring constant reconfigurations. Propane [7] and Propane/AT [8] are also limited to BGP protocol, while Aura supports multiple routing protocols (BGP and OpenR). Finally, some configuration synthesis systems like SyNet [11] and ConfigAsure [21] are hard to scale to the size of large datacenter networks, like Meta. Beyond BGP configuration synthesis, Robotron [25] uses high-level intents to low-level device configurations with minimal human intervention. This involves, designing a network, generating appropriate switch configurations for the network and monitoring them. Aura also uses some Robotron components such as FBNet, and extends the flexibility of BGP configuration generation via synthesis.

Supporting dynamic configurations: Typically, systems focus on supporting dynamic configurations by switching from one configuration to the other. For instance, zUpdate [18] and Snowcap [23] determine a transition plan from one configuration to the other. However, these techniques involve shifting from one configuration to the other via several intermediate configurations. As seen in § 2.4, configuration updates to switches in a large network can take a lot of time, and transitioning across different configuration would only exacerbate this issue. Our technique of supporting multiple policies is similar to fast failover [17] and MPLS reroute [4] techniques, where backup options are pre-configured without the need of operator intervention. Alternatively, support for parallel configuration can be done via virtual routers such as VROOM [28] which is similar to FBOSS instances where each instance has its own control plane. However, the key difference is that only the active instance in FBOSS use the switch hardware resources, whereas all VROOM instances use the switch hardware resources.

There have been several other works in SDN [16, 19, 20], that help in transitioning from one configuration to the other. However, re-configuring in SDN context is different, than switch reconfiguration, as forwarding state is changed directly from a centralized controller, avoiding the challenges of a large distributed network. Moreover, there are problems unique to a SDN setting which are not applicable to BGP. For instance, planning rule updates [19] in SDN is crucial to avoid packet drops and unintended network behavior. However, in our setting, configuration change happens a phased manner, where traffic is drained from switches before a configuration update. This ensures that the transition between configuration do not intefere with traffic.

Expressing intents: In recent years, there have been many efforts to raise the level of abstraction for low level configurations. Jinjing [27] introduces LAI to express ACL update synthesis and Propane [7] introduces RIR to express constraints on policy. Propane's RIR cannot express intents across different scopes, whereas both LAI and RIR do not support device state specifications and preferences based on these specifications (e.g., I_4, I_5 and I_6).

9 Conclusion

Providing stable and efficient routing in large data centers is crucial. Existing synthesis systems generate configuration only once, but production networks require multiple reconfiguration to support their scale and dynamics. We present Aura, that enables network operators to express high-level intents to be automatically configured into the switch policy implementation with minimal reconfiguration.

Acknowledgment We would like to thank many of Meta colleagues who have contributed to this work over the years and toward this paper. These include Jason Wilson, Yan Cai, Maaz Mohiuddin, Hyojeong Kim, Jingyi Yang, Michael Liu and many others. We also thank our shepherd Dr. Soudeh Ghorbani and the anonymous reviewers to help making a better version of this paper. This work is supported by the National Science Foundation grants NeTS-2211383.

References

[1] Antlr (another tool for language recognition). https://www.antlr.org/.

[2] United airlines jets grounded by computer router glitch. https://www.bbc.com/news/technology-33449693, 2015.

[3] Update about the 4 october outage. https://www.facebook.com/business/news/update-about-the-october-4th-outage, 2021.

[4] Mpls traffic engineering fast reroute. https://www.cisco.com/en/US/docs/ios/12_0st/12_0st10/feature/guide/fastrout.html#wp1015327, 2023.

[5] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running BGP in data centers at scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 65–81. USENIX Association, April 2021.

[6] Alexey Andreyev, Xu Wang, and Alex Eckert. Reinventing facebook's data center network. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/.

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. *SIGPLAN Not.*, 52(6):437–451, June 2017.

[9] Richard Chirgwin. Google routing blunder sent japan's internet dark on friday. https://www.theregister.com/2017/08/27/google_routing_blunder_sent_japans_internet_dark/, 2017.

[10] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356, 2018.

[11] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In *International Conference on Computer Aided Verification*, pages 261–281. Springer, 2017.

[12] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[13] P Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. *ACM SIGCOMM Computer Communication Review*, 39(4):111–122, 2009.

[14] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Saif Hasan, Petr Lapukhov, Anuj Madan, and Omar Baldonado. Open/r: Open routing for modern networks. https://engineering.fb.com/2017/11/15/connectivity/open-r-open-routing-for-modern-networks/, 2017.

[16] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review*, 44(4):539–550, 2014.

[17] Ying-Dar Lin, Hung-Yi Teng, Chia-Rong Hsu, Chun-Chieh Liao, and Yuan-Cheng Lai. Fast failover and switchover for link failures and congestion in software defined networks. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, 2016.

[18] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 411–422, 2013.

[19] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[20] Jedidiah McClurg, Hossein Hojjat, Pavol Černỳ, and Nate Foster. Efficient synthesis of network updates. *Acm Sigplan Notices*, 50(6):196–207, 2015.

[21] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[22] Jenni Ryall. Facebook, tinder, instagram suffer widespread issues. https://mashable.com/archive/facebook-tinder-instagram-issues, 2015.

[23] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 33–49, 2021.

[24] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439, 2016.

[26] Carl A Sunshine. Source routing in computer networks. *ACM SIGCOMM Computer Communication Review*, 7(1):29–33, 1977.

[27] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[28] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus Van Der Merwe, and Jennifer Rexford. Virtual routers on the move: live router migration as a network-management primitive. *ACM SIGCOMM Computer Communication Review*, 38(4):231–242, 2008.

[29] Zhiping Yao, Hany Morsy, and Jasmeet Bagga. Opening designs for 6-pack and wedge 100. https://engineering.fb.com/data-center-engineering/opening-designs-for-6-pack-and-wedge-100/.

A Routing Policy Validation Algorithm

Algorithm 1 shows the algorithm used during emulation to verify Aura generated configuration.

Algorithm 1 RPL-based routing policy validation

```
1:  procedure traverse(path, tag)
2:      ▷ path: a propagation path from RPL
3:      ▷ tag: a community tag that identifies policy

4:      ▷ Enqueue the originated prefixes into BFS queue
5:      Q = Queue()
6:      for sw in get_switches(path[0], all) do
7:          for R in RIB(sw) do
8:              if tag in R.tag then
9:                  visited_sw = [sw]
10:                 L = 0
11:                 N = node(sw, route.prefix, L, visited_sw)
12:                 Q.enque(N)

13:     ▷ Check if the prefix exists in RIBs
14:     while !Q.isEmpty() do
15:         N = Q.pop()
16:         L = N.L + 1
17:         for sw in get_switches(path[L], N.sw) do
18:             if sw in N.visited_sw then continue
19:             route_found = False
20:             for R in RIB(sw) do
21:                 if R.prefix == N.prefix
22:                     and tag in R.tag
23:                     and R.nexthop == N.sw then
24:                     route_found = True
25:                     R.visited = True
26:                     if L < length(path) then
27:                         visited_sw.append(sw)
28:                         N_new = node(sw,R.prefix, L, visited_sw)
29:                         Q.enque(N_new)
30:             assert route_found == True

31: procedure routing_policy_validation(tag)
32:     prop_paths = get_propagation_paths_from_rpl(tag)
33:     for path in prop_paths do
34:         traverse(path, tag)
35:     ▷ Verify if there is any prefix leakage
36:     for sw in get_switches(all, all) do
37:         for R in RIB(sw) do
38:             if tag not in R.tag then continue
39:             assert R.visited == True
```

B Issues detected via emulation

We show the issues detected during emulation in Table 2. Broadly, they are categorized into compiler errors and BGP agent errors.

C Detecting Ambiguous Statements

Once RPL specification is complete, we use it as input into a compiler, which generates switch configurations. To avoid ambiguity, Aura compiler performs certain verification steps on the RPL specification.

Rule 1: Explicitly specify preference across all paths. It is possible that an operator designing a policy does not specify preferences across all the paths. For example, if there were four paths, P1–P4, and the preference rule stated $P_1 > P_2 > P_4$, preference for P3 is unspecified. One possible approach would be to assign a default preference value. However, depending on the preference values assigned to the other three paths, the total ordering of paths is unpredictable. For instance, if the default preference value is 100, there could exist two sets of ordering, $P_1(100) = P_3(100) > P_2(50) > P_4(25)$ or $P_1(200) >$

Category	Identified Issues
Compiler	Leaked intra-fabric prefix to FA Announced FA loopback address to unwanted scope VIP prefixes not withdrawn from drained devices VIP priority misconfigured Prefix originated from SSWs in one DC was not propagated to another DC Drained SSW did not forward default route to FSWs Prefixes from backbone not propagated to FADUs Drained FAUU does not withdraw default route FA loopback addresses leaked to RSW
BGP agent	Unsupported BGP action Config parsing crash in switches

Table 2: Identified issues by emulation.

$P_2(100) = P_3(100) > P_4(50)$. This non-determinism could make debugging routing behavior more challenging, and even worse, result in a hidden intent violation. Our solution is to detect underspecified preferences during verification and prompt the operator to explicitly define each preference.

A similar issue arises when the operator specifies two parallel preferences: $P_1 > P_2, P_3 > P_4$. The order between paths in these different preferences can be interpreted as $P_1 > P_2 > P_3 > P_4$ or $P_3 > P_4 > P_1 > P_2$, and in several other ways. When verifying a RPL specification, Aura notifies operators to define ordering between all paths.

Rule 2: Detect hidden conditions. RPL supports adding a condition to a preference (Section 4). For example, the preference $P_1(BB_DEFAULT_ROUTE) > P_2 > P_3 > P_4$ means that for an announcement containing a community $BB_DEFAULT_ROUTE$, P_1 should have higher preference than other paths. However, there is an ambiguity. If the $(BB_DEFAULT_ROUTE)$ community is not attached, then the preference for P_1 is not specified, and the order of P_1 $(\neg BB_DEFAULT_ROUTE)$ and the rest of paths is unspecified as well. During verification, the operator is prompted to clear this ambiguity by explicitly specifying the preference for P_1 with and without the attached condition, and ensuring that the partial ordering is maintained.

D RPL changes to change intent

Figure 11 shows the RPL change done by network operators to allow DRAINED switches to propagate routes with a lower preference.

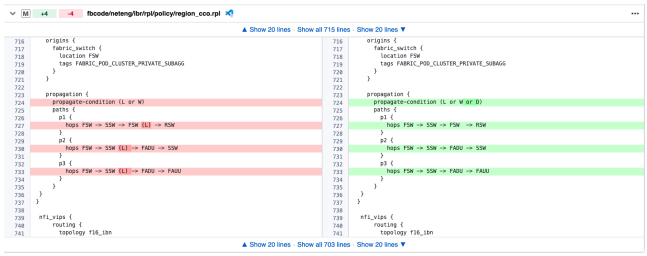


Figure 11: Aura changes to support I_2.

Formal Methods for Network Performance Analysis

Mina Tahmasbi Arashloo
University of Waterloo

Ryan Beckett
Microsoft Research

Rachit Agarwal
Cornell University

Abstract

Accurate and thorough analysis of network performance is challenging. Network simulations and emulations can only cover a subset of the continuously evolving set of workloads networks can experience, leaving room for unexplored corner cases and bugs that can cause sub-optimal performance on live traffic. Techniques from queuing theory and network calculus can provide rigorous bounds on performance metrics, but typically require the behavior of network components and the arrival pattern of traffic to be approximated with concise and well-behaved mathematical functions. As such, they are not immediately applicable to emerging workloads and the new algorithms and protocols developed for handling them.

We explore a different approach: using formal methods to analyze network performance. We show that it is possible to accurately model network components and their queues in logic, and use techniques from program synthesis to automatically generate concise interpretable workloads as answers to queries about performance metrics. Our approach offers a new point in the space of existing tools for analyzing network performance: it is more exhaustive than simulation and emulation, and can be readily applied to algorithms and protocols that are expressible in first-order logic. We demonstrate the effectiveness of our approach by analyzing packet scheduling algorithms and a small leaf-spine network and generating concise workloads that can cause throughput, fairness, starvation, and latency problems.

1 Introduction

New network functionality is often analyzed, both empirically and analytically, to predict how it would perform under different kinds of input traffic. However, it is growing increasingly difficult to do such analysis in a manner that is reasonably *exhaustive*, i.e., does not miss traffic patterns that are probable to occur in production, and *general*, i.e., is not only applicable to a limited set of network functionality.

To see why, consider the existing empirical approaches, i.e., using simulators, emulators, and testbeds [1–3]. These approaches allow operators to realize a model of their network in software and/or hardware, push a concrete sequences of packets through it, and measure how well the network performs for that specific input traffic pattern. As such, operators have to pick and choose which input traffic patterns to try. This leaves room for unexplored traffic patterns that may experience poor performance because of overlooked corner cases and bugs. The problem is only exacerbated as new networked applications emerge, expanding the set of traffic patterns that a network may face in production.

On the analytical side, there is a substantial body of work that applies techniques from queuing theory and network calculus to derive bounds on performance metrics such as throughput, delay, and loss [4–9]. However, to obtain tight and useful bounds, the network functionality and the arrival pattern of traffic need to be closely approximated with concise and well-behaved mathematical functions. This limits the set of network functionality and traffic patterns that can be reasoned about using these frameworks [10], a problem that is, again, aggravated with the continuous evolution of networked applications and the network functionality that supports them.

This paper explores an alternative approach: *using formal methods to analyze network performance*. That is, we would like to use logical formulas to model packets, how packets are processed by each piece of network functionality as they traverse the network, and how performance metrics such as throughput and delay change over time. We can then use verification and synthesis techniques from the formal methods community to exhaustively explore the space of all possible traffic patterns and find those for which the network cannot provide satisfactory performance.

Why use formal methods? Because they can nicely complement existing empirical and analytical approaches for performance analysis. Unlike empirical approaches, they do not need to explicitly try out every single traffic pattern to find ones that experience performance problems. Moreover, they enable us to reason about network functionalities that are expressible in logical formulas, many of which may be not feasible to approximate in a way that is suitable for existing analytical approaches. Finally, past efforts in using formal methods to solve networking problems have proven quite successful. Over the past decade, researchers and practitioners in both academia and industry have coupled advances in formal methods tools and techniques with domain-specific optimizations to rigorously reason about the functional correctness of various aspects of networks [11–28]. This provides further encouragement that formal methods could bring similar benefits to reasoning about *performance*.

Realizing this vision is not without its own unique challenges. First, performance metrics (e.g., throughput and delay) are statistics over packet streams and are affected by the order and the time at which packets from competing traffic enter and exit network components. As such, reasoning about performance requires reasoning about the enormous space of possible packet-level interactions and finding those that can lead to unsatisfactory performance. For that, we have found efficient ways to encode interactions that can significantly affect performance, e.g., interactions across network queues, in Satisfiability Modulo Theories (SMT) formulas and over a bounded number of time steps.

Second, when it comes to performance analysis, finding a single counter-example is not always useful. In this case, a counter-example is an assignment to the model's logical variables that leads to poor performance, where the model variables describe packets and the order and time at which they enter the network. Such a detailed packet trace is not easily interpretable. Moreover, unlike counter-examples for correctness properties, it may not even point to a bug, but an extremely rare scenario where providing sub-optimal performance does not have tangible consequences.

We argue that a more useful output (also a more challenging one to produce) is a traffic pattern or *workload* that can succinctly capture the commonality between a whole *set* of packet traces that can experience poor performance. To generate such workloads efficiently, our main insight is to use syntax-guided synthesis (SyGuS) [29]. That is, we define a language for specifying workloads. Then, we systematically search through the space of all workloads to find one such that *all* packet traces represented by it experience poor performance. The search algorithm, inspired by prior work on invariant synthesis [30], is based on Markov Chain Monte Carlo (MCMC), with a cost function that guides the algorithm towards the parts of the search space where there is a higher chance of finding an answer.

To demonstrate the effectiveness of our approach, we apply it to packet scheduling algorithms as well as a small leaf-spine network and ask queries about throughput, fairness, latency, and starvation. The fact that packet sequences are bounded, as well as other optimizations in our SMT encoding, enables our framework to analyze each workload in $< 1s$ on average. Moreover, our search algorithm can successfully find workloads that convey high-level insights about traffic classes that can experience transient or persistent performance problems in 6-18 minutes.

These results provide an encouraging indication that using formal methods to analyze network performance has the potential to grow into a valuable tool for understanding network behavior and making networks more robust. Our proof-of-concept prototype can analyze compositions of a few tens of network components, modeling small-scale networks or host-based scenarios where flows from different applications or VMs contend for end-host resources across a few layers of classifiers and packet schedulers. That said, we recognize that, similar to data and control-plane verification tools that have matured over a decade to scale to large-scale networks [31], much work needs to be done to improve the scalability of formal methods tools for network performance analysis. We discuss potential future research directions in §9.

2 Overview and Motivation

In this section, we use a buggy packet scheduler to demonstrate our approach in using formal methods to reason about performance.

The scheduler. FQ-CoDel [32] is the default queuing discipline in Linux. It is a hybrid packet scheduler and active queue management (AQM) algorithm. Flows are classified into queues which are managed by the CoDel AQM algorithm. The scheduler decides which queue gets to transmit next. It prioritizes the transmission of the first few packets of new flows so that short latency-sensitive flows with a few packets are not blocked by longer flows.

Our motivating example is inspired by the scheduler in FQ-CoDel. When a packet comes in, it is first classified (e.g., based on its 5-tuple) and assigned to a queue. For simplicity, let's assume hash collisions are rare and each queue is holding packets of one flow at any point in time. The scheduler maintains a list of pointers to queues with potential short flows called *new_queues*, and another list called *old_queues* with pointers to all the other queues with outstanding packets. At a high level, queues in *new_queues* are prioritized over those in *old_queues*, and the queues in the same list are serviced using a deficit-round-robin (DRR)-like algorithm [33].

How does the scheduler determine which list each queue belongs to? Suppose the incoming packet is assigned to q_i. If q_i is in neither of the lists, it means it has not received packets recently and this could be the start of a short flow. So, the scheduler will add a pointer to q_i to the end of *new_queues*. Otherwise, the queue will remain in its current list. In both cases, the packet is enqueued in q_i.

On dequeue, the scheduler first looks at *new_queues*. Suppose q_h is the queue at the head of the list. It will be selected to send a packet unless (1) it is empty, in which case it is removed from the new_queues and marked as inactive, or (2) it has packets but has already sent at least a (configurable) quantum of bytes, in which case it is not considered a short flow, is removed from *new_queues* and is inserted at the end of *old_queues*. If q_h is not eligible, the scheduler moves on to the next queue in *new_queues*.

The bug. When a queue in *new_queues* is empty, it is immediately deactivated. When it receives another packet, it is placed in *new_queues* and gets priority over the queues in *old_queues*. This can potentially cause starvation for queue in *old_queues*. When proposing the separation between new and old queues, the FQ-CoDel RFC [32] warns against this bug: "the queue could reappear (the next time a packet arrives for it) before the list of old queues is visited; this can go on indefinitely,

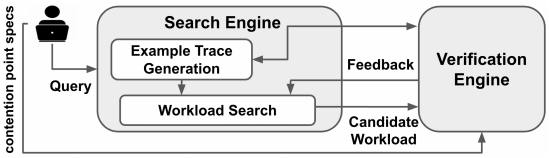


Figure 1: Overview of workload synthesis.

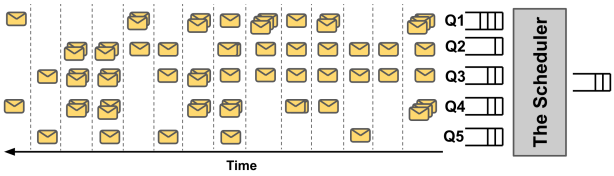


Figure 2: An example trace.

even with a small number of active flows, if the flow providing packets to the queue in question transmits at just the right rate." To avoid this problem, the RFC suggests that when a queue in *new_queues* becomes empty, it should be first demoted to the *old_queues*, and only deactivated if it still stays empty after all the old queues are visited on subsequent dequeues.

This is a subtle bug that is difficult to catch with existing approaches for performance analysis. The traffic pattern that reveals the bug consists of a flow sending packets at a very specific rate, which could depend on the number and traffic pattern of other active flows that traverse the scheduler. As such, it is not likely to be part of the common traffic patterns that are tried out in simulation and emulation and can be overlooked in empirical experiments. Similarly, approaches based on queuing theory and network calculus focus on schedulers and arrival patterns that have concise and well-defined mathematical approximations and cannot be readily applied to this specific variation of DRR scheduler or this traffic pattern.

2.1 Using Synthesis to Analyze Performance

Figure 1 shows an overview of our approach using formal methods and workload synthesis to reason about the performance of network components like our example scheduler.

Modeling contention points (§3). First, the users specify which network component(s) they are interested in, and if there are more than one, how those components are connected together. Given this input, the verification engine generates logical variables and constraints that model the network components and their interactions. Each component is modeled as one or more *queuing modules*, each with n input queues, m output queues, and a processing block in the middle. The processing block takes packets from input queues, processes them, and puts the resulting packets into output queues.

Here, we model the scheduler as a queuing module with five input queues and one output queue (Figure 2). Specifically, in the verification engine, we generate SMT formulas that model these queues and their content for T consecutive time steps, where time advances on every dequeue operation. Every time step, each input queue receives a bounded number of packets. Moreover, the processing block selects one input queue according to the scheduler's dequeue logic, dequeues a packet from it, and places the packet into the scheduler's output queue. The formulas describe how the scheduler state, e.g., *new_queues* and *old_queues* lists, and queue contents at time t connects with those at time $t+1$.

Performance queries (§4). Next, the user asks a query about a performance metric of interest such as throughput, latency, or fairness. Since the FQ-CoDel scheduler is supposed to provide fairness, suppose the user asks whether or not one queue can take more than its share of the bandwidth:

$$\underbrace{\left(\wedge_{i=1}^{4}\, \forall t \in [1,T],\, cenq(Q_i,t) \geq t\right)}_{\text{assuming other queues are backlogged}} \rightarrow \quad \underbrace{cdeq(Q_5,T) > 2\lfloor T/5 \rfloor}_{\text{can } Q_5 \text{ get more than its fair share?}}$$

where $cenq(q,t)$ and $cdeq(q,t)$ are the number of packets respectively enqueued into and dequeued from queue q by the end of time step t. The query can be read as "if queues Q_1 to Q_4 are backlogged the entire T time steps, is it possible for Q_5 to dequeue more than twice its fair share (i.e., $\lfloor T/5 \rfloor$)?". Note that we use lower case letters for variables and upper case letters for literals and constants.

Introducing workloads (§5). At this point, we can ask the verification engine to use an SMT solver like Z3 [34] to find an input packet *trace* that satisfies the query. A packet trace is simply an assignment to the variables that represent how many packets enter each queue in every time step. Figure 2 shows an example trace, found by the verification engine, that satisfies the query. While the trace does provide a concrete scenario in which Q_5 receives more than its fair share of the bandwidth, it is not easy to interpret as it specifies the ordering and timing of the entry of every single packet: is the fact that three packets entered Q_1 in the same time step in the beginning crucial to Q_5 getting a larger share of the bandwidth or is it just an arbitrary choice? Would the query still be satisfied if instead Q_2 had received three packets in the first time step? Even if these details actually do matter, it is not clear if the trace is actually pointing to a subtle but prominent performance problem, or just an extremely rare and uninteresting scenario.

Rather than output a single packet trace, our search engine synthesizes a *workload* that can concisely describe a set of packet traces that can cause performance problems:

$$\begin{aligned} &\forall t \in [1,6] : cenq(Q_5,t) \leq 1 \\ \wedge\, &\forall t \in [7,14] : aipg(Q_5,t) \geq 2 \\ \wedge\, &\forall t \in [13,14] : cenq(Q_5,t) \geq 5 \\ \wedge\, &\underbrace{\left(\wedge_{i=1}^{4}\, \forall t \in [1,14],\, cenq(Q_i,t) \geq t\right)}_{\text{backlog assumption from the query}} \end{aligned}$$

$cenq(q,t)$ is the total number of packets that enter q by $t+1$, and $aipg(q,t)$ is the inter-packet gap between the last two packets entering q by $t+1$. Such an answer is more interpretable as it captures the commonality of a set of traces that satisfy the query. Moreover, the fact that a set of similar packet traces all cause the same performance problem is a preliminary indication that it represents more than just a rare scenario.

We define workloads as a conjunction of constraints, each specifying a traffic pattern for one or a subset of queues over

a period of time. For $T = 14$, the above workload specifies a traffic pattern that will always satisfy the query, i.e., cause Q_5 to dequeue at least five packets when it should not have dequeued more than three. The first constraint states that at most one packet enters Q_5 in the first 6 timesteps, so that unlike the other four queues, Q_5 is not demoted to the list of old queues. After that, the second constraint ensures that Q_5 receives traffic at a specific rate, at most one packet every other time step. This ensures Q_5 becomes empty and gets deactivated after dequeuing each packet (the bug). So, it is activated as a new queue when it receives its next packet and is prioritized over others for dequeue. The third constraint ensures that Q_5 receives at least five packets by $T = 14$, so it has enough packets to dequeue and satisfy the query. The final constraint ensures that the other queues are always backlogged, which comes from the assumption specified in the query.

Synthesizing workloads (§6). The search engine is responsible for generating a workload that satisfies the query. It first uses the verification engine to generate a set of example traces that can guide the search towards finding a suitable workload. Next, inspired by prior work on program and invariant synthesis [30, 35], it starts a stochastic search process based on Markov Chain Monte Carlo (MCMC) that synthesizes a candidate workload and asks the verification engine to verify if all traces in that workload satisfy the query. If not, the violating trace is returned to the search engine and added to the example traces to help guide finding the next candidate workload. This process repeats until an answer is found and returned to the user, who can either terminate the analysis or ask for other workloads that satisfy the query.

User Interface. Similar to many other existing work that use formal methods for networks, our queries and workloads are specified as logical expressions. Moreover, the encoding of packet processing algorithms and protocols into logic is done manually. To enable widespread adaptation of formal approaches, it is important to develop front ends that interface with these "logical backends", abstract away the details of logical expressions and formulas, and enable users to interact with them using higher-level and more familiar interfaces. In fact, there are several ongoing efforts for providing higher level query interfaces and automated generation of logical models and SMT formulas from implementations [36–40]. With the right interface in the middle, the front-end and the logical back-end can evolve independently. As such, for our case study in §8, we create a simple front-end to demonstrate an example of one such interface, and leave the design of a more general front-end for future work.

3 Modeling Contention Points

Queues are an integral part of networks. In fact, networks can be viewed as multiple layers of queues with well-defined functionality in between that describes how to deliver data from one layer to the next. From source to destination, packets go from socket buffers, to queues in packet schedulers

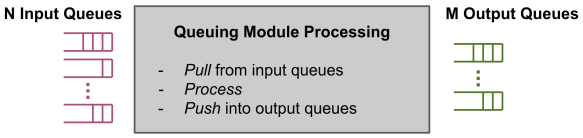


Figure 3: A queueing module.

in the end-host stack (e.g., Linux qdiscs), to NIC transmit queues on the sender. Then, they traverse switch input and output queues in the network, and the NIC, qdisc, and socket buffers on the receiving end. Network contention points, e.g., switches, NICs, and network stacks, heavily rely on queues to decide how to allocate network resources to competing traffic streams. Indeed, performance metrics of a packet stream are significantly affected by the queues it traverses and the frequency at which those queues are selected to pass on their traffic. As such, queues are a fundamental part of our model.

Figure 3 shows a simple yet expressive building block for modeling layers of queues: a *queuing module* with $n \geq 1$ input queues, $m \geq 1$ output queues, and a processing block in the middle. Every timestep, the processing block takes a batch of packets from the input queues, processes them, and puts the resulting packets into the output queues. To keep track of performance metrics, we designate extra variables per queue for each metric that get updated as packets enter and exit queues, and ask queries about their values for different queues (see §4 for examples).

Composition. Queuing modules can be easily composed by feeding the output queues of one module to the input queues of another (e.g., figures 6 and 9). So, one could start by modeling a single bottleneck, e.g., a qdisc, or a NIC/switch scheduler, and compose them together to reason about segments or paths in the network (§7 and §8). We can even close the loop by designating a queuing module for congestion control algorithms, with one input queue receiving acks and other control packets and one output queue transmitting data packets. While we have not explored this last direction in this paper, we believe it is a very interesting avenue for future work.

Modeling Time. We define a time step as the time between the dequeue operations of the slowest output queue. That is, time advances when a new dequeue happens. Modeling time based on dequeues is a natural choice. It is coarser-grained than wall-clock time, and since our verification engine performs bounded model checking over time, this helps performance problems manifest over fewer time steps. It is also fine-grained enough for capturing the arrival order of packets within and across time steps. To capture the arrival order within a time step, we put an upper bound K on the number of packets that can enter a queue between dequeues. This bound allows us to create K variables $p_{t_1}, \cdots, p_{t_K}$ to capture the order at which those K packets enter the queue at time t (p_{t_i} comes before p_{t_j} if $i < j$). It also helps us avoid finding "trivial" workloads that simply flood the queuing module every time step with an unrealistically large number of packets.

Modeling Packets. We model packets as tuples consisting

of multiple "metadata" variables. Depending on the query, these variables can include the arrival and departure time of the packet into and out of different queues of interest, flow id, application id, or packet size.

Modeling Queues. A queue is specified with two parameters, its size S and the maximum number of enqueues K allowed at every time step. We define three sets of variables for each queue for every time step t: (1) $enqs[t][1:K]$ consists of K tuples to capture the packets that are sent to the queue at time t, (2) $elems[t][1:S]$, consists of S tuples to capture the packets that are inside the queue at time t, and (3) an integer variable $deq_cnt[t]$ that captures how many packets will be dequeued from this queue at time t. We constrain these variables such that they collectively behave like a FIFO. Finding the right constraints to model FIFOs in a scalable manner is not straightforward, especially when there are multiple enqueues and dequeues per time step. We describe our encoding of FIFOs in SMT in Appendix A.

When two modules are composed, we need extra constraints to move packets from the output queue of one to the input queue of the other. Suppose the first output queue of module A ($A.out_1$) is connected to the first input queue of module B ($B.in_1$). B decides how many packets to dequeue from $A.out_1$ at time t and sets its $deq_cnt[t]$ to, say, k. We add constraints to enqueue the first k packets in $A.out_1$ into $B.in_1$. We provide two different kinds of composition. In *sequential* composition, $B.in_1.enqs[t][1:k] = A.out_1.elems[t][1:k]$, that is, packets from $A.out_1$ will appear in $B.in_1$ at time $t+1$. In contrast, in *immediate* composition, packets leaving $A.out_1$ at time t will be visible in $B.in_1$ in the same time step. We discuss examples of different kinds of composition and their implications in our case studies in §7 and §8.

Modeling packet-processing algorithms. The packet-processing algorithm in a queuing module sets the $deq_cnt[t]$ variables of the module's input queues to denote how many packets will be dequeued from each input queue at time t. It also decides which one of those packets to move to the $enqs[t]$ of which of the output queues, potentially changing, dropping, or cloning some packets in the process. As long as an algorithm's logic can be expressed in SMT, we can plug it into our framework.

4 Performance Queries

Performance queries are logical formulas over one or more performance metric. Users can define their metrics of interest to be tracked for all or a subset of queues in the queuing modules. They can then ask queries about the value of a certain metric for one queue or a set of queues, or compare the values of metrics between different queues.

Performance metrics. A performance metric $m(q,t)$ is a function that computes a value over the packets that have entered or departed the queue q until the end of time step t. Most metrics can be defined as recursive functions over time. For instance, metric d that tracks the maximum delay experienced

$$
\begin{array}{l|l}
\mathsf{qry} := \mathsf{wl} \rightarrow \mathsf{tr} : \mathsf{qlhs} \oplus \mathsf{rhs} & \mathsf{wl} := \mathsf{true} \,|\, \mathsf{con} \wedge \mathsf{wl} \\
\mathsf{tr} := \{\forall \,|\, \exists\} t \in [T_1, T_2] & \mathsf{con} := \forall t \in [T_1, T_2] : \mathsf{lhs} \oplus \mathsf{rhs} \\
\mathsf{qlhs} := \mathsf{lhs} \,|\, m(Q_1,t) - m(Q_2,t) & \mathsf{lhs} := m(Q,t) \,|\, \Sigma_{q \in S}\, m(q,t) \\
 & \mathsf{rhs} := C \cdot t \,|\, C
\end{array}
$$

Figure 4: Syntax for queries (§4) and workloads (§5).

by packets in a queue, can be defined in the following way:

$$
\begin{aligned}
d(q,0) &= 0 \\
d(q,t) &= max(t-a,\ d(q,t-1)) \\
&\quad \text{where } a = min_{p \in depart(q,t)}\, arrival(p,q)
\end{aligned}
$$

where $depart(q,t)$ is the set of packets that depart from queue q at time t, and $arrival(p,q)$ is packet p's arrival time into q. In defining metrics, one can use simple operations such as addition, subtraction, multiplication with a constant, and taking the maximum and minimum between values.

Queries. Queries are logical formulas over performance metrics. As shown in Figure 4, they ask questions about the values of metrics for one queue or a set of queues, or compare the values of metrics for two queues, over a certain period of time.

For instance, using the metric d defined above, we can ask if packets could face significant delay in queue Q with the following query: $\exists t \in [1,T] : d(Q,t) > D$. Moreover, as part of the query, users can specify base_wl, a workload (formally defined in Figure 4 and §5) that constrains the space of traces the user is interested in. That is, the final workload returned by the search algorithm should be a subset of base_wl. For instance, suppose we want to know, assuming a minimum input rate R for a queue Q, whether it will transmit fewer than K packets during T time steps. For that, we can use the following query:

$$
\underbrace{\forall t \in [1,T] : cenq(Q,t) \geq R \cdot t}_{\mathsf{base_wl}} \rightarrow \exists t \in [T,T] : cdeq(Q,t) < K
$$

where $cenq(q,t)$ and $cdeq(q,t)$ track the total number of packets that have entered and exited q by end of t.

Similarly, we can investigate fairness between two queues by assuming minimum input rates for both in base_wl and comparing the number of packets they transmit:

$$
\begin{aligned}
\mathsf{base_wl} &= \forall t \in [1,T] : cenq(Q_1,t) \geq R_1 \cdot t \\
&\quad \wedge \forall t \in [1,T] : cenq(Q_2,t) \geq R_2 \cdot t \\
\mathsf{base_wl} &\rightarrow \exists t \in [T,T] : cdeq(Q_1,t) - cdeq(Q_2,t) \geq T/2
\end{aligned}
$$

Thus, queries can ask about many performance metrics, including latency, throughput, fairness, and starvation.

5 The Workload Language

Workloads are also specified as logical formulas over a set of metrics. A workload is a set of constraints, each specifying the traffic pattern for a subset of queues over a period of time. Workloads only constrain *input queues*, i.e., queues that receive packets from "outside" as opposed to another queuing module. This can help users analyze how a contention point will perform under different classes of external traffic.

More formally, as shown in Figure 4, a workload wl is a conjunction of constraints (con). Each con is of the form

Algorithm 1: Workload synthesis search.

Input: User query (qry) from Figure 4.
Output: Workload formula (wl) from Figure 4.

```
1  Procedure Search(qry)
2      if (not feasibleBaseWorkload(qry)) return BadQuery
3      wl = true; G = goodSet(qry); B = badSet(qry)
4      c1 = cost(wl, G, B)
5      while (true) do
6          (found, bad_trace) = verify(wl, qry)
7          if (found) break else B.insert(bad_trace)
8          op = randomOperation()
9          next_wl = wl.apply(op)
10         c2 = cost(next_wl, G, B)
11         if (c1 > c2) then wl = next_wl; c1 = c2
12         else if (e^{-λ·(c2−c1)} > rand()) wl = next_wl; c1 = c2
13     shrink(wl); broaden(wl); return wl
```

$\forall t \in [T_1, T_2] : \mathsf{lhs} \oplus \mathsf{rhs}$, where T_1 and T_2 are integers denoting the interval of time over which con constrains the input traffic, lhs is either a metric for one queue, or the aggregate of a metric over a set of queues, and $\oplus$ is a comparison operator ($>, \geq, <, \leq, =$) that shows how the lhs will be constrained by the rhs, which is time, or a constant.

Workloads can describe sets of traces in a concise and intuitive manner. Consider the single-constraint workload: $\forall t \in [1, 10] : cenq(Q_1, t) \geq t$. Any trace that satisfies this constraint, that is, sends at least t packets into Q_1 by time t and, as a result, does not leave the queue idle, is part of this workload. It does not matter if the traffic enters one packet at a time, or if 10 packets all come in at time step 1, or if 5 packets enter in the beginning, and 5 more at time step 5. That is, workloads can abstract away small details of packet traces as long as a higher-level property, as specified with a metric, is satisfied.

Workload metrics. The search algorithm explores the space of all workloads to find one that satisfies the query. When synthesizing candidate workloads, it decides how many constraints to include in the workload, and what metrics and queues to include in each constraint (§6).

While we leave the metrics that can be used in queries unconstrained, deciding the set of metrics that are used in synthesizing workloads requires careful consideration. We want the set to be small to keep the search space tractable but expressive to enable specifying common workloads in a concise and intuitive manner. We define our workloads over two metrics: (1) cumulative enqueues ($cenq(q,t)$), the total number of packets that enter q by the end of time step t, and (2) arrival inter-packet gap ($aipg(q,t)$), the inter-packet gap between the last two packets that enter q by time t.

While small, this is a quite expressive set. *cenq* constrains the total number of packets entering the queue, independent of the exact time they arrive. So, it can abstract away the timing details of traces when they are not important for the query. *aipg*, on the other hand, constrains the gap between packets and their arrival pace. So, it can capture low-level timing details if necessary in answering the query. Together, they create a good balance in capturing the commonalities of traces that satisfy a query, abstracting away unnecessary details and including necessary ones.

Our experience in the case studies has shown that this set is capable of expressing a variety of workloads. But, we view this as a suitable starting point and not necessarily the final answer. We hope that as using formal methods, and specifically workload synthesis, for performance analysis evolves, the set of metrics will mature as well. In fact, our search algorithm is parametrized over the set of metrics and, if needed, any metric that can be encoded in SMT can be easily added to our search algorithm (see §8 for examples).

6 Synthesizing Answers

The search engine uses a guided randomized search over the space of workloads to find one that satisfies the query. The search algorithm (Algorithm 1) is based on the Metropolis Hastings Markov Chain Monte Carlo (MCMC) sampler, which combines random walks with hill climbing and has been successfully used for synthesizing optimized programs and loop invariants [20, 30, 35]. Starting from an initial workload wl = true (line 3), which imposes no constraints on the input queues, the search algorithm asks the verification engine to verify the workload, i.e., check if all the traces in the workload satisfy the query (line 6). If yes, the search engine returns the workload as the answer to the query (line 7). If not, using the feedback from the verification engine, the search algorithm moves on to synthesize and try another candidate workload until a suitable workload is found (lines 8-12).

6.1 Verifying workloads

Given a workload wl, and a query base_wl $\rightarrow$ qry, the verification engine uses an SMT solver [34] to check if the following formula is satisfiable

$$\mathsf{model} \wedge \mathsf{base_wl} \wedge \mathsf{wl} \wedge \neg \mathsf{qry}$$

where model is the logical encoding of the queueing modules the user is interested in (§3).

If the formula is satisfiable, there is at least one trace that is (1) valid, i.e., satisfies the constraints specified in model such as the maximum number of packets that are allowed to enter a queue between dequeues, (2) satisfies both base_wl (the space of traces in which user is interested (§4)) and wl, and (3) does not satisfy the query. This means that our current candidate workload, wl, is not a suitable answer to the query. So, this trace is returned to the search algorithm to guide the synthesis of the next candidate workload.

If the formula is not satisfiable, either (1) base_wl or wl or their combination with respect to model is *infeasible*, meaning that no valid trace can satisfy their constraints and they actually represent the empty set, or (2) there are no valid traces in base_wl $\wedge$ wl that do not satisfy the query. Only in the second case wl is a valid and non-trivial answer to the query.

To distinguish between these two cases, the search engine asks the verification engine if model $\wedge$ base_wl is satisfiable (line 2). If not, the set of valid traces specified by base_wl is

empty. So, the search engine does not start the search and notifies the user to modify base_wl. Moreover, when verifying a candidate workload wl (verify on line 6), the verification engine checks whether model $\wedge$ base_wl $\wedge$ wl is satisfiable. If not, the fact that wl is infeasible is also returned as feedback to guide the selection of the next candidate.

6.2 Generating the next candidate

If a candidate workload is not the final answer, the search algorithm synthesizes another workload to try. The next candidate workload, next_wl, is a mutation of the previous one, wl. Suppose the previous candidate is $\mathsf{wl} = \wedge_{i=1}^{k} \mathsf{con}_i$. The search algorithm chooses one of the following operations at random (line 8), and applies it to wl (line 9), to obtain next_wl:

- **Add** a new random constraint con, so $\mathsf{next_wl} = \mathsf{con} \wedge \mathsf{wl}$.
- **Remove** a random constraint con_j from wl. That is, $\mathsf{next_wl} = (\wedge_{i=1}^{j-1} \mathsf{con}_i) \wedge (\wedge_{i=j+1}^{k} \mathsf{con}_i)$.
- **Modify** a random constraint con_j. Suppose $\mathsf{con}_j = \forall t \in [T1_j, T2_j] : \mathsf{lhs}_j \oplus_j \mathsf{rhs}_j$. The search algorithm randomly picks whether to change one of $T1_j$, $T2_j$, lhs_j, $\oplus_j$, or rhs_j to obtain con'_j, so that $\mathsf{next_wl} = (\wedge_{i=1}^{j-1} \mathsf{con}_i) \wedge \mathsf{con}'_j \wedge (\wedge_{i=j+1}^{k} \mathsf{con}_i)$.

These operations are motivated by prior work that has empirically shown MCMC to work well with a mixture of major (add and remove) and minor (modify) changes to the current candidate to obtain the next one [30, 35].

Next, the search algorithm uses a cost function (§6.3) to decide whether it is "worth" transitioning to next_wl and try it out. If next_wl has a lower cost compared to wl, next_wl becomes the current candidate (line 11). If next_wl has a higher cost, to avoid getting stuck in local minima, the algorithm may still choose to make the transition with a probability proportional to the difference in next_wl and wl's costs (line 12). The algorithm repeats this process until it finds the next candidate workload.

6.3 The Cost Function

Intuitively, a good cost function should (1) favor workloads when they include a large number of packet traces that satisfy the query, and (2) penalize those that include traces that do not satisfy the query. Inspired by prior work [30], we quantify these criteria into a cost function using *example traces*.

We create two sets of traces before starting the search (line 3): a set of *good* example traces (G), all of which satisfy the query, and a set of *bad* examples (B), none of which satisfy the query. Suppose $\mathsf{match}(\mathsf{wl}, \mathsf{E})$ is the number of traces in E that are also in wl. The cost function is then defined as:

$$\mathsf{cost_E}(\mathsf{wl}, \mathsf{G}, \mathsf{B}) = \mathsf{match}(\mathsf{wl}, \mathsf{B}) - \mathsf{match}(\mathsf{wl}, \mathsf{G}).$$

Recall that if a workload is feasible but does not satisfy the query, the verification engine returns a trace in that workload that does not satisfy the query as feedback. These traces are added to B as the search goes on, further refining the cost function (line 7).

In our experience, $\mathsf{cost_E}$ can effectively guide the search toward workloads that satisfy the query. But there could be multiple such workloads. So, we define another function $\mathsf{cost_S}(\mathsf{wl})$ that favors *concise* workloads, i.e., those with fewer constraints that constrain fewer queues over longer, less fragmented periods of time (details in Appendix B). We define our final cost function as $\mathsf{cost}(\mathsf{wl}) = C_E \cdot \mathsf{cost_E}(\mathsf{wl}) + C_S \cdot \mathsf{cost_S}(\mathsf{wl})$. We set $C_E > C_S$ so that in the beginning, the search algorithm probes the space of workloads for *any* answer that satisfies many good examples and no bad ones. Once the algorithm has reduced $\mathsf{cost_E}(\mathsf{wl})$ and is in the "right" part of the space, $\mathsf{cost_S}(\mathsf{wl})$ helps direct it towards a more concise answer.

Note that even if a workload matches some bad examples, it can still have a low cost and get selected as the next candidate. This is acceptable because the search algorithm may need to go through "obviously" bad workloads to explore different regions of the search space and find the answer. Also, example traces are used only to guide the search; each candidate workload is *verified* in the verification engine to ensure that every trace represented by the workload satisfies the query.

6.4 Generating The Example Sets

Before the search starts, the search engine asks the verification engine to generate traces for G and B. A trace *eg* is a 2D array that concretely specifies how many packets enter each queue at each time step. For example, if $eg[Q_1][5] = 3$, it means that in this trace, Q_1 receives 3 packets at time 5.

The bad examples set (B). The ith bad example is a trace that satisfies the following formula:

$$\mathsf{model} \wedge \mathsf{base_wl} \wedge \neg \mathsf{qry} \wedge \neg(\vee_{j=1}^{i-1} \mathsf{eg_i} \neq \mathsf{eg}_j) \wedge \mathsf{local_mods}_i.$$

Having $\neg(\vee_{j=1}^{i-1} \mathsf{eg_i} \neq \mathsf{eg}_j)$ ensures that the trace is different from the previous $i-1$ traces. $\mathsf{local_mods}_i$ ensures that there is variety across the traces in B so that the search algorithm can prune the search space faster. For that, we pick P random points $(q_1, t_1), \cdots, (q_P, t_P)$ in the $(i-1)$th trace and P random integers $v_1, \cdots, v_P$ such that $\mathsf{eg}_{i-1}[q_j][t_j] \neq v_j$. Then, we define $\mathsf{local_mods}_i = \wedge_{j=0}^{P} \mathsf{eg}_i[q_j][t_j] = v_j$. This way, the ith trace is different from the previous one in at least P points. If no such trace could be found after two tries, we decrement P and retry until a new trace is found.

The good examples set (G). Generating G is more complicated. To see why, consider the diagram in Figure 5 and an arbitrary workload ans that satisfies the query. No matter how we choose B, ans cannot not include any of its traces. So, by minimizing $\mathsf{match}(\mathsf{wl}, \mathsf{B})$ in the cost function, the search algorithm is always moving in the direction of an answer. However, depending on how we pick G, ans may include all (ans_1 in Figure 5), a subset (ans_2), or none (ans_3) of G's traces. There may not even exist a workload like ans_1 that can express all the traces in G without including any bad traces. So, if we do not pick G's traces carefully, by maximizing $\mathsf{match}(\mathsf{wl}, \mathsf{G})$, the algorithm may repeatedly be directed towards a part of the search space where there are no suitable answers.

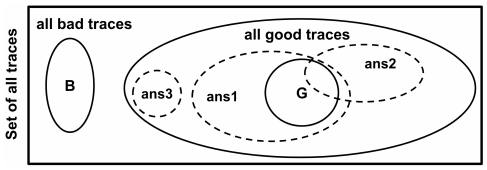


Figure 5: Relationship between answer workloads and G and B.

Intuitively, we want to pick traces for G that are (1) not radically different from each other, so that the majority of them can be represented with a single workload, and (2) not too similar, so that the workload found by the search algorithm is sufficiently general. To do so, the search engine asks the verification engine to create a *base* trace that it will use as the basis of generating the rest. Specifically, it asks for a trace eg_0 that satisfies model $\wedge$ base_wl $\wedge$ qry while using Max-SMT to minimize the following criteria in the specified order:

1. **Total number of queues with traffic**, i.e., $\Sigma_q I_q$, where $I_q = 1$ if $\Sigma_t eg_0[q][t] = 0$, and is zero otherwise. This helps keep the search focused. To see why, suppose we find an eg_0 that satisfies the query and in which only Q_1, Q_2, and Q_3 have traffic. When generating the rest of G, we add constraints that make sure the queues that have no traffic in eg_0 stay empty in the rest of the traces as well. So if there is another set of traces with traffic in, say, Q_2, Q_4, and Q_5, that satisfy the query, they will not be included in G, making sure G's traces are not radically different from each other.
2. **Total number of time steps queues do not receive traffic**, i.e., $\Sigma_{q,t} I_{q,t}$, where $I_{q,t} = 1$ if $eg_0[q][t] = 0$ and is zero otherwise. This means that in the base trace, every queue receives at least one packet every time step as long as it is "harmless", i.e., it does not stop the trace from satisfying the query. This smooth background traffic in the base trace can be randomly changed in the rest of the traces to ensure diversity in the good examples set.
3. **Total traffic in the trace**, i.e., $\Sigma_{q,t} eg_0[q][t]$. Given the above optimization criteria, this ensures that the background traffic is not flooding the contention point with too many packets and only allows more than one packet per time step in the base trace if necessary for satisfying the query.

The rest of G's traces are generated from eg_0. Similar to generating B, the search engine asks for a trace that satisfies the query, is not any of the previous traces, and is different from the previous trace in at least P random places. There are two differences: for the ith trace eg_i, we add extra constraints so that queues that have no traffic in eg_0 (see optimization criteria 1) have no traffic in eg_i either, and we minimize the "distance" between eg_i and eg_0, i.e., minimize $\Sigma_{q,t} d_{q,t}$, where $d_{q,t} = 1$ if $eg_i[q][t] \neq eg_0[q][t]$ and zero otherwise.

6.5 Optimizations

We have employed several optimizations to improve the synthesis process and the final answer. We discuss some of the major ones here and the rest in Appendix C.

Reducing the search space. If a queue q has no traffic in our base trace eg_0 (§6.4), it means that as long as the other "non-idle" queues have traffic, its traffic is either not important or has to be zero for satisfying the query. So, we temporarily add $\forall t \in [1,T] : cenq(q,t) = 0$ to base_wl and only look for workloads that constrain the rest of the queues during search. This reduces the space of workloads the search algorithm needs to explore. These constraints will be removed in post-processing if not necessary.

Post-processing. Once the search engine finds a workload wl that satisfies the query, it creates a new workload ans that includes all the constraints from wl and base_wl. It then performs "workload shrinking" (Algorithm 1, line 13) by removing the constraints in ans one at a time and checking if ans still satisfies the query. This helps remove any constraint that is added to base_wl during example generation or to wl during search but is not necessary for satisfying the query. Next, we try "workload broadening". For a queue Q_i that is not in ans and a constraint con in ans, if con's left hand side is $m(Q_j,t)$, it is changed to $\Sigma_{q\in\{Q_i,Q_j\}} m(q,t)$, and if it is $\Sigma_{q\in S} m(q,t)$, it is changed to $\Sigma_{q\in S\cup\{Q_i\}} m(q,t)$. If the workload still satisfies the query, this helps include even more traces in the workload.

Reducing calls to the verification engine. Each call to the verification engine can be expensive as it checks the satisfiability of non-trivial formulas. So, if the search algorithm selects a candidate workload that matches a trace in B, it moves on to finding the next candidate without consulting with the verification engine, as it already knows that the current candidate includes a trace that does not satisfy the query.

Escaping local minima. To avoid getting stuck in local minima, the search algorithm keeps track of its progress, i.e., the difference between the cost of the previous candidate workload and the next one. If the progress is below a threshold for a number of rounds, it "looks ahead" a couple of hops when generating the next candidate by applying a sequence of moves in §6.2 to generate the next workload. If it still cannot make enough progress in another several rounds, it restarts the search from a workload with no constraints.

7 Case Study: Packet Scheduling

We have prototyped our techniques in a tool we call FPerf in ~10K lines of C++ code, which is publicly available [41]. We use Z3 [34] in the verification engine for checking the satisfiability of SMT formulas. In this section, we describe our experience in using FPerf to analyze packet scheduling algorithms. Our goal is to explore *expressiveness*, i.e., whether our workload language can express a wide range of workloads in answering queries, *interpretability*, i.e., whether the final workloads are concise, intuitive, and interpretable, and *tractability*, i.e., whether workloads are generated in reasonable human timescales (i.e., minutes).

7.1 Stand-alone Schedulers

The priority scheduler is a single queuing module with four input queues and one output queue. Q_i has a higher prior-

ity than Q_j if $i < j$. Every time step, the scheduler picks the highest-priority non-empty input queue, dequeues a packet from it, and puts the packet in the output queue. In a strict priority scheduler, lower priority queues may get starved, which we quantify with the metric $blocked(q,t)$ defined as the number of consecutive time steps that q has packets but is not chosen for transmission. We then ask $\exists t \in [1,T] : blocked(Q_3,t) > 5$.

Starting from $T = 5$, we increment T until the verification engine finds a satisfying trace in the example generation phase at $T = 7$. Then, the search algorithm finds $\forall t \in [1,7] : \Sigma_{q\in\{Q_1,Q_2\}} cenq(q,t) \geq t \wedge \forall t \in [1,7] : cenq(Q_3,t) \geq 1$. That is, to satisfy the query, Q_1 and Q_2, which have higher priorities than Q_3, should collectively receive a consistent flow of traffic (constraint 1), and Q_3 should at least have one packet to be considered blocked (constraint 2). While the answer is not surprising, it demonstrates FPerf's ability to abstract away the details of which higher priority queue receives packets at exactly what time step, only capturing the necessary details.

The round-robin scheduler is a single queuing module with five input queues and one output queue. The input queues are serviced in a round-robin fashion (independent of packet size, see §9). If every queue receives steady traffic over a time period T, each should be selected for dequeue at least $T/5$ times. So, when we ask if Q_3 can dequeue more packets than Q_2 with query $\forall t \in [10,10] : cdeq(Q_3,t) - cdeq(Q_2,t) \geq 3$ and base workload $\wedge_{i=1}^{5} \forall t \in [1,10] : cenq(Q_i,t) \geq t$, the verification engine cannot find any traces that satisfy the query.

Now suppose we relax base_wl to restrict the average rate of traffic every 5 time steps as opposed to every time step, i.e., have the queues receive at least 4 packets every 5 time steps: $\text{base_wl} = \left(\wedge_{i=1}^{5} \forall t \in [5,5] : cenq(Q_i,t) \geq 4\right) \wedge \left(\wedge_{i=1}^{5} \forall t \in [10,10] : cenq(Q_i,t) \geq 8\right)$. FPerf finds $\forall t \in [1,4] : \Sigma_{q\in\{Q_1,Q_2,Q_4,Q_5\}} cenq(q,t) \leq 0 \wedge \forall t \in [1,4] : cenq(Q_3,t) \geq t \wedge$ base_wl, which describes a workload where all queues except Q_3 receive no packets in the first four time steps and receive a burst of at least 4 packets at time 5, while Q_3 continuously receives traffic. So, while the average input rate of all queues is the same, other queues temporarily fall behind Q_3 in terms of dequeues due to the burstiness of their traffic.

FQ-Codel. This case study analyzes the buggy scheduler inspired from the FQ-Codel qdisc [32]. The scheduler's logic, the query, and the workload are described in §2 as our motivating example. Here, we only report that the same workload was overwhelmingly returned as the answer across all runs.

7.2 Composing Host and NIC Schedulers

Modern NICs expose multiple transmit queues to the host, so that CPU cores can concurrently send traffic to the NIC, each through a dedicated transmit queue [42–44]. This provides significant performance benefits but makes it difficult to enforce policies about how applications on the same host should share network resources. Prior work [44] demonstrates this using an example, which we analyze in this case study.

Suppose two tenants reside on a server with multiple CPU cores. Tenant 1 runs spark [45], and tenant 2 runs both spark and memcached [46]. All applications are multi-threaded and can use all the cores. We want to ensure that the two tenants fairly share the network bandwidth, and tenant 2's memcached traffic is prioritized over its own spark traffic. One option, described in [44], is to use software packet schedulers (e.g., Linux qdiscs) to enforce fair sharing between the tenants on the host, and a priority scheduler on the NIC to enforce prioritization of memcached traffic. Note that to avoid overhead, software schedulers enforce policies per core not across cores.

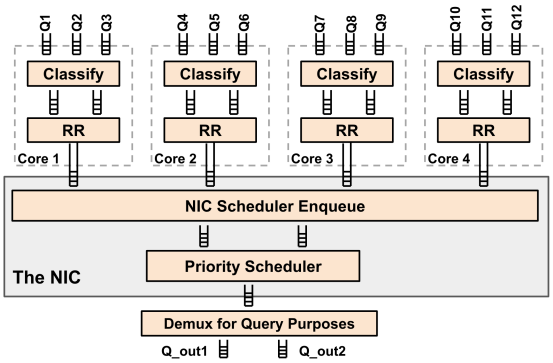


Figure 6: Setup for the case study in §7.2.

Figure 6 shows how we model this in FPerf for 4 CPU cores. There is one input queue for traffic from each application on each core. That is, $Q_{3(i-1)+1}$, where i is the core number, are for tenant 1's spark, $Q_{3(i-1)+2}$ for tenant 2's spark, and Q_{3i} for tenant 2's memcached traffic. For each core, a queuing module first classifies traffic from that core's input queues into two output queues, one for each tenant. Then, a round-robin scheduler shares bandwidth between the two tenants into the NIC's transmit queue. On the NIC, a module classifies traffic from the cores into two output queues, one for spark and one for memcached traffic. Finally, a priority scheduler that prioritizes memcached traffic decides what packet to send out of the NIC. We add a "dummy" module that takes the output from the NIC and "demultiplexes" it into Q_{out1} queue for tenant 1 and Q_{out2} for tenant 2 to use in our queries.

Since the final scheduler always prioritizes memcached traffic over spark, it is easy to see how the second half of the policy is always enforced. So, we ask whether tenant 1 and tenant 2 will get equal access to the NIC output link:

$$\text{base_wl} \rightarrow \forall t \in [10,10] : \; cdeq(Q_{out2},t) - cdeq(Q_{out1},t) \geq 3$$
$$\text{base_wl} = (\forall t \in [1,10] : \; \Sigma_{q\in S_{tenant1}} cenq(q,t) \geq t) \wedge$$
$$(\forall t \in [1,10] : \Sigma_{q\in S_{tenant2}} cenq(q,t) \geq t)$$

$S_{tenant1}$ are $Q_{3(i-1)+1}$ $(1 \leq i \leq 4)$, which carry tenant 1's spark traffic, and the rest of the queues are in $S_{tenant2}$. That is, in base_wl, we ensure that both tenants receive a steady stream of packets. For this query, FPerf finds the workload

$$\forall t \in [1,10] : \; cenq(Q_4,t) \geq t \wedge \forall t \in [1,10] : \; cenq(Q_9,t) \geq t$$
$$\wedge \; (\wedge_{i\in S_{rest}} \forall t \in [1,10] : \; cenq(Q_i,t) \leq 0),$$

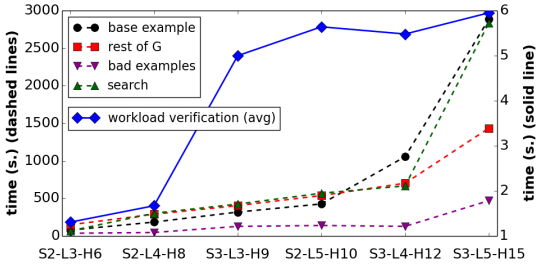
where S_{rest} is $\{1,\cdots,3,5,\cdots,8,10,\cdots 12\}$. That is, if tenant 2's memcached runs on core 3 and tenant 1's spark on core 2, they

Phases		Prio	RR	FQ-C	Comp	LS-T	LS-L
Example Generation	Generating base trace (s)	0.3	1.0	3.1	70.3	59.5	71.5
	Generating good set (G) (s)	6.8	309	346	321	310	632
	Generating bad set (B) (s)	1.2	3.0	7.1	179	74.1	69.6
Verification Engine	Verifying workload (avg) (s)	0.02	0.02	0.11	0.65	0.73	0.94
	Verifying workload (max) (s)	0.04	0.13	0.86	7.60	3.26	2.18
	Verifying query (avg) (s)	0.03	0.04	0.10	0.81	1.66	1.46
	Verifying query (max) (s)	0.06	0.18	0.73	9.90	4.09	2.26
Search (see §6.5)	# Rounds	65	268	769	361	949	117
	# Rounds w.o/verification	39	107	537	131	836	65
	# Rounds w/"look-ahead"	1	11	44	20	162	1
	Total search time (s)	2.4	59	223	461	420	121
Post Processing	Workload shrinking (s)	0.2	0.0	0.9	107	25.9	16.4
	Workload broadening (s)	1.0	0.0	0.1	45	0.0	2.6
Total Time (min.)		0.2	6.2	9.6	18.5	13.8	14.0

(a) Statistics from running FPerf 10 times for each case study. **Parameters:** Queue parameters are $S = 10$ and $K = 4$ packets. For example sets, $|G| = |B| = 50$, and $P = min(10, \frac{trace_size}{5})$. During search, the maximum number of constraints in the workload is set to twice the number of input queues, threshold for slow progress is $0.03 \cdot max(\mathsf{cost_E})$, $\frac{\mathsf{C_E}}{\mathsf{C_S}} = 10$, and number of rounds of slow progress tolerated before look-ahead and restart is 10 and 20 respectively.

	Prio	RR	FQ-C	Comp	LS-T	LS-L
Queuing Modules	1	1	1	11	23	23
Queues	5	6	7	29	66	66
Boolean Vars (×1000)	0.8	0.2	2.6	10.6	21.3	21.3
Integer Vars (×1000)	0.6	1.1	1.9	7.3	23.2	23.2
Constraints (×1000)	7.2	13	2.2	93.8	45.8	45.5
Max timesteps	7	10	14	10	10	10

(b) Case study statistics



(c) Workload synthesis statistics for the latency query on leaf-spine networks of increasing size. S_i-L_j-H_k has i spines, j leaves, and i hosts per leaf to avoid oversubscription ($k = i \times j$). Parameters are same as figure 7a except $|\mathsf{G}| = |\mathsf{B}| = 25$ (see §9).

Figure 7: Case study statistics and results (§7 and §8).

will only compete in the priority scheduler in the NIC, where memcached traffic is prioritized over spark traffic, and tenant 2 is favored to access the link. The same phenomena can happen as long as tenant 2's memcached traffic and tenant1's spark traffic come from two different cores.

Next, we modify the base workload to see if the problem still exists if there is no memcached traffic. We add $\bigwedge_{i\in\{3,6,9\}} cenq(Q_i,t) = 0$ to the base workload and repeat the query. This time, we get the following workload as answer:

$$\begin{aligned}&\forall t \in [1,10] : cenq(Q_8,t) \geq t \wedge \forall t \in [1,10] :\ cenq(Q_{11},t) \geq t\\ &\wedge \forall t \in [1,10] :\ \Sigma_{q\in\{Q_2,Q_5\}} cenq(q,t) \geq t\\ &\wedge \forall t \in [1,10] :\ cenq(Q_{10},t) \geq t\\ &\wedge (\wedge_{i\in rest} \forall t \in [1,10] :\ cenq(Q_i,t) \leq 0).\end{aligned}$$

where $rest = \{1,3,4,6,7,9,12\}$. Here tenant 2's spark runs on all cores and tenant 1's spark only on core 4. The first three constraints ensure that there are three different concurrent streams of traffic from tenant 2's spark. Since there is only one stream of spark traffic for tenant 1, that is, because the spark flows are not uniformly spread across cores, tenant 1 gets access to the link less frequently than tenant 2. Both workloads are consistent with the empirical results in [44].

7.3 Tractability

Tables 7a and 7b summarize statistic about different phases involved in workload synthesis for the packet scheduling case studies across 10 runs. We use a server with 4-socket NUMA-enabled Intel Xeon Gold 6128 3.4GHz CPU with 6 cores per socket. For Comp, we only include the results for the second query as, compared to the first query, its analysis is more involved in all synthesis phases.

Generating G is expensive. This is not surprising: when generating trace eg_i, we constrain it to have different randomly chosen values from eg_{i-1} in $p = P$ random places. If no trace is found in two tries, we decrement p and try again. Each time, we also ask the verification engine to minimize the distance between eg_i and the base trace (§6.4). The more such calls, either due to the sheer size of G, or because we have to retry a lot for each trace, the longer generating G will take.

Compared to Prio and Comp, it takes more tries to find a trace for RR and FQ-Codel (average of 16 and 6 tries, respectively). This is because the answers to their queries are more sensitive to the timing of packets in the trace: RR needs a specific kind of burstiness and FQ-CoDel needs a specific rate. So, making p random modifications to eg_{i-1} for larger values of p makes it improbable to satisfy the query for eg_i, increasing the number of retries. By default, $P = min(10, \frac{trace_size}{5})$, which is 10 for RR and FQ-CoDel, translating to a maximum of 20 tries per example trace. Instead, we can potentially set P to the moving average of the p that has worked for the previous traces and reduce the number of retries, and consequently, the total time for generating G.

Another option is to reduce $|\mathsf{G}|$, potentially increasing search time as shown in Figure 10 (Appendix). For all but FQ-Codel, the decrease in example generation time outweighs the increase in search time, and workload synthesis is fastest for $|\mathsf{G}| = 25$. For FQ-CoDel the sweetspot is $|\mathsf{G}| = 50$. One could execute workload synthesis in parallel for different values of $|\mathsf{G}|$ and use the results from whichever finishes first. Moreover, once we have the base trace eg_0, we can potentially parallelize the generation of G into x threads, each generating $\frac{|\mathsf{G}|}{x}$ traces. Our randomized changes from one trace to the next reduces the risk of getting duplicate traces across threads, and even if there are a few, it will not affect the correctness of the search.

The verification engine is efficient. The verification engine can verify a workload in $< 1sec$ on average. The worst case

is $\sim 10sec.$ for Comp, and $< 1sec.$ in other case studies. The efficiency of the verification engine is thanks to the advances in SMT solvers [34], our efficient encoding of queues (Appendix), and our choice to use immediate composition.

In *immediate* composition, packets leaving output queues at time t are visible in the input queues of the next module at time t, as opposed to $t+1$ in sequential composition (§3). So, if we have a chain of L queuing modules, a packet arriving at time t is processed by all the queuing modules one-by-one but in the same time step and appears in the final output queue at $t+1$, as opposed to $t+1+L$. In Comp, all the paths from input to output queues through the queuing modules have the same length. So, if we use immediate composition across *all* modules, it will not change the relative ordering of packets across queuing modules and allows us to analyze the model in 10 as opposed to 15 time steps and get the same output.

Optimizations (§6.5) are effective. For instance, when a candidate workload matches one of the bad example traces, we can move on to the next candidate without calling the verification engine. Using this optimization, we avoided calling the verification engine in ~ 40 to 70% of the rounds, which translates to saving ~ 113 and 192 seconds in FQ-CoDel and Comp that are more complex. Moreover, The "look-ahead" strategy is used in 5% of the rounds, helping the search algorithm to avoid local minima and plateaus.

8 Case Study: A Small Leaf-Spine Network

In this section, we use FPerf to analyze a small leaf-spine network (Figure 8). Our goal is to demonstrate the expressiveness of our model and the generality of our techniques.

Modeling switches. We model our switches after input-queued switches with virtual output queues (VOQs). Suppose the switch has P ports. Each input port i has P VOQs, where the jth VOQ stores packets that are destined for output port j. The switch crossbar decides which input ports can simultaneously send packets to which output ports without interfering with each other, and delivers packets from the corresponding VOQs at those input ports to their destination output ports.

Figure 9 shows how we model this in FPerf. A switch with P ports is a composition of $P+1$ queuing modules. There are P forwarding modules, one for each input port. The ith forwarding module takes a packet from input port i, decides the destination port j it should be forwarded to, and places the packet in the jth VOQ for port i. The crossbar queuing module models the switch crossbar. In each time step, using the iSlip algorithm [47], it matches input ports and output ports such that each input port is matched with at most one output port and vice versa. If an input port i is matched with an output port j, the crossbar moves a packet from the jth VOQ at port i to the output queue for port j. iSlip and its variants are widely used in switching fabrics as they provide high throughput in the crossbar and fairness across inputs.

Packet metadata and workload metrics. The per-packet metadata variables include dst, a variable representing the final destination of the packet, and $ecmp$, a variable with values in $[0,S]$ where S is the number of spines switches, representing the result of the ECMP hash of the packet's flow id modulo the number of spines. We extend example traces and our algorithms for generating them (§6.4) to include packet metadata (details are in Appendix D.1). Moreover, we add two corresponding metrics, $dst(q,t)$ and $ecmp(q,t)$, to our workload language to track the values of these variables for packets entering q at time t. These metrics, together with $cenq(q,t)$ and $aipg(q,t)$, can be used in the synthesized workloads to describe a range of traffic patterns and flows.

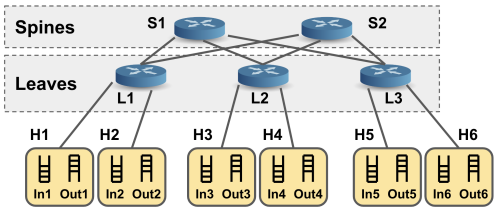


Figure 8: A small leaf-spine network as case study (§8).

User Interface. To create a model of a network of switches, the users need to specify the topology (switches and links) and the forwarding rules (mapping per-packet metadata to an output port) for each switch. For a leaf-spine network, FPerf provides a special interface that only requires specifying the number of leaves and spines. For the base workload, the users can provide a list of constraints using an interface that abstracts away the logical operators and expressions in Figure 4. Each constraint is either (1) [t1, t2] q.m $\oplus$ c, constraining metric m for queue q against a constant ($\oplus$ is any comparison operator), (2) [t1, t2] (q1.m + ... + qn.m) $\oplus$ c, (3) [t1, t2] q1.m $\oplus$ q2.m, or (4) [t1, t2] (q1.m + ... + qn.m)/t $\oplus$ c, constraining a metric's value for one or more queues over time. Queues are identified by switch id and port number.

For queries, the users provide a list of questions with a similar interface, asking if the value of a metric over a time period for one or more queues can go above or below a threshold. There are two special shorthands for common queries: q.avg_rate is the average input rate for queue q, which translates to q.cenq/t, and lat(s1, ..., sn) is the latency through the specified sequence of switches, which translates to sum of queue sizes (q1.qsize + ... + qn.qsize) along the path. Appendix D.2 includes more details on the translation between this interface and the syntax in Figure 4.

The throughput query (LS-T). Since there is no oversubscription in our leaf-spine network, we first ask whether the throughput between hosts 1 and 6 can drop below line-rate:

$$\text{base_wl} \rightarrow \forall t \in [10,10]: \; cenq(Out_6,t) < 5$$
$$\text{base_wl} = (\forall t \in [1,10]: \; dst(In_1,t) = 6) \wedge (\wedge_{i,j\in[1,6],i\neq j} \forall t \in [1,10]: dst(In_i,t) \neq dst(In_j,t))$$

In_i and Out_i are the queues host i uses to send traffic into and receive traffic from the network, respectively. The query asks whether the total number of packets received by host i at time 10 is less than half of what it should have received at line rate. The base workload ensures that host 1 sends a steady stream

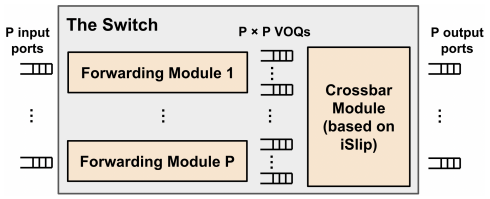


Figure 9: Modeling an input-queued switch with VOQs in FPerf.

of traffic to host 6 (at least 1 packet per time step), and no two hosts send traffic to the same destination so there is no traffic concentration at the hosts. For this query, FPerf finds the following workload:

$$\forall t \in [1,10]:\ dst(In_3,t) \geq 5 \wedge \forall t \in [1,7]:\ ecmp(In_1,t) = 1$$
$$\wedge \forall t \in [1,7]:\ ecmp(In_3,t) = 1 \wedge \forall t \in [1,10]:\ cenq(In_4,t) \leq 0.$$

That is, if there is another flow from host 3 to the third pod (constraint 1) with the same *ecmp* as the flow from host 1 to 6 (constraints 2 and 3), the two flows have to compete for bandwidth on the link from S_1 to L_3 (Figure 8) and host 6 will not receive traffic from host 1 at line rate. The last constraint ensures that the flow from host 3 has all the bandwidth from the second pod to itself to compete with the flow from host 1.

The latency query (LS-L). In the absence of queuing delay, it takes packets three time steps to go from a host in one pod to a host in a different pod. We want to know if it can take much longer, say, 13 time steps, for packets to go from host 1 to 6. To do so, we ask if it is possible to have a queue build up of at least 10 packets along the path of the flow:

$$\mathsf{base_wl} \rightarrow \forall t \in [10,10]:\ \Sigma_{q \in path}\, qsize(q,t) \geq 10$$
$$\mathsf{base_wl} = (\forall t \in [1,10]:\ dst(In_1,t) = 6) \wedge$$
$$(\forall t \in [1,10]:\ ecmp(In_1,t) = 1) \wedge (\forall t \in [1,10]:\ cenq(In_1,t) \leq t)$$

Here, the base workload ensures there is a flow from host 1 to 6 (constraints 1 and 2), and that the flow sends at most at line rate (constraint 3), so that there is no artificial queue build up from the flow's own packets. *path* is the set of queues the flow visits as it traverses L_1, S_1 (since in base_wl, $ecmp(In_t,t) = 1$), and L_3 (see Figure 8).

For this query, FPerf finds the following workload:

$$\forall t \in [1,8]:\ dst(In_3,t) = 6 \wedge \forall t \in [1,8]:\ ecmp(In_3,t) = 1$$
$$\wedge \forall t \in [1,10]:\ ecmp(In_5,t) = 6 \wedge \forall t \in [1,10]:\ cenq(In_4,t) \leq 0.$$

That is, if hosts 3 and 5 (from pods 2 and 3) send traffic to host 6 at the same time (constraints 1 to 3), there will be a queue build up of at least 10 packets along the path of the flow and its packets will experience high latency. Similar to the throughput query, the last constraint ensures that the flow from host 3 has all the bandwidth from the second pod to itself to contribute to quickly building up the queues.

Tractability. Tables 7b and 7a summarize statistics about different phases of workload synthesis for the queries about the leaf-spine network. The results are consistent with our observations in §7.3. Generating the good example set is expensive but there are opportunities for parallelization and further optimizations. The verification engine is efficient, verifying workloads in $< 1.7sec.$ on average and $\sim 4sec.$ in the worst case. Beyond what is described in §7.3, we employ other optimizations that contribute to this efficiency. Specifically, we use the forwarding rules in the leaf-spine topology to (1) remove certain VOQs from the switch crossbar if their corresponding input and output ports are not expected to communicate (e.g., a packet entering a leaf from a spine is expected to go to one of the output ports connected to the hosts and not other spines), and (2) constrain the values of the per-packet metadata to reduce the search space (e.g., all packets going into S_1 have *ecmp* set to 1). Finally, our search optimizations remain effective, avoiding calls to the verification engine in ~ 55 to 89% of the rounds, and the example traces effectively guide the search towards workloads that satisfy the query.

Takeaways. Queuing modules and their composition are expressive enough to model a variety of network components, from packet schedulers and classifiers to a network of switches. We did not need to make any changes to how we model contention points (§3) to model the leaf-spine network. Similarly, our workload synthesis techniques generalize beyond packet scheduling. Our example generation strategy and workload metrics need only minor changes to include the packet metadata needed for forwarding over the network, and our search algorithm can find workloads for the queries as effectively without any modifications.

9 Discussion and Future Directions

Scaling to large networks. Figure 7c shows how example generation, search, and workload verification times increase for the latency query on leaf- spine networks of increasing size. As the network size increases, the number of variables and constraints go from $45k$ and $46k$ to $182k$ and $181k$, and automated theorem provers (e.g., Max-SMT and SMT solvers) which we use in example generation and workload verification, can take exponentially longer as the problem size increases. For our largest evaluated network (not shown in the figure) with 4 spines, 4 leaves, and 16 servers (56 modules and 288 queues), it takes 224 minutes to find an answer.

There is room for more optimizations, some of which we have implemented for this experiment and discuss in Appendix D.3. But, as with any other approach that relies on similar formal methods tools, there is a limit to how many variables and constraints we can jointly reason about within a reasonable amount of time. As such, similar to data and control-plane verification tools that have matured over a decade to scale to large-scale networks, much work needs to be done to improve the scalability of formal methods tools for network performance analysis.

For example, we may need to develop new techniques, e.g., domain-specific algorithms for reasoning about network properties to replace SAT/SMT solvers [12, 48] or decompose global properties into local properties for modular analysis [15, 31]. Specifically, given that performance queries and properties such as latency lend themselves well to decomposi-

tion over regions and paths, we believe modular analysis to be key in scaling automated formal reasoning about performance and an important direction for future work.

Bounded Time. We model queuing modules for a bounded number of time steps, starting with empty queues and every module in its initial state. Nevertheless, the workloads in our case studies often include "repeatable" patterns. For instance, in FQ-CoDel, Q_5 can continue causing fairness problems after 14 timesteps if it keeps sending at the rate specified in the workload. Prior work explores how to find periodic adversarial input patterns for P4 programs [49], and it would be interesting to explore similar ideas in our context. Moreover, to cover different portions of the time horizon, we can explore ways to compute a subset set of reachable states in queuing modules and start from those as opposed to the initial state. Finally, exploring verification techniques for reasoning about unbounded time [50] is an interesting avenue for future work.

Packets vs. bits. In our prototype, metrics, and therefore, queries and workloads, are defined in terms of packets rather than bits. We plan to extend FPerf to include packet sizes as extra variables, so they can be used in defining metrics and potentially reveal even more interesting workloads.

Generating traces from workloads. We can use the verification engine to generate example traces from the final workloads. Transforming these traces or the traces in G to concrete packets that can be injected into real-world networks (e.g., see Adapters in [37]) is an interesting future direction.

10 Related Work

Network Calculus. Network Calculus [6, 51] offers a uniform mathematical framework for analyzing performance guarantees. To use network calculus, one needs to model the input workload as an "arrival curve", which bounds the arrival pattern of bits into a network component, and the network component as a "service curve", which bounds the number of serviced bits. Using these curves and $(min, +)$ algebra, one can then derive bounds on performance metrics such as throughput, latency, jitter, and loss [4, 52–54]. However, these curves need to be reasonably concise and provide tight bounds on the behavior of the network component and the input traffic pattern for the final bounds to be tight and useful. Deriving arrival and service curves is challenging, particularly for today's complex network functionality and traffic patterns [10].

Quantitative Reasoning. There is a line of work for reasoning about quantitative network properties: Some extend dataplane verifiers to reason about quantities such as link loads and hop counts [55–57]. Others reason about probabilistic aspects of networks (e.g., weighted ECMP) and answer probabilistic questions (e.g., the probability that packets reach a destination) [58,59]. Our work is similar in that performance properties are quantitative. However, these tools focus on *verification*, whereas, we propose to use *synthesis* to automatically generate not just one counter-example, but a workload that violates user-defined performance-related properties. Moreover, these tools abstract away many low level network details as nondeterministic, which we are able to model more precisely in SMT in our queueing modules.

Automated protocol analysis. Recent work uses techniques such as bounded model checking and guided search to check if congestion control algorithms can be driven into undesirable states or underutilize the network [60–62]. Khan et al. propose to train Markov models that capture the temporal behavior and throughput and delay distributions of delay-based congestion control protocols [63]. Gilad et al. use reinforcement learning (RL) to train agents that generate adversarial traces for protocols and use it to demonstrate sub-optimal performance in some RL-driven protocols [64]. We propose generating workloads describing sets of traces in response to user-defined queries about performance problems.

Synthesis. Syntax-Guided Synthesis (SyGuS) is a general approach to program synthesis that uses a verifier together with a candidate program grammar. There are various ways to do SyGuS; there are enumerative [65, 66], stochastic [20, 30, 35], and logical approaches [67]. Our work is based on stochastic search. Moreover, recent work explores using synthesis in networking to generate packet processing code [19, 20], network configuration [21–23], configuration updates [24], or control-plane repairs [25, 26]. We use synthesis to generate workloads to reason about network performance.

11 Conclusion

Over the past decade, a large body of academic and industry work has demonstrated the feasibility and benefits of using formal methods to reason about the functional correctness of networks. Inspired by their success, we set out to bring the same benefits to analyzing network performance.

Along the way, we have developed efficient encodings of packet-level interactions that affect network performance. We have also found that when it comes to performance analysis, returning isolated packet traces that violate performance properties is not always useful. Instead, we argue that a more useful output is a workload that can concisely describe the commonality of a set of traces that can experience performance problems. We have shown how to apply existing synthesis techniques to generating such workloads and demonstrated the tractability of our approach using case studies.

This is only the start; as with other applications of formal methods to systems and networking, much work needs to be done to make such formal performance analysis approaches suitable for analyzing real-world networks, some of which we have outlined in this paper as future research directions.

Acknowledgments

We thank Laurent Vanbever, our shepherd, the anonymous reviewers, Jennifer Rexford, Shir Landau-Feibish, and Nate Foster for their helpful feedback. This work was supported in part by NSF grants CNS-2047283 and CNS-1704742, a Google faculty research award, and a Sloan fellowship.

References

[1] NS3 Network Simulator. https://www.nsnam.org/. Accessed: 09-2022.

[2] Mininet. http://mininet.org/. Accessed: 09-2022.

[3] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully emulating large production networks. In *ACM SOSP*, 2017.

[4] Victor Firoiu, J-Y Le Boudec, Don Towsley, and Zhi-Li Zhang. Theories and models for internet quality of service. *Proceedings of the IEEE*, 2002.

[5] Rayadurgam Srikant. *The mathematics of Internet congestion control*. Springer, 2004.

[6] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: A theory of deterministic queuing systems for the internet*. Springer, 2001.

[7] Steven H Low. A duality model of TCP and queue management algorithms. *IEEE/ACM Transactions On Networking*, 2003.

[8] Jiayue He, Mung Chiang, and Jennifer Rexford. TCP/IP interaction based on congestion price: Stability and optimality. In *2006 IEEE International Conference on Communications*, 2006.

[9] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: Stability, convergence, and fairness. *ACM SIGMETRICS*, 2011.

[10] Florin Ciucu and Jens Schmitt. Perspectives on network calculus: No free lunch, but still good value. In *ACM SIGCOMM*, 2012.

[11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

[12] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[13] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[14] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-Net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[15] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, et al. Validating datacenters at scale. In *ACM SIGCOMM*. 2019.

[16] Announcing Network Intelligence Center – towards proactive network operations. https://cloud.google.com/blog/products/networking/announcing-network-intelligence-center. Accessed: 09-2022.

[17] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *ACM SIGCOMM*, 2020.

[18] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. GRoot: Proactive verification of DNS configurations. In *ACM SIGCOMM*, 2020.

[19] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, 2020.

[20] Qiongwen Xu, Michael D Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *ACM SIGCOMM*, 2021.

[21] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *ACM SIGCOMM*, 2016.

[22] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *ACM SIGPLAN PLDI*, 2017.

[23] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *USENIX NSDI*, 2018.

[24] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *ACM SIGCOMM*, 2021.

[25] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *ACM SOSP*, 2017.

[26] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: incrementally synthesizing policy-compliant and manageable configurations. In *ACM CoNEXT*, 2020.

[27] VMware to Advance Network Monitoring with Acquisition of Veriflow. https://blogs.vmware.com/management/2019/08/vmware-to-advance-network-monitoring-with-acquisition-of-veriflow.html. Accessed: 09-2022.

[28] Intentionet. https://www.intentionet.com/. Accessed: 09-2022.

[29] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *FMCAD*, 2013.

[30] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 2016.

[31] Ryan Beckett and Ratul Mahajan. Capturing the state of research on network verification. https://netverify.fun/2-current-state-of-research/. Accessed: 09-2022.

[32] Toke Høeiland-Jørgensen, Paul McKenny, Dave Taht, Jim Gettys, and Eric Dumazet. The Flow Queue CoDel packet scheduler and active queue management algorithm. RFC 8290, 2018.

[33] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM*, 1995.

[34] Z3. https://github.com/Z3Prover/z3. Accessed: 09-2022.

[35] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM ASPLOS*, 2013.

[36] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2Text: Query-Guided summarization of network forwarding behaviors. In *USENIX NSDI*, 2018.

[37] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *ACM SIGCOMM*, 2021.

[38] Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In *ACM OOPSLA*, 2014.

[39] Emina Torlak and Rastislav Bodik. A Lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN PLDI*, 2014.

[40] CBMC. https://www.cprover.org/cbmc/. Accessed: 09-2022.

[41] FPerf Github Repository. https://github.com/minmit/fperf. Accessed: 09-2022.

[42] Nvidia ConnectX SmartNICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/. Accessed: 09-2022.

[43] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *USENIX NSDI*, 2014.

[44] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *USENIX NSDI*, 2019.

[45] Apache Spark. https://spark.apache.org/. Accessed: 09-2022.

[46] Memcached: a Distributed Memory Object Caching System. http://www.memcached.org/. Accessed: 09-2022.

[47] Nick McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 1999.

[48] Todd Millstein. Toward modular network verification. https://netverify.fun/toward-modular-network-verification/. Accessed: 09-2022.

[49] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *ACM ASPLOS*, 2021.

[50] Anthony Lin. *Model Checking Infinite-State Systems: Generic and Specific Approaches*. PhD thesis, 2010.

[51] Rene L Cruz. A calculus for network delay. parts I and II. *IEEE Transactions on Information Theory*, 1991.

[52] Jorg Liebeherr, Yashar Ghiassi-Farrokhfal, and Almut Burchard. On the impact of link scheduling on end-to-end delays in large networks. *IEEE Journal on Selected Areas in Communications*, 2011.

[53] Jörg Liebeherr, Almut Burchard, and Florin Ciucu. Delay bounds in communication networks with heavy-tailed and self-similar traffic. *IEEE Transactions on Information Theory*, 2012.

[54] C-S Chang. Stability, queue length and delay. II. Stochastic queueing networks. In *IEEE Conference on Decision and Control*, 1992.

[55] Garvit Juniwal, Nikolaj Bjorner, Ratul Mahajan, Sanjit Seshia, and George Varghese. Quantitative network analysis. *Technical report*, 2016.

[56] Ying Zhang, Wenfei Wu, Sujata Banerjee, Joon-Myung Kang, and Mario A Sanchez. SLA-verifier: Stateful and quantitative verification for service chaining. In *IEEE INFOCOM*, 2017.

[57] Kim G Larsen, Stefan Schmid, and Bingtian Xue. WNetKAT: A weighted SDN programming and verification language. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2017.

[58] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic NetKAT. In *European Symposium on Programming*, 2016.

[59] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. Bayonet: Probabilistic inference for networks. In *ACM SIGPLAN PLDI*, 2018.

[60] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *ACM SIGCOMM*, 2021.

[61] Wei Sun, Lisong Xu, Sebastian Elbaum, and Di Zhao. Model-Agnostic and efficient exploration of numerical state space of real-world TCP congestion control implementations. In *USENIX NSDI*, 2019.

[62] Samuel Jero, Md Endadul Hoque, David R Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *NDSS*, 2018.

[63] Muhammad Khan, Yasir Zaki, Shiva Iyer, Talal Ahamd, Thomas Pötsch, Jay Chen, Anirudh Sivaraman, and Lakshmi Subramanian. The case for model-driven interpretability of delay-based congestion control protocols. *ACM SIGCOMM Computer Communication Review*, 2021.

[64] Tomer Gilad, Nathan H Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying network protocols with adversarial examples. In *ACM HotNets*, 2019.

[65] Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed program synthesis. In *ACM SIGPLAN PLDI*, 2015.

[66] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *ACM SIGPLAN-SIGACT POPL*, 2016.

[67] Sumit Gulwani and Ramarathnam Venkatesan. Component-Based synthesis applied to bitvector circuits. *Technical report*, 2009.

A Efficient Encoding of FIFOs in SMT

A queue is specified with two parameters, its size S and the maximum number of enqueues K allowed at every time step. We define the following variables for each queue for every time step t:

1. $enqs[t][1:K]$ consists of K tuples, where each tuple represents a packet. This captures the packets that are sent to the queue at time t. These packets will enter that queue at time $t+1$,
2. $elems[t][1:S]$, consists of S tuples, where each tuple represents a packet. This captures the packets that are inside the queue at time t, and
3. an integer variable $deq_cnt[t]$ that captures how many packets will be dequeued from this queue at time t.

We also define a set of helper boolean variables $val_elem[t][i]$. $val_elem[t][i]$ is true if there is a packet in $elems[t][1:S]$ and is false if $elems[t][i]$ is empty. $val_enq[t][1:K]$ is defined similarly.

Our queues can have up to K enqueues and up to S dequeues in every time step. To model that, we take care of the dequeues first. We define an extra set of helper variables $tmp_val[t][1:S]$ to denote which indexes in the queue would still have packets and which ones would become empty at $t+1$ if we were to only do $deq_cnt(t)$ number of dequeues and not any enqueues. Specifically, $tmp_val[t][i]$ is true if the ith element of the queue will still contain a packet after the dequeues in that time step assuming no enqueues happen.

To capture that, for every $1 \le i \le S$ and $1 \le d \le S$, if $i+d \le S$, we add

$$\big(deq_cnt[t] = d\big) \rightarrow \big(tmp_valid[t][i] = val_elem[t][i+d]\big)$$

Otherwise, we add

$$\big(deq_cnt[t] = d\big) \rightarrow \big(\neg tmp_val[t][i]\big).$$

We also have some standard constraints to shift the packets in $elem$ forward depending on the $deq_cnt(t)$.

The next set of constraints handle the enqueues in a "sliding window" fashion. That is, starting from the head of the queue, we consider all possible $K+1$ consecutive positions in the queue to find a window where the first element has a packet and the next K are empty. This will be the tail of the queue and where we will be enqueuing the new packets.

Specifically, for every $1 \le i \le S-K$ and $1 \le j \le K$, we add the following constraint:

$$tmp_val[t][i] \wedge \neg tmp_val[t][i+1] \rightarrow elem[t][i+j] = enqs[t-1][j]$$

We add extra constraints for the start and end of the queue and when there is not enough space for K packets.

Finally, we add constraints to make sure there is no "hole" in the queue. That is, suppose the queue has a packet at index i and no packets at index $i+1$. Then, there is a packet at any index $j \le i$ queue. Moreover, at any index $j > i$, the queue is empty. Specifically, for every $1 \le i < S$, we add:

$$val_elem[t][i] \vee \neg val_elem[t][t]$$

B Defintion of cost_S

Consider a workload $\text{wl} = \wedge_{i=1}^{k} \text{con}_i$, where $\text{con}_i = \forall t \in [T_{1_i}, T_{2_i}]$: $\text{lhs}_i \oplus_i \text{rhs}_i$. $\text{cost}_S(\text{wl})$ is defined in the following way:

$$\text{cost}_S(\text{wl}) = \Sigma_{i=1}^{k} \text{queue_cnt}(\text{spec}_i) + \text{interval_cnt}(\text{wl})$$

where $\text{queue_cnt}(\text{spec}_i)$ is the number of queues constrained by con_i, which is equal to the number of queues specified in lhs_i. $\text{interval_cnt}(\text{wl})$ captures the degree of time fragmentation in the workload and is defined as the number of non-overlapping time intervals with unique sets of constraints.

As an example, suppose wl has three constraints, $\forall t \in [1,15] : cenq(Q1,t) \geq 2$, $\forall t \in [3,7] : cenq(Q2,t) \leq 5$, and $\forall t \in [5,10] : aipg(Q5,t) = 3$. These three constraints are specifying a traffic pattern over five non-overlapping time intervals ($[1,2],[3,4],[5,7],[8,10],[11,15]$) each with a unique set of constraints. So, interval_cnt is equal to five in this example. We favor workloads that cause less time fragmentation as they are less likely to overfit to the example sets, more concise, and more interpretable.

C Details on Search Engine Optimizations

Reducing the search space. We have described one of our optimizations to reduce the search space in §6.5. Another optimization is detecting and ignoring "duplicate" workloads. Our workload language allows for easy mutation of workloads with simple operations during search to generate new candidates. So, it is possible for a workload's mutation to represent the same set of traces while being syntactically different. We perform several checks to detect such workloads and avoid generating them as candidates, reducing the space of workloads the search algorithm needs to explore.

Reducing calls to the verification engine. As we describe in §6.5, if the search algorithm selects a candidate workload that matches a trace in B, it can move on to finding the next candidate without consulting with the verification engine, as it already knows that the current candidate includes a trace that does not satisfy the query. Note that the search algorithm selects these candidates despite that fact that they match traces in B as they could help it explore different regions of the search space. Similarly, if a workload is rejected because it is infeasible (§6.1), the search engine will keep track of it and avoid a potentially expensive call to the verification engine if that workload comes up in the future.

Other optimizations. Instead of only applying one of the operations in §6.2 and generating one candidate workload, we apply all of them one at a time, generate a set of candidates, and pick one randomly from the ones with the lowest cost. This helps the algorithm explore the lower-cost regions of the search space earlier. Moreover, we introduce a new operation, replace, which replaces a randomly-chosen constraint in the workload with a new random constraint. Replace is equivalent to a remove followed by an add, but it helps the algorithm to generate a more diverse set of candidates faster. Finally, if the search algorithm selects a candidate workload that is infeasible, adding or modifying constraints in the workload to generate the next one are likely to yield another infeasible candidate. So, the algorithm backtracks to the last known feasible candidate and continues from that point.

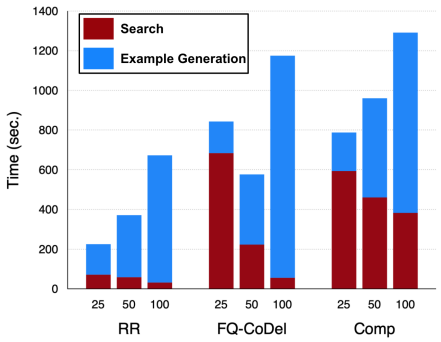


Figure 10: Search and example generations times for $|\mathsf{G}| = |\mathsf{B}| = 25$, 50, and 100. Prio results are not shown as the total time was less then 20s for Prio and would not be visible in the plot.

D More Details on the Leaf-Spine Case Study

D.1 Introducing New Packet Metadata

In the leaf-spine case study §8, packets have two metadata variables: dst, representing the final destination of the packet, and $ecmp$, with values in $[0,S]$ where S is the number of spines switches, representing the result of the ECMP hash of the packet's flow id modulo the number of spines.

We define the metrics $dst(q,t)$ as the destination of packets that enter q at time t and $ecmp(q,t)$ as the ECMP hash modulo number of spines for those packets. So, our workloads describe traffic patterns in which packets entering a queue at the same time have the same dst and $ecmp$.

For generating the base example, we add another optimization criteria, to maximize the "smoothness" of flows. That is, for the traffic entering from the hosts, the trace should not introduce new flows or go back and forth between flows with different dst and $ecmp$ if not needed for satisfying the query. When generating the rest of the good examples, we maintain the same "smoothness" criteria, and when minimizing the distance between eg_0 and eg_i, we include the difference in per-packet metadata in the computing the distance.

D.2 From User Interface to Logical Formulas

In the user interface described in §8, for the base workload, the users can provide a list of constraints using an interface that abstracts away the logical operators and expressions in Figure 4. Here, we describe how these constraints are translated to the logical formulas in Figure 4.

- [t1, t2] q.m $\oplus$ c becomes $\forall t \in [t_1,t_2] : m(q,t) \oplus c$.
- [t1, t2] (q1.m + ... + qn.m) $\oplus$ c becomes $\forall t \in [t_1,t_2] : \Sigma_{q \in \{q_1,\cdots,q_m\}} m(q,t) \oplus c$.
- [t1, t2] q1.m $\oplus$ q2.m becomes $\forall t \in [t_1,t_2] : m(q_1,t) \oplus m(q_2,t)$

- [t1, t2] (q1.m + ... + qn.m)/t $\oplus$ c becomes $\forall t \in [t_1, t_2]$: $\Sigma_{q \in \{q_1, \cdots, q_m\}} m(q,t) \oplus c \cdot t$

For queries, the users provide a list of questions with a similar interface, asking if the value of a metric over a time period for one or more queues can go above or below a threshold. A single question [t1, t2] lhs $\oplus$ rhs becomes $\exists t \in [t_1, t_2]$: trans(qlhs $\oplus$ rhs), where trans(qlhs $\oplus$ rhs) is the translation of the left hand side similar to what is described above. A list of questions will translate to the conjunction of their equivalent logical formulas. Note that queries in the form of $\forall t \in [t_1, t_2] lhs \oplus rhs$ are still possible in the user interface by creating a separate question for each time step between t_1 and t_2. It is also possible to extend the user interface to directly specify $\forall$ queries.

D.3 Example Generation Optimizations

For our scalability experiments in Figure 7c, we started with the default $|\mathsf{G}| = |\mathsf{B}| = 50$. However, example generation is expensive, and automated theorem provers (e.g., Max-SMT and SMT solvers) which we use in example generation and workload verification, can take exponentially longer as the problem size increases. So, to be able to observe the trends for larger networks, we used $|\mathsf{G}| = |\mathsf{B}| = 25$

We also employed extra optimizations when generating G. Recall that when generating the base example, we minimize the number of queues that have traffic in them. If a queue does not receive any traffic in the base example trace eg_0, it will stay empty in the rest of the traces in G. So, once eg_0 is generated, we create a "reduced" model in which remove the input queues that are marked as empty in eg_0 as they would be empty in the rest of the examples anyway. This helps reduce the number of variables and constraints, specifically in the crossbar modules of the leaf switches.

Moreover, recall that when generating trace eg_i, we constrain it to have different randomly chosen values from eg_{i-1} in $p = P$ random places. If no trace is found in two tries, we decrement p and try again. Instead of fixing the starting point to $p = P$, we set it to the moving average of the ps the worked when generating the last K examples.

Flattened Clos: Designing High-performance Deadlock-free Expander Data Center Networks Using Graph Contraction

Shizhen Zhao[1,*], Qizhou Zhang[1,*], Peirui Cao[1], Xiao Zhang[1], Xinbing Wang[1], Chenghu Zhou[1,2]
[1]*Shanghai Jiao Tong University,* [2]*Chinese Academy of Sciences*

Abstract

Clos networks have witnessed the successful deployment of RoCE in production data centers. However, as DCN bandwidth keeps increasing, building Clos networks is becoming cost-prohibitive and thus the more cost-efficient expander graph has received much attention in recent literature. Unfortunately, the existing expander graphs' topology and routing designs may contain Cyclic Buffer Dependency (CBD) and incur deadlocks in PFC-enabled RoCE networks.

We propose Flattened Clos (FC), a topology/routing co-designed approach, to eliminate the PFC-induced deadlocks in expander networks. FC's topology and routing are designed in three steps: 1) logically divide each ToR switch into k virtual layers and establish connections only between adjacent virtual layers; 2) generate *virtual up-down* paths for routing; 3) flatten the virtual multi-layered network and the virtual up-down paths using graph contraction. We rigorously prove that FC's design is deadlock-free and validate this property using a real testbed and packet-level simulation. Compared to expander graphs with the edge-disjoint-spanning-tree (EDST) based routing (a state-of-art CBD-free routing algorithm for expander graphs), FC reduces the average hop count by at least 50% and improves network throughput by $2-10\times$ or more. Compared to Clos networks with up-down routing, FC increases network throughput by $1.1-2\times$ under all-to-all and uniform random traffic patterns.

1 Introduction

Driven by the need of low latency, high throughput and low CPU overhead, large Internet service providers such as Microsoft and Alibaba have deployed RDMA over Commodity Ethernet (RoCE) [14, 20] in their Clos data centers. RoCE requires a lossless network for optimal performance. To avoid packet loss in Ethernet, Priority-based Flow Control (PFC) is usually enabled to perform a hop-by-hop flow control to avoid exhausting switch buffers by upstreaming flows. However, enabling PFC introduces the risk of deadlocks, especially for the large-scale deployment of RoCEv2. Thanks to the layered structure of Clos data centers, the up-down routing in Clos networks can prevent deadlocks with proper safety mechanisms under normal operations [20] and failure scenarios [24].

However, as the data center traffic and the network bandwidth keep increasing, building Clos topologies is becoming cost-prohibitive [4]. In order to reduce the network cost, flatter expander graphs, such as Jellyfish [46], SlimFly [5], Xpander [50], FatClique [54], etc., have been proposed to build data centers. A recent study [36] shows that a full throughput expander uses 25% fewer switches than a full throughput Clos. Note that the throughput values of expander graphs are attained using a multi-commodity flow formulation based on the K-Shortest Path (KSP) routing [53]. Unfortunately, the KSP routing in expander graphs may contain Cyclic Buffer Dependency (CBD), and thus could incur severe PFC deadlocks. Therefore, the performance-gain or cost-reduction of expander graphs over Clos becomes questionable for RoCEv2 traffic.

The key to supporting RoCE in expander graphs is to eliminate CBD. Approaches to eliminate CBD can be generally grouped into three classes. The first approach is to assign different lossless priorities for packets at different hops [12, 15, 27]. This approach has been widely adopted in HPCs, in which the underlying Infiniband network supports 15 lossless priorities (*a.k.a.* Virtual Channel). However, due to the limited switch buffer space, data center switches can support at most two or three lossless priorities [20]. The second approach is to disable PFC and redesign RoCE to work with lossy networks, e.g., NDP [21], IRN [35], FatPaths [6], etc. However, lossy RoCE requires hardware support. For example, Mellanox ConnectX-4 onwards NICs support lossy RoCE, but Mellanox ConnectX-3 NICs do not. In addition, lossy RoCE may incur higher latency for mice flows, especially when a sender has to rely on a timeout to retransmit a lost packet. iWarp [41] is another RDMA technology that runs on lossy networks. However, its performance is poor because it relies on TCP to guarantee lossless delivery.

*These authors contribute equally to this work.

The third approach is to design a routing solution that is fundamentally free of CBD, just as the up-down routing in Clos. Along this direction, TCP Bolt [48] and DF-EDST [47] were proposed, and the key idea is to find as many edge disjoint spanning trees (EDST) in an expander graph as possible and then route each packet in one of the spanning tree from its source to its destination. The EDST-based routing is CBD-free, but its throughput performance is poor. The key reason is that the EDST-based routing cannot effectively utilize all the network resources: 1) the average path length is usually large and increases quickly with network size; 2) some network links could remain idle as they do not belong to any EDST.

Our work called **Flattened Clos (FC)** offers a novel topology-routing co-design to eliminate CBDs in RoCE networks. FC's topology is essentially a random regular graph that is mappable to a multi-layered topology. We construct FC's topology in two steps: 1) virtually split each ToR switch into k virtual switches, each of which belongs to a virtual layer, and 2) randomly interconnect the layer-i ($i = 2, ..., k-1$) virtual switches to the layer-$(i-1)$ and the layer-$(i+1)$ virtual switches. The multi-layered virtual structure of FC allows performing up-down routing based on virtual layers. To this end, we propose the Edge-Disjoint Virtual Up-Down Routing for FC. For every source-destination pair, FC's routing transforms the path-finding problem into a min-cost-flow problem and then finds the maximum number of edge-disjoint paths. We analyze FC's design as follows to demonstrate its feasibility:

1. We offer a theoretical guidance for choosing the right number k of virtual layers when constructing FC's topology (see the strategy (*) in Section 3.2.3), and validate the strategy via numerical analysis.

2. We prove that FC's routing is CBD-free, and thus is deadlock-free. In fact, FC's topology and routing paths can be viewed as contracted graphs of a virtual multi-layered network and virtual up-down paths. This *graph contraction* operation preserves the CBD-free property.

3. We show that FC's cabling complexity can be dramatically reduced by introducing a layer of Patch Panels (PP) or Optical Circuit Switches (OCS) to interconnect all the ToR switches. Admittedly, having this PP/OCS layer increases cable length and cable cost. As network size becomes large, the overall network cost of FC is still lower than that of Clos under similar bisection bandwidth.

4. We demonstrate that FC outperforms expander graphs with EDST routing [47, 48] (the state-of-art CBD-free routing for expanders). Specifically, FC reduces the average hop count (AHC) by at least 50% and increases network throughput by $2-10\times$ or more.

5. We compare the throughput performance between FC and Clos networks with up-down routing, built using the same number of hosts and electrical switches. FC achieves $1.1-2\times$ throughput for all-to-all and uniform random traffic patterns, but its near-worst throughput is lower. We argue that when OCSs are used to construct FC, vendors do not have to worry much about FC's near-worst throughput. By simply generating a different FC's topology, one can avoid matching an FC's topology with its near-worst traffic patterns.

6. We validate that FC is deadlock free using a small test bed and a packet-level simulator, even under extreme (but practical) cases where congestion control is disabled and switches are misconfigured with a very small PFC PAUSE threshold. In contrast, we see deadlocks triggered under shortest-path routing and thus ECMP&KSP routing is not safe.

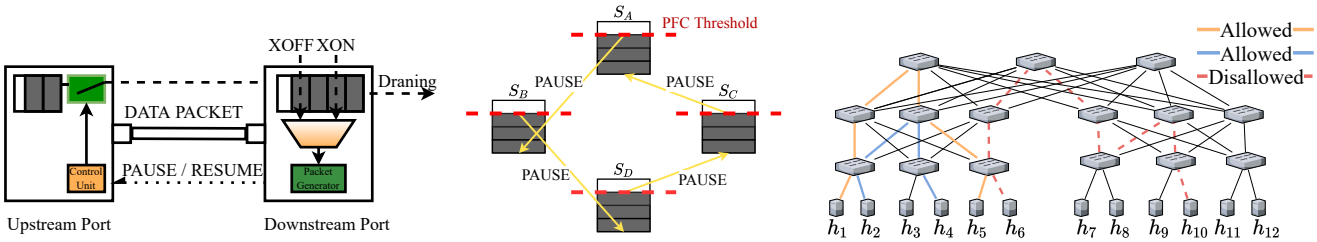
2 Background & Motivation

2.1 Deploy RDMA over Ethernet in Clos

Clos network, a.k.a. fat-tree, was proposed for data center network (DCN) architecture in [3], and has become the de-facto standard for large service providers, such as Google [45], Microsoft [19], Facebook [44], etc. TCP/IP is the dominant transport/network stack in today's data centers. However, the traditional TCP/IP stack cannot offer high throughput ($>$ 40Gbps or more) and ultra-low latency ($<$ 10us per hop) for modern data center applications such as cloud storage, deep learning framework and database [20, 30, 57]. Therefore, data center operators, e.g., Microsoft [20], Alibaba [30], etc., have started large-scale deployment of RDMA in Clos data centers to attain better network performance.

RDMA is a hardware offloading technology that offers several benefits such as high throughput, low latency and low CPU overhead by bypassing the host networking stack. HPC community has long used RDMA in special-purpose clusters, and deployed RDMA using Infiniband (IB) technology. However, modern data centers are built with IP and Ethernet technologies. For technical and economical reasons, RoCE was proposed for RDMA deployment in data centers.

The commonly used RoCE protocol is RoCEv2. RoCEv2 encapsulates an RDMA transport packet within a UDP packet to be compatible with the existing networking infrastructure of data centers. RoCEv2 was initially designed to run on a lossless network, which can be guaranteed by enabling the Priority-based Flow Control (PFC) [25]. Admittedly, there have been advanced RoCE designs, e.g., Resilient RoCE, IRN [35], etc., that could work with a lossy network. However, supporting RoCE in lossy networks requires handling packet retransmission using time out, selective ack, etc., which may not only complicate the NIC design, but also hurt network latency and throughput performance. As a result, lossy RDMA may not be able to substitue lossless RDMA in all cases. In this paper, we focus on lossless networks to support RoCEv2.



(a) Priority-based flow control. (b) PFC-induced deadlock. (c) Up-Down routing in a Clos network.

Figure 1: Key technologies of supporting RoCE in a lossless Clos network.

2.1.1 Priority-based Flow Control (PFC)

PFC is a hop-by-hop flow control approach to prevent switch buffer overflow, which is the primary cause of packet loss in data centers. As shown in Fig. 1(a), the downstream switch sends a PAUSE frame to its upstream switch when its ingress queue length exceeds a certain threshold (**XOFF**). The upstream switch stops transmission after receiving the PAUSE frame. A RESUME frame is sent when the downstream queue drains below another threshold (**XON**). It takes some time for the upstream switch to react to the PAUSE frame and stop transmission. So the downstream switch needs to reserve some buffer space to accommodate the packets sent by the upstream switch during this time. The buffer space is called *headroom*. PFC can guarantee zero packet loss when the headroom size is configured correctly. Typically, data center switches can support at most two or three lossless priorities [20] due to the buffer size limit. Although the switch buffers keep increasing, the data center link bandwidth has been increasing much faster and the buffer/bandwidth ratio is actually decreasing over time [18]. Hence, we believe that supporting more lossless priorities can be even more difficult for the foreseeable future.

2.1.2 PFC-induced Deadlocks

PFC can raise some performance issues such as unfairness, PFC storms and deadlocks [14, 20, 30, 57]. Specifically, the PFC-induced deadlocks may hinder the large-scale deployment of RoCEv2. When cyclic buffer dependency (CBD) exists, deadlocks can be triggered by PFC PAUSEs [23], causing packets to wait indefinitely for buffer resources [48]. As shown in Fig. 1(b), four switches S_A, S_B, S_C, S_D have reached the PFC threshold and start to send PAUSE frames; then the network is trapped into a deadlock and no switch can make any progress. Note that, the PFC-induced deadlock cannot go away once it occurs even if we restart all the servers.

Deadlock recovery is a common approach to combat deadlocks. It contains two steps: deadlock detection and deadlock resolution. Traditional approaches detect deadlocks in the control plane [34]. However, these solutions cannot react to deadlocks quickly enough due to the large communication latency between data planes and control planes. A recent work, ITSY [51], could detect deadlocks in the data plane and achieve at least 3.2× faster detection speed. However, ITSY requires programmable switch hardware (e.g., P4) support. As for deadlock resolution, temporary rerouting [34] is a common approach, but may create new congestion and deadlocks. ITSY [51] tried to resolve deadlocks completely in the data plane without rerouting, but the proposed solutions either incur packet loss or cannot efficiently handle concurrent deadlocks. To sum up, existing deadlock recovery mechanisms are not ideal. As a result, deadlock prevention has received much attention in the recent literature.

2.1.3 Avoiding Deadlocks in Clos Networks

Large vendors have gained years of experience in deploying RDMA in Clos data centers [20]. The following strategies are adopted to avoid deadlocks:

1. Perform up-down routing, which is CBD and deadlock-free under normal network conditions in Clos networks. (Note that containing a CBD is a necessary condition to have deadlocks.) As shown in Fig. 1(c), the paths of $h_1 \rightarrow h_5$ and $h_2 \rightarrow h_4$ obey the "up-down" rule and are allowed; but the path of $h_6 \rightarrow h_{10}$ contains a "down-up" segment and thus is not allowed.

2. Do not put multicast and broadcast packets into lossless classes. It was reported in [20] that ARP broadcasts+up-down routing can cause PFC deadlocks.

3. Use a different lossless class for rerouted packets upon network failures. [24] shows that packet rerouting may break the "up-down" rule and trigger PFC deadlocks.

2.2 From Clos to Expander

Despite of the success of deploying Clos data centers, however, a Clos network is inherently suboptimal in terms of bandwidth provision. As the Ethernet speed keeps increasing, the network cost, especially the power consumption of Clos networks, is becoming prohibitively high [4]. To reduce

the network cost, researchers have started seeking for more cost-effective network architectures.

One of the promising alternative for DCNs is expander graph. As shown in Fig. 2, expander graphs adopt a flat topology design: servers connect to ToR switches and these ToRs are directly interconnected without a layered structure. Examples of expander graphs include Jellyfish [46], SlimFly [5], Xpander [50], FatClique [54], etc. Expander graphs is more cost-effective for bandwidth provision than Clos networks. Using KSP routing, a full throughput expander uses 25% fewer switches than a full throughput Clos [36]. The network performance of expander graphs was also studied under other routing protocols, including ECMP, Valient Load Balancing (VLB) and a hybrid of the two [28]. However, none of these widely studied routing strategies is CBD-free.

2.2.1 ECMP/KSP are not CBD-free in Expanders

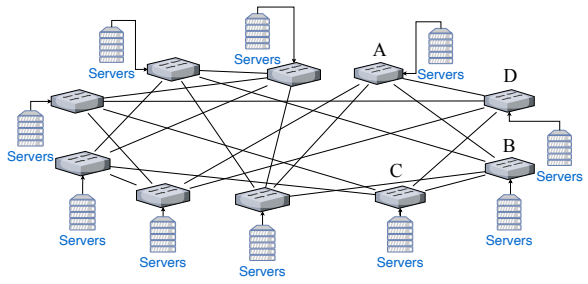


Figure 2: An expander graph.

Consider the expander graph in Fig. 2. This expander is a random regular graph with 4 inter-ToR links per ToR. Consider the four ToRs A, B, C, D and the shortest paths for $A \to C$, $B \to D$, $C \to A$, $D \to B$. Assume that there is a flow routed along the shortest path $A \to B \to C$. Then, if the egress port at link (B,C) is paused, the egress port at link (A,B) will be paused. If there is another flow routed along the shortest path $D \to A \to B$, since the egress port at link (A,B) is paused, the egress port at link (D,A) will also be paused. Similarly, if we have another two flows routed along the shortest paths $C \to D \to A$ and $B \to C \to D$, then the egress ports at link (C,D) and link (B,C) will be paused. Now, we find a CBD in this expander graph under shortest-path routing. To sum up, when there are 4 flows routed along the paths $A \to B \to C, D \to A \to B, C \to D \to A$ and $B \to C \to D$, if one of the egress ports $(A,B),(B,C),(C,D),(D,A)$ is paused for a sufficiently long time, a deadlock will be triggered.

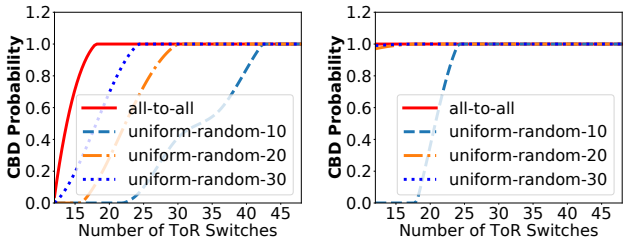
The above analysis indicates that shortest-path routing is not CBD-free. Now we consider ECMP and KSP routings. ECMP uniformly split traffic among all the shortest paths, while KSP split traffic among the first K shortest paths. (To improve network performance under ECMP/KSP, one can also optimize the path weights using a multi-commodity flow formulation.) Under ECMP or KSP routing, it is still possible to have four flows taking the paths $A \to B \to C, D \to A \to B, C \to D \to A$ and $B \to C \to D$ in the above example. Therefore, both ECMP and KSP routings are not CBD-free. Using the same approach, we can prove that the VLB routing and the hybrid of ECMP&VLB in [28] are not CBD-free, either.

2.2.1.1 Probability of Containing CBDs

We further analyze the probability of an expander graph containing CBDs under different traffic patterns. We generate two classes of expander graphs, Jellyfish [46] and Xpander [50]. In each expander graph, each ToR switch has 5 ports connected to other ToRs. For each expander graph, we evaluate two classes of traffic patterns under shortest-path routing (the algorithm that determines if a set of paths is CBD free in a given topology is offered in Appendix A.2):

All to All: Every source-destination pair has an on-going flow. This represents the most-likely case of having CBDs.

Uniform Random-p: Every ToR randomly picks p fraction of ToRs to communicate. This represents practical DCN traffic patterns in which the majority of traffic of a server is often destined to a few racks [44].



(a) Jellyfish CBD probability. (b) Xpander CBD probability.

Figure 3: Jellyfish and Xpander CBD analysis.

The results are depicted in Fig. 3. We can see that as the number of ToRs increases, the CBD probability quickly increases to one and Xpander graphs are more likely to encounter CBDs than Jellyfish graphs. Note that even under shortest-path routing (ECMP), the CBD probability becomes one with only tens of ToR switches. Other routing algorithms, including KSP, VLB, etc., contain even higher CBDs.

Remark on the necessity of eliminating CBDs: Even if the probability that the ECMP/KSP/VLB routing policies lead to CBDs is close to 1, the possibility that these CBDs eventually turn into deadlocks may not be that high. Nevertheless, eliminating CBDs can be still important. Some network applications requires five-nines availability, which means that the maximum downtime in a month must be less than 26.3 seconds. As long as the deadlock probability is non-zero, when a data center runs for a long time, a deadlock may be triggered eventually and hurts the overall system availability.

Remark on Tagger's approach: Given any routing paths, Tagger [24] offered a generic approach to eliminate CBDs. The key idea is to break each path into several segments

and assign each segment a lossless priority. As long as the path segments belonging to the same lossless priority are CBD-free, the entire network is CBD-free. Unfortunately, this approach may require too many lossless priorities to eliminate CBDs in large expander networks.

3 Flattened Clos

We propose a new class of expander graphs, called **Flattened Clos (FC)**, for efficient deadlock prevention. Our design is motivated by the CBD-free up-down routing in Clos networks and the *flattened butterfly* topology [29]. FC is a topology built on top of ToR switches. The key idea of FC is to split each ToR switch into a few virtual switches, and assign each virtual switch a virtual layer id. By creating links only between virtual switches in adjacent virtual layers, FC can adopt virtual up-down routing to avoid deadlocks. The detailed topology and routing designs of FC are described below.

3.1 Topology

We study a data center network with N ToR (Top of Rack) switches $S = \{S_1, S_2, ..., S_N\}$. Each switch has $p = s + h$ ports, h of which connected to the hosts and s of which connected to other ToRs. We construct an FC topology in two steps:

Step 1: Splitting Virtual Switches. To create an FC with k virtual layers, we logically split each switch $S_i, i = 1, 2, ..., N$ into k virtual switches, and label these virtual switches as $S_i^1, S_i^2, ..., S_i^k$. The virtual switch S_i^j belongs to the j-th layer, and has l_j number of links to connect to other switches. The total link count of the virtual switches $S_i^1, S_i^2, ..., S_i^k$ is equal to the total link count of S_i that connect to other ToRs, i.e.,

$$\sum_{j=1}^{k} l_j = s. \quad (1)$$

Step 2: Random Wiring. For each $j = 1, 2, ..., k-1$, we randomly generate a bipartite graph between the virtual switches in layer j and the virtual switches in layer $j+1$. Let $a_j, j = 1, 2, ..., k-1$ be the degree of each virtual switch in the j-th random bipartite graph. We must have

$$l_1 = a_1, l_2 = a_1 + a_2, ..., l_{k-1} = a_{k-2} + a_{k-1}, l_k = a_{k-1}. \quad (2)$$

When we generate random bipartite graphs, we never create links between S_i^j and S_i^{j+1} for $i = 1, 2, ..., N, j = 1, 2, ..., k-1$. The reason is that S_i^j and S_i^{j+1} actually belong to the same switch, and there is no need to create a link in between.

3.1.1 Theoretical Topology Properties of FC

In an FC's topology, each ToR switch has s links connected to other ToRs. Thus, FC falls into the category of random regular graphs (RRG). Here, we restate some useful theoretical results for RRGs in literature, which also applies to FC.

We represent a network by $G = (V, E)$, where V is the vertex set and E is the edge set. The bisection bandwidth of G can be characterized by *Edge Expansion*, which is defined as $EE(G) = min_{|S| \leq \frac{N}{2}} \frac{|\partial S|}{|S|}$, where N is the number of vertices in V, S is a subset of V, $|S|$ is the size of S, ∂S is the set of edges leaving S. The Edge Expansion of an s-regular graph is upper bounded by $s/2$ [50]. The following theorem indicates that random regular graphs attain near-optimal edge expansion.

Theorem 1 *(Near-optimal Edge Expansion [7]) For every $s \geq 3$ and $0 < \eta < 1$ satisfying $2^{4/s} < (1-\eta)^{1-\eta}(1+\eta)^{1+\eta}$, almost every s-regular graph G has its edge expansion*

$$EE(G) \geq (1-\eta)s/2.$$

Given a traffic matrix $T = [t_{uv}]$, where t_{uv} is the amount of requested flows from ToR switch u to ToR switch v. The throughput $\alpha(G, T)$ of a network G under the traffic matrix T is defined as the maximum value $\theta(T)$ for which $T \cdot \theta(T)$ is feasible in G. The following two theorems guarantee that random regular graphs achieve good throughput under both uniform and adversarial patterns.

Theorem 2 *(High throughput under all-to-all pattern [50]): For the all-to-all traffic pattern $T_{all\text{-}to\text{-}all}$, almost every s-regular graph G achieves a throughput*

$$\alpha(G, T_{all\text{-}to\text{-}all}) \geq \frac{1}{O(\log s)} \alpha(G^*, T_{all\text{-}to\text{-}all}),$$

where G^* is the s-regular graph that attains the optimal throughput under $T_{all\text{-}to\text{-}all}$.

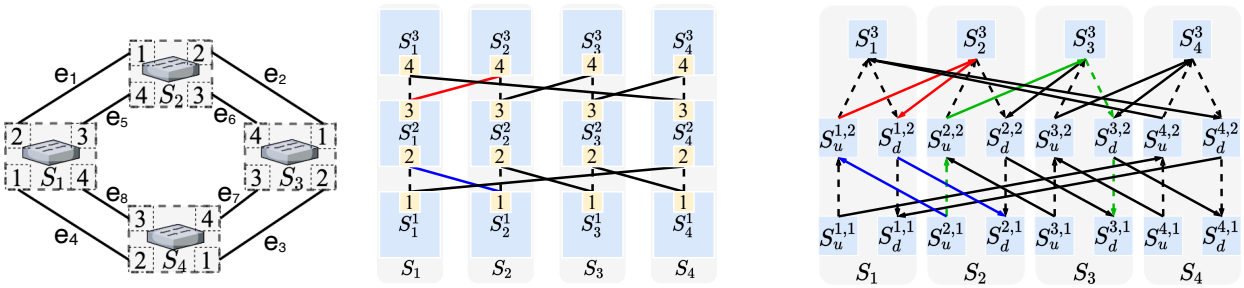
Theorem 3 *(Resilience to adversarial patterns [50]): For almost every s-regular graph G and every traffic pattern T, the throughput $\alpha(G, T) \geq \frac{1}{O(\log N)} \alpha(G^*, T)$, where G^* is the s-regular graph that attains the optimal throughput under T.*

3.2 Routing

3.2.1 Edge-disjoint Virtual Up-down Routing

Although FC's topology exhibits high network throughput in theory, such a throughput may not be achievable in PFC-enabled RoCE networks due to the potential risk of deadlocks. To completely eliminate the risk of deadlocks, we propose the CBD-free **Edge-disjoint Virtual Up-down Routing**. This routing strategy computes paths in three steps:

Step 1: Construct a Multi-layered Virtual Topology. According to the construction of FC's topology, each FC's topology is mappable to a multi-layered topology. Consider the toy example in Fig. 4(a). While we construct this topology, we have virtually divided each ToR switch S_i into $k = 3$ sub-switches S_i^1, S_i^2 and S_i^3. The first port of S_i belongs to S_i^1; the second and the third ports belong to S_i^2; the fourth port belongs to S_i^3. It is easy to check that each edge in Fig. 4(a) corresponds to a solid line in Fig. 4(b). Note that the k sub-switches



(a) Flattend Clos. (b) Virtualized topology. (c) Directed virtual up-down topology.

Figure 4: FC's topology and routing design.

$S_i^1, S_i^2, ..., S_i^k$ can communicate with each other, because they belong to the same physical switch. Hence, we also create an edge between S_i^j and S_i^{j+1} for every $j = 1, 2, ..., k-1$ (see the dashed lines) in Fig. 4(b).

Step 2: Construct a Directed Virtual Up-Down Topology. Our objective is to find the maximum number of virtual up-down paths in the multi-layered virtual topology. To enforce this "up-down" constraint, we further convert the undirected multi-layered graph in Fig. 4(b) to a directed virtual up-down graph in Fig. 4(c). Specifically, we first split each virtual node $S_i^j (j < k)$ into one "up node" $S_u^{i,j}$ and one "down node" $S_d^{i,j}$ in the directed up-down graph. (Note that we do not split the top layer virtual nodes S_i^k.) We then map each link in Fig. 4(b) to two directed links in Fig. 4(c): each undirected link $(S_{i_1}^{k-1}, S_{i_2}^k)$ in the top layer (see the red line in Fig. 4(b) as an example) is mapped to two directed links $(S_u^{i_1,k-1}, S_{i_2}^k)$ and $(S_{i_2}^k, S_d^{i_1,k-1})$; each undirected link $(S_{i_1}^{j-1}, S_{i_2}^j)$ $(j = 2, ..., k-1)$ (see the blue line in Fig. 4(b) as an example) is mapped to two directed links $(S_u^{i_1,j-1}, S_u^{i_2,j})$ and $(S_d^{i_2,j}, S_d^{i_1,j-1})$.

Step 3: Compute CBD-free Paths. For every source-destination pair (S_i, S_j), we first find a path set $\mathcal{P}_{ij}$ with the maximum number of virtual up-down paths from the node $S_u^{i,1}$ to the node $S_d^{j,1}$ in the directed virtual up-down topology using min-cost max-flow (see Appendix A.1 for more details). In this set $\mathcal{P}_{ij}$ of paths, each solid link is used at most once while each dashed link can be used multiple times. Then, for every path $P \in \mathcal{P}_{ij}$, we map it to a path in FC's topology (Fig. 4(a)). For example, as shown in Fig. 4(c), we find one up-down path $S_u^{2,1} \to S_u^{2,2} \to S_3^3 \to S_d^{3,2} \to S_d^{3,1}$ (marked with green) for the source-destination pair (S_1, S_2). Since $S_u^{2,1}$, $S_u^{2,2}$ are from the ToR switch S_2 and S_3^3, $S_d^{3,2}$, $S_d^{3,1}$ are from the ToR switch S_3, this path can be contracted to $S_2 \to S_3$ in the FC's topology. Since each solid link is used at most once in $\mathcal{P}_{ij}$, the resulting paths in the FC's topology must be edge-disjoint.

3.2.2 FC's Routing is CBD Free

In FC's edge-disjoint virtual up-down routing, we first compute an up-down path set $\mathcal{P}_{ij}$ based on the directed virtual up-down topology, and then contract all the paths in $\mathcal{P}_{ij}$ to obtain the final paths for FC's topology. Let $\mathcal{P} = \cup_{i,j}\mathcal{P}_{ij}$ be the set of virtual up-down paths obtained from the directed virtual up-down topology. According to Theorem 8 in Appendix A.2, $\mathcal{P}$ is CBD free. In order to prove that the final set of paths in FC's topology is CBD free, we need the following definition and lemma (see Appendix A.2.1 for the proof).

Definition 1 *Given a set of nodes V, $\{V_1, V_2, ..., V_m\}$ is called a partition of V, if the following conditions are met: 1) $V_{m_1} \cap V_{m_2} = \emptyset$ for every $m_1 \neq m_2$; 2) $V_1 \cup V_2 \cup \cdots \cup V_m = V$.*

Lemma 4 *Given a graph $G(V,E)$, a path set $\mathcal{P}$ and a partition $\{V_1, V_2, ..., V_m\}$ of V, a graph and path set pair $(\hat{G}(\hat{V}, \hat{E}), \hat{\mathcal{P}})$ is called a contraction of $(G(V,E), \mathcal{P})$ if*

1. *every node in $\hat{v}_i \in \hat{V}$ corresponds to the vertex set V_i;*
2. *the number of edges between $\hat{v}_i$ and $\hat{v}_j$ is the same as the total number of edges between V_i and V_j in $G(V,E)$;*
3. *each path $\hat{P} \in \hat{\mathcal{P}}$ is a contraction of a path $P \in \mathcal{P}$, i.e., $\hat{P}$ is obtained by first replacing each vertex in P by a vertex in $\hat{V}$ and then removing cycles and duplicated vertices.*

Then, if the path set $\mathcal{P}$ is CBD-free in $G(V,E)$, the path set $\hat{\mathcal{P}}$ must be CBD-free in $\hat{G}(\hat{V}, \hat{E})$.

Apparently, FC's topology and routing path set can be viewed as a contraction of the directed virtual up-down topology and the corresponding virtual up-down path set $\mathcal{P}$. Since the path set $\mathcal{P}$ is CBD free in the directed virtual up-down topology, then according to Lemma 4, we immediately know that FC's routing path set is CBD free.

3.2.3 How Routing Affects FC's Topology Design?

We have described FC's routing and topology designs. Note that there is a critical parameter k in the design. If k is not properly chosen, FC's routing policy may not be able to find a path for some switch pair, thus hurting the connectivity of FC. In this section, we offer a theoretical guideline to determine the number of virtual layers in FC.

Number of Switches	Number of Servers	k_{min}	k	Average Number of Edge Disjoint Up-down Paths	Average Path Length of Edge Disjoint Up-down Paths	Minimum Number of Edge Disjoint Up-down Paths
500	12000	3	3	10.05	4.29	4.00
			4	16.08	4.57	12.00
1000	24000	3	3	7.10	4.43	1.00
			4	14.01	4.86	9.00
2000	48000	4	4	11.85	5.13	7.00
			5	15.77	5.36	11.00
3000	72000	4	4	10.63	5.30	6.00
			5	14.66	5.54	10.00
5000	120000	4	4	9.17	5.52	3.00
			5	13.28	5.75	9.00

Table 1: Choosing the right k for FC.

Lemma 5 *Let x be the number of ancestors in the virtual layer k for each layer-1 virtual node. If $x > \sqrt{(2+\varepsilon)N\ln N}$, where $\varepsilon > 0$ is an infinitesimal value, then as $N \to +\infty$, with probability 1, every pair of layer-1 virtual nodes has a common ancestor in the virtual layer k.*

Proof 1 *We use A_{ij} to denote the event that the virtual nodes S_i^1 and S_j^1 have no common ancestor in the virtual layer k. Then, the probability that A_{ij} happens is*

$$\begin{aligned} P(A_{ij}) &= \frac{C_{N-x}^x}{C_N^x} \le (1-\frac{x}{N})^x \\ &< \left(1-\frac{\sqrt{(2+\varepsilon)\ln N}}{\sqrt{N}}\right)^{\sqrt{(2+\varepsilon)N\ln N}} \\ &= \left(\left(1-\frac{\sqrt{(2+\varepsilon)\ln N}}{\sqrt{N}}\right)^{\frac{\sqrt{N}}{\sqrt{(2+\varepsilon)\ln N}}}\right)^{(2+\varepsilon)\ln N} \\ &< (1/e)^{(2+\varepsilon)\ln N} = N^{-(2+\varepsilon)}. \end{aligned}$$

Let A be the event that at least one pair of virtual nodes in layer 1 has no common ancestor in layer k. Then,

$$\begin{aligned} P(A) &= P(\cup_{i\neq j}A_{ij}) \le \sum_{i\neq j} P(A_{ij}) \\ &= \frac{N(N-1)}{2} \times N^{-(2+\varepsilon)} < \frac{1}{2}N^{-\varepsilon}. \end{aligned}$$

Then, $\lim_{N\to+\infty} P(A) = 0$. This completes the proof.

Based on FC's routing, it is easy to calculate that the number of distinct up-paths from a virtual node in layer 1 is $(a_1+1)(a_2+1)\cdots(a_{k-1}+1)$, which is an upper bound of the number of ancestors in layer k. According to Lemma 5, we can choose a k such that

$$(a_1+1)(a_2+1)\cdots(a_{k-1}+1) > \sqrt{(2+\varepsilon)N\ln N}. \quad (3)$$

According to Equation (1) and (2), it is easy to obtain $\sum_{i=1}^{k-1} a_i = s/2$. We could choose $a_1, a_2, ..., a_{k-1}$ to maximize the left hand side of (3), and obtain

$$\left(1+\frac{s}{2(k-1)}\right)^{k-1} > \sqrt{(2+\varepsilon)N\ln N}. \quad (4)$$

Let $k_{\min}$ be the smallest integer solution of (4). We could choose a k value around $k_{\min}$. As shown in Fig. 5, $k_{\min}$ does not grow fast with respect to N.

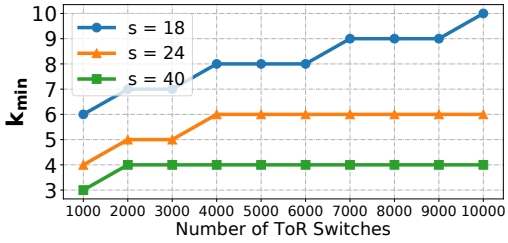


Figure 5: Relationship between $k_{\min}$ and network size N.

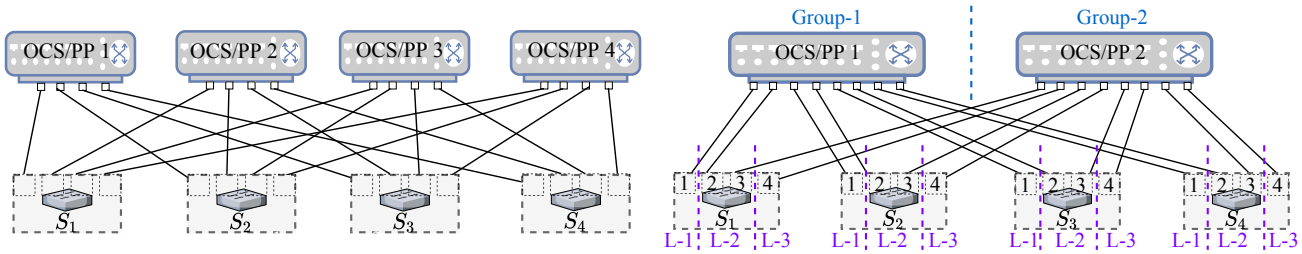
Numerical Verification: To verify the above theoretical result, we perform a numerical analysis using 64-port ToR switches. Each ToR switch has $h = 24$ ports connected to the hosts and $s = 40$ ports connected to other ToR switches. For different number of ToR switches ($N = 500/1000/2000/3000/5000$), we choose $k = k_{\min}, k_{\min}+1$, generate an FC's topology and count the number of distinct virtual up-down paths. As shown in Table 1, as we increase k, more paths can be found for every source-destination ToR switch pairs. Note that the average path length increases with respect to k. Hence, it is better to choose a smaller k. On the other hand, if we choose k to be too small, some ToR switch pairs may not have sufficient number of distinct paths. Here we suggest a simple strategy that works well for FC:

Strategy (*): Try $k_{\min}$ first; if not working, try $k_{\min}+1$.

For example, in the case where $N = 1000, k = k_{\min} = 3$, the minimum number of distinct paths between ToR pairs is 1. This creates a bottleneck in the network. Hence, $k = k_{\min} + 1 = 4$ will be chosen instead.

3.2.4 Computational Complexity of FC's Routing

The main complexity comes from using the min-cost max-flow solver to find edge-disjoint virtual up-down paths. Given a graph $G = (V, E)$ with n vertices and m edges, the computational complexity of the min-cost max-flow algorithm



(a) Uniform cabling through OCSs/PPs. Any inter-ToR topology is realizable by properly configuring the s OCSs/PPs one by one.

(b) Virtual-layered cabling through OCSs/PPs. There is 1 port for virtual layer-1 (L-1), 2 ports for L-2, and 1 port for L-3 in every switch. OCSs/PPs are divided into 2 groups.

Figure 6: Example of cabling with the help of optical circuit switches (OCS) / patch panels (PP).

implemented in Ortools is $O(m \cdot n^2 \cdot \log(n \cdot C))$, where C is the value of the largest link cost in the graph [1] (in our case, $C = 1$). If we choose the parameters k and $a_1, a_2, ..., a_{k-1}$ such that $(a_1+1)(a_2+1)\cdots(a_{k-1}+1) = \Theta(\sqrt{N \log N})$, each virtual node in the first virtual layer will be able to reach at most $\Theta(k\sqrt{N \log N})$ virtual nodes through at most $\Theta(k\sqrt{N \log N})$ edges. When we compute edge disjoint paths from S_i to S_j, we only need to focus on a subgraph of the directed virtual up-down topology, which contains all the nodes reachable from $S_u^{i,1}$ and $S_d^{j,1}$. This subgraph has $\Theta(k\sqrt{N \log N})$ nodes and edges. Thus, the overall computational complexity is $\Theta((k\sqrt{N \log N})^3 \cdot \log(k\sqrt{N \log N})) = \Theta(k^3 N^{3/2} (\log N)^{5/2})$.

3.3 Cabling

FC adopts random wiring for its topology design. However, random wiring has long been criticized for its high cabling complexity [50, 54]. Indeed, if we directly connect different ToR switch pairs, the number of distinct fiber lengths would be in the order of $\Theta(N^2)$. Directly connecting ToR switches could also increase the management complexity when we perform data center expansion [56].

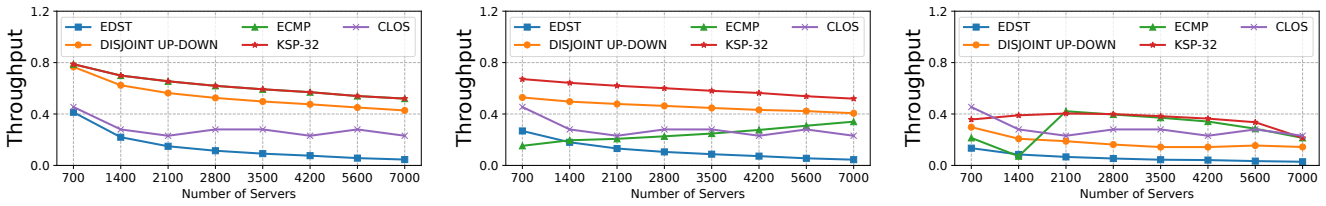
To reduce cabling complexity, motivated by TROD [9] and Google's Jupiter data center [40], we propose to use a set of co-located optical circuit switches (OCS) or patch panels (PP) to interconnect different ToR pairs and form FC's topology. Since these PPs/OCSs are co-located, the number of distinct fiber lengths reduces to $\Theta(N)$. Next, we offer two strategies to interconnect PPs/OCSs with ToR switches.

Uniform Cabling (see Figure 6(a)): Note that each ToR switch has s ports to be connected to other ToR switches. We use s OCSs/PPs, and construct a uniform bipartite graph between ToR switches and OCSs/PPs. Under this cabling strategy, it was proven in [55] (see Lemma 4 and Theorem 5 therein) that any inter-ToR topology is realizable by properly configuring the s OCSs/PPs one by one. According to this fact, we could first generate an FC topology without considering the layer of OCSs/PPs, and then decomposite this topology into s sub-topologies that can be mapped to each OCS/PP. This approach reduces cabling complexity. However, it encounters scalability challenge. Specifically, the port count of the commercially available OCSs/PPs is on the order of a few hundred. For example, a Calient s320 OCS [8] can offer 320 TX/RX ports. Thus, the number of ToR switches can be at most a few hundreds. Since each ToR switch typically connects to tens of servers, this uniform cabling strategy can support at most a few thousands of servers.

Virtual-Layered Cabling (see Figure 6(b)): Note that FC's topology is designed based on the concept of virtual layers. Assume that there are a_1 ports for layer-1, $a_1 + a_2$ ports for layer-2, ..., $a_{k-1} + a_k$ ports for layer-$(k-1)$, and a_k ports for layer-k. We group all the OCSs/PPs into $k-1$ groups, and connect $2a_i$ ports of each ToR switch to the i-th OCS/PP group. In the i-th OCS/PP group, each OCS/PP have half of its ports connected to ToR switches' layer-i ports and half of its ports connected to ToR switches' layer-$(i+1)$ ports. If we enforce that every OCS/PP should connect to all the ToR switches, we will encounter the same scalability challenge as the uniform cabling strategy. In the virtual-layered cabling strategy, $2\eta a_i$ number of OCSs/PPs are used in the i-th group, and each ToR switch will randomly choose $2a_i$ OCSs/PPs in group-i to connect its layer-i and layer-$(i+1)$ ports. Under this cabling strategy, the total number of ToR switches that can be supported becomes "$\eta \times$ port count of an OCS/PP". This strategy scales well. For example, if we use 320-port OCSs, 64-port ToR switches (assume that each ToR connects to 24 servers), and choose $\eta = 20$, then the maximum number of servers would be $20 \times 320 \times 24 = 153600$, which can definitely support a large-scale data center.

Remark on the parameter η: When $\eta > 1$, a cabling constraint is imposed to FC when we generate the topology between adjacent virtual layers, i.e., not all topologies are realizable because the interconnection between ToRs and each group of OCSs/PPs is not uniform. Fortunately, Appendix A.5.1 shows that enabling this cabling constraint when generating FC's topology has little impact on FC's routing statistics.

Remark on the network cost: Compared to traditional expander graphs, having a layer of OCSs in FC reduces the



(a) Throughput of the all to all traffic matrix. (The ECMP and KSP curves overlap together.)

(b) Throughput of uniform random traffic matrices (averaged over 10 runs).

(c) Throughput of the near-worst permutation traffic matrix.

Figure 7: Throughput simulation results under ECMP, EDST, Edge-disjoint Virutal Up-down and 32-way KSP routing.

number of distinct cable lengths, but unfortuanately increases the total cable length. We compare the total network cost between FC and Clos in Appendix A.4. To achieve similar bisection bandwidth, FC and Clos have similar network cost when the network size is small and FC's cost becomes lower as the network size increases to a point where 4 switch layers are required to build a Clos. In addition, FC uses fewer number of electrical switches and thus its network power consumption is lower.

A potential future direction: Using OCSs introduces another interesting problem: how to design deadlock-free and traffic-aware topology & routing policies. As reconfiguring OCSs incurs non-negligible delay, it may not be possible to reconfigure OCSs for every traffic-pattern change. Google's Jupiter data center [40] and our prior work [9, 49] both showed that low-frequency reconfiguration might be sufficent, because the traffic patterns in real data centers do not change arbitrarily. Low-frequency reconfiguration may also work for lossless RDMA networks, but requires further investigation.

4 Numerical Throughput Analysis

We numerically evaluate the throughput for FC in this section. We evaluate two scenarios. Due to space constraints, we only present one here and put the other one in Appendix A.5.2.

We generate FC's topologies of different sizes using up to 500 32-port ToR switches. Each ToR switch has 18 ports connected to other switches and 14 ports connected to servers. The number of virtual layers k is chosen based on the strategy (*) in Section 3.2.3. For each FC's topology, we evaluate four routing strategies: 1) FC's edge-disjoint virtual up-down routing, 2) EDST routing, 3) ECMP or Shortest-Path routing, and 4) KSP routing. Given a traffic matrix T, we use a multi-commodity flow formulation to calculate the maximum throughput value $\theta(T)$ such that $T \cdot \theta(T)$ is feasible under the given topology and routing paths. (For ECMP, the throughput is also calculated based on the multi-commodity flow formulation. Evenly spreading traffic among all the shortest paths may yield very poor throughput.) In addition, for each FC's topology, we also compare it with a Clos network generated using roughly the same number of switches with throughput optimized (see Appendix A.3).

We compute throughput values under all-to-all traffic patterns, uniform random traffic patterns and near-worst traffic patterns. In an all-to-all pattern, each server sends an equal amount of traffic to all other servers. In a uniform random pattern, each ToR randomly picks 10% of ToRs to communicate. To generate near-worst patterns, we 1) first construct a complete bipartite graph B with N source nodes and N destination nodes, where the weight of the edge (s,d) is the length of the shortest path from ToR s to ToR d; 2) and then find the permutation matrix with the maximum weight. This approach was also adopted in [26, 36] to generate near-worst patterns. We believe that the above three classes of traffic patterns offer an adequate coverage of real data center traffic patterns. The uniform random pattern is highly representative in real data centers. Indeed, Google's data center traffic patterns are approximately uniform random [40]. The all-to-all pattern is widely used in MPI communication. The near-worst pattern allows us to understand network's performance lower bound.

4.1 FC's Routing vs EDST Routing

The EDST routing is CBD-free for expander graphs. A random s-regular graph has $s/2$ edge-disjoint spanning trees with high probability [38]. Thus, EDST is a direct competitor of the Edge-disjoint Virtual Up-down Routing for FC's topology.

As shown in Fig. 7, FC's edge-disjoint virtual up-down routing (denoted by "DISJOINT UP-DOWN") performs consistently better than EDST for all the traffic patterns. When the network is small ($N = 50$), FC's routing achieves $2\times$ throughput of the EDST routing. As the network size increases, the performance of the EDST routing deteriorates quickly. When $N = 500$, the performance gain of FC's routing becomes $10\times$ and the gain keeps increasing with the network scale.

There are two reasons that lead to the poor performance of the EDST routing. First, existing edge-disjoint spanning tree (EDST) algorithms [42, 43] can find the maximum number of spanning trees, but there is no guarantee that the height of each spanning tree found is small. When we perform routing in a *tall* spanning tree, the average hop count would be large. This is also justified in the following routing-path analysis.

Number of Switches	Number of servers	k	Port Count of Virtual Switches	Routing	Average Number of Paths	Average Path Length	Average Shortest Path Length
50	700	4	[3, 6, 6, 3]	Edge Disjoint Up-down	8.02	3.86	2.68
				EDST	9.00	7.69	3.12
100	1400	4	[3, 6, 6, 3]	Edge Disjoint Up-down	6.43	4.22	3.04
				EDST	9.00	10.01	4.04
200	2800	4	[3, 6, 6, 3]	Edge Disjoint Up-down	4.99	4.56	3.44
				EDST	9.00	14.01	5.50
300	4200	4	[3, 6, 6, 3]	Edge Disjoint Up-down	4.24	4.75	3.70
				EDST	9.00	16.70	6.52
500	7000	5	[2, 4, 4, 5, 3]	Edge Disjoint Up-down	4.55	5.22	4.00
				EDST	9.00	21.96	8.14

Table 2: Edge-Disjoint Virtual Up-down Routing vs. the EDST Routing (32-port Switches are Used).

Second, some links remain unused in the EDST routing. In an expander graph with N ToR switches, each spanning tree contains $N-1$ links. Note that the total number of ToR-to-ToR links is $Ns/2$. When $Ns/2$ is not divisible by $N-1$, there must be links not used by any spanning tree.

Routing-Path Analysis: For several FC's topologies of different sizes ($N = 50/100/200/300/500$), we analyze the routing paths under FC's routing and the EDST routing. We calculate three metrics, including average number of paths, average length of paths and average length of the shortest paths. As shown in Table 2, although the EDST routing could find more paths than FC's routing, its average path length is much higher. When $N = 50$, the average path length under FC's routing is $1-3.86/7.69 \approx 50\%$ lower than that under the EDST routing. As N increases to 500, the reduction of average path length becomes $1-5.22/21.96 \approx 76\%$. We expect that this number will continue to increase for larger networks. The EDST routing cannot guarantee a small routing path length. In contrast, the parameter k restricts that FC's routing path length cannot exceed $2k$ and k increases slowly with N.

4.2 FC's Routing vs ECMP/KSP Routing

ECMP/KSP are widely-used routing protocols for expander graphs. In FC's topology, ECMP's throughput fluctuates significantly because ECMP cannot provide enough path diversity; KSP's throughput is more stable under different traffic patterns. This coincides with the findings in Jellyfish [46].

Fig. 7 shows that KSP's throughput is consistently higher than that of the FC's edge-disjoint virtual up-down routing. However, deploying KSP routing in expander networks poses a deadlock risk. We have shown in Section 2.2.1.1 that the probability that ECMP/KSP routing contains CBDs is close to 1. Although containing CBDs is not sufficient to trigger deadlocks, we will show in Section 6.1 that ECMP/KSP could indeed trigger deadlocks in certain cases in a real testbed.

How to close the throughput gap: FC uses only one lossless queue, and its throughput performance is lower than that of the KSP routing. The reason is that FC's routing has lower path diversity than the KSP routing. To improve path diversity, we could let FC use more than one lossless queues. We will explore this further in our future work.

4.3 FC vs Clos

Clos is the de facto standard topology for data centers and has witnessed the successful deployment of RDMA in production [20]. To ensure fair comparisons, given an FC's topology with N ToR switches and H servers, we choose a Clos network that offers the maximum throughput to the H servers using roughly the same number of switches (Appendix A.3).

As shown in Fig. 7(a) and 7(b), FC attains $1.1-2\times$ the throughput of Clos networks. Note that there is a decrease in throughput when the network size changes from 700 to 1400. The reason is that when the switch port count is 32, we can build a two-layered Clos to support 700 servers, but at least 3 layers are required in order for a Clos to support 1400 servers.

However, under near-worst traffic patterns, Fig. 7(c) shows that FC's throughput can be $15\%-50\%$ lower than that of the Clos networks. We argue that this issue can be resolved when a layer of OCSs is used to interconnect different ToRs. If the real traffic pattern is close to a near-worst pattern of the current topology, we can reconfigure the OCSs to generate a topology that matches this traffic pattern. Then, a natural question arises. How frequent should we reconfigure the topology? Certainly, the answer to this question depends on the traffic patterns. If the traffic patterns exbihit some long-term stability [40], occasional reconfigration might be sufficient. We will study this problem further in our future work.

5 Packet-Level Simulation

We cross-validate our throughput analysis using a packet-level simulator [22]. We generate an FC's topology using 144 32-port switches. Each switch has 8 ports connected to hosts and 24 ports connected to other switches. In total, there are 1152 hosts. On top of this topology, we run FC's routing or the EDST routing. We also generate a Clos network using 148 32-port switches with throughput optimized. This Clos network has 64 ToR switches, 56 aggregation switches and 28 spine switches. Each ToR has 18 ports connected to hosts and 14 ports connected to the aggregation switches. The toal number of hosts is still 1152. For this Clos topology, we use up-down routing. The port speed is set as 25Gbps.

We generate three sets of flows based on the all-to-all traffic pattern, a uniform random traffic pattern (each ToR choose 12.5% of ToRs to communicate) and the near-worst traffic pattern. In Facebook's data centers, the median link utilization varies between 10% to 20% and the busiest 5% utilization of links is between 23% to 46% [44]. Here, we set the network load as 0.3, meaning that the maximum ingress/egress traffic of each ToR is 0.3 × Number of Hosts per ToR × 25Gbps. For all the flows, we enable DCQCN for congestion control. We adopt dynamic PFC threshold such that the PFC is triggered when an ingress queue consumes more than 11% of the free switch buffer as suggested by HPCC [30]. We evaluate performance based on the flow completion time (FCT).

We summarize the FCT results in Table 3. FC attains higher throughput under the all-to-all pattern and the uniform random patterns. Correspondingly, FC achieves lower FCT in the packet-level simulation. FC's near-worst-case throughput is lower than that of Clos. But Fortunately, FC has lower average hop count and thus its FCT performance is not much worse. More detailed results are available in Appendix A.5.4.

6 Formation and Impact of Deadlocks

We study how to trigger deadlocks and understand the impact of deadlocks via both testbed experiments and simulations. We will show that under extreme but practical cases, FC's edge-disjoint virtual up-down routing is still deadlock-free; but ECMP/KSP could trigger deadlocks.

6.1 Trigger Deadlocks in a Real Testbed

We build a small testbed using four switches, each with 8 50Gbps ports. (The four switches are virtualized from a single CE12800 switch. The original port speed is 100Gbps and we limit the port speed as 50Gbps.) This testbed has 16 servers, each equipped with one Mellanox CX5 NIC with maximum rate configured as 50Gbps. (We use PCIE-3.0×8 to connect to the NICs, and thus these NICs cannot run at a rate higher than 64Gbps.) Each switch in this testbed has four ports connected to other switches and four ports connected to four servers. We virtually split each switch into 3 virtual switches, with $1,2,1$ number of ports respectively. The connections between FC's virtual switches are shown in Fig. 4(b), and the resulting topology is shown in Fig. 4(a). This topology can be also viewed as a subgraph of a large expander graph (see switches A,B,C,D in Fig. 2). If a deadlock occurs in this subgraph, a PFC storm will quickly propogate to the entire network.

We implement ECMP, edge-disjoint virtual up-down and EDST routings using ACL rules in our testbed. We enable PFC to guarantee that the network is loss-free. The PFC-pause threshold $XOFF$ is set to 50KB and the PFC-resume threshold XON is set to 47KB. Note that these PFC thresholds are lower than the recommended values. This allows the network to trigger more PFC pauses. As we will see shortly, the virtual up-down routing is deadlock-free even in this extreme situation. Note that this setup can be viewed as a misconfiguration of network switches. Microsoft reports that switch misconfiguration accounts for 38% of the high-impact failures in their data centers [52]. In a PFC-storm incident reported also by Microsoft [20], a switch parameter was misconfigured such that PFC PAUSE frames could be triggered more easily.

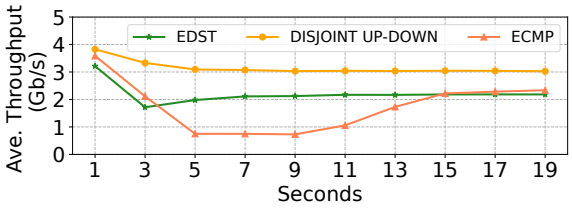


Figure 8: Average throughput of the testbed experiment.

We generate RoCEv2 traffic using the "ib_write_bw [31]" command. For every NIC, we establish an RDMA connection with every NIC under a different ToR switch. For example, NIC1 under the first ToR switch sends traffic to NIC5, NIC6, ..., NIC16. In total, we establish $16 \times 12 = 192$ RDMA connections. We configure the "- -run_infinitely" parameter at the client side of each connection to run the test indefinitely until interrupted by external.

Results: In the first experiment, we apply ECMP routing. ECMP is not CBD-free in this testbed. We see a deadlock after running our testbed for just a few seconds. (KSP typically generates more paths than ECMP, and thus KSP could also trigger deadlocks.) When deadlock happens, a large number of RDMA connections are broken. We deep dive into the source code of "ib_write_bw" to understand why many connections are tear down abnormally. We found that the PFC-deadlocks cause the verbs API "ibv_post_send" to fail and return an error code to the main program of "ib_write_bw". Once the main program catch the exception code, "ib_write_bw" will stop sending traffic and clean up the resources. Note that, if we use dynamic PFC thresholds or use the recommended values to set static PFC thresholds ($XOFF = 800KB, XON = 797KB$), we could not observe PFC deadlocks under ECMP routing. However, this does not eliminate the deadlock risk for ECMP.

In the second and third experiments, we set up the edge-disjoint virtual up-down and the EDST routing respectively to run the same test. In this case, we do not see any deadlock even under low PFC pause/resume thresholds and all RDMA connections can work continuously. This experiment demonstrate that both the virtual up-down routing and the EDST routing can avoid PFC-deadlock in lossless Ethernet.

Finally, we track the average throughput for all the 192 RDMA connections under different routing strategies over one minute, and plot the results in Fig. 8. The virtual up-down routing attains the highest average throughput, which is about 50% higher than that of the EDST routing. Under ECMP routing, the average throughput drops quickly at the

Network Setup	Num. of Hosts	Num. of Switches	Copper / Fiber Cable (km)	Num. of Transceivers	Network Cost (Million $)	All-to-All (load= 0.3)			Uniform Random (load= 0.3)			Near-Worst (load= 0.3)		
						Tput	P50 FCT	P99 FCT	Tput	P50 FCT	P99 FCT	Tput	P50 FCT	P99 FCT
FC	1152	144	2.3 / 55.5	3456	13.98	1.49	6.11	32.03	1.25	4.30	18.62	0.55	35.48	121.55
FC+EDST	1152	144	2.3 / 55.5	3456	13.98	0.48	12.42	99.88	0.39	196.46	509.73	0.16	7081.40	9889.53
Clos	1152	148	2.3 / 10.8	3584	10.77	0.78	11.65	44.68	0.78	6.95	39.55	0.78	42.67	118.74

Table 3: FCT Results vs. Throughput Analysis. ("Tput" is short for "Throughput".)

first 5 seconds due to PFC deadlocks. Although the average throughput of ECMP increases after deadlock recovery, 66% of the RDMA connections have already failed.

6.2 Understanding Deadlocks via Simulation

We perform packet-level simulation to understand how a deadlock is triggered. We use the same testbed topology (Fig. 4(a)) in our simulation. We generate 192 flows at time 0, and set all the flow sizes as 100MB. For different flows, we either disable congestion control or enable DCQCN for congestion control. When DCQCN is enabled, we set the ECN-marking related parameters as $Kmin = 5KB, Kmax = 200KB, P_{\max} = 0.01$ as suggested by the DCQCN paper [57]. To simulate the extreme cases where lots of PFC pauses are triggered, we set a small PFC-pause threshold and a small PFC-resume threshold ($XOFF = 50KB, XON = 47KB$).

We evaluate ECMP and FC's edge-disjoint virtual up-down routing. For adjacent switch pairs, there are two paths under both ECMP and FC's edge-disjoint virtual up-down routing. For non-adjacent switch pairs, there are 8 shortest paths (4 clock-wise paths and 4 counter-clock-wise paths) under ECMP routing and 2 edge-disjoint virtual up-down paths (1 clock-wise path and 1 counter-clock-wise path) under FC's routing. We assign a path to each flow using two strategies:

Balanced Allocation: There are 16 flows generated between every switch pair and we assign the same number of flows to each path under both routing strategies. In this case, every link between adjacent switch pair is shared by exactly 16 flows.

Imbalanced Allocation: Flows between adjacent switch pairs are still equally assigned to all the paths; but flows between non-adjacent switch pairs are only assigned to the clock-wise paths. This situation could happen due to hashing imbalance. In this case, every clock-wise link is shared by 24 flows, which becomes the bottleneck of the network. Incast can thus happen at the 4 switches. In addtion, under ECMP routing, the following paths $\{[e_1,e_2],[e_2,e_3],[e_3,e_4],[e_4,e_1]\}$ form a CBD (actually there are more CBDs), which makes ECMP prone to deadlocks.

Results: Under balanced allocation, we do not see deadlocks even if we use a small static PFC threshold and disable DCQCN. In Fig. 9, we compare the CDFs of the FCTs (Flow Completion Time) under both ECMP and FC's edge-disjoint virtual up-down routing with and without DCQCN. Both routing strategies yield similar FCT performancce.

Under imbalanced allocation, FC's routing can still finish all the flows and the FCT performance is shown in Fig. 9.

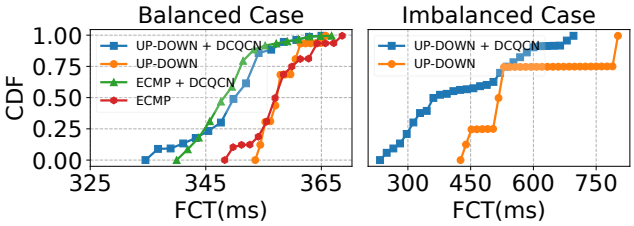


Figure 9: CDF of the FCTs of ECMP and FC's routing under balanced/imbalanced allocation in the testbed topology.

In contrast, the ECMP routing triggers a deadlock even if we enable DCQCN. To rootcause this issue, we record all the PFC pauses and PFC resumes. We find 4 critical PAUSE signals that lead to the deadlock: 1) at time 531 us, S_1 sends a PAUSE to the link e_4; 2) at time 537 us, S_4 sends a PAUSE to the link e_3; 3) at time 543 us, S_3 sends a PAUSE to the link e_2; 4) at time 552 us, S_2 sends a PAUSE to the link e_1. These events happen within just 21 us.

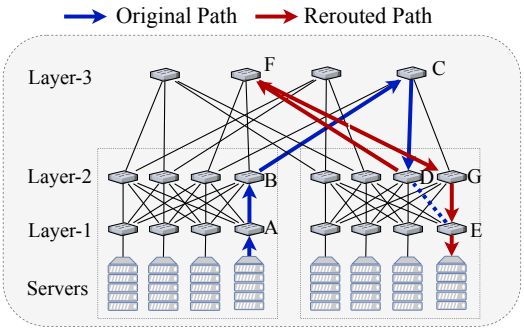
Takeaway: A DCN suffers from a high risk of deadlocks, when the following three conditions are met: 1) there exist CBDs in the network; 2) links in the CBDs are congested; 3) PFCs are triggered more frequently than usual. If we apply ECMP/KSP routing in an expander graph, we may have to constantly monitor the congested links and the abnormal switch behaviors. FC's design completely eliminates CBDs, and thus could significantly simplify the RoCEv2 deployment.

7 Discussion

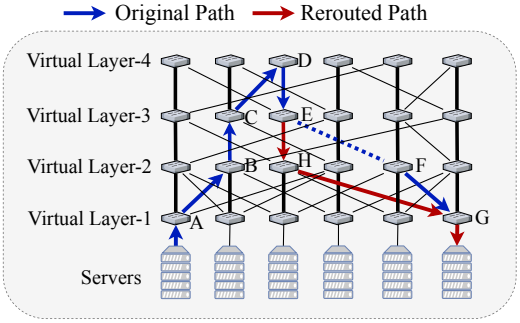
7.1 Handling Link/Node Failures

Link/node failures are common in practical data centers [16]. When a link/node fails, to avoid packet drop, local rerouting is performed to forward the affected packets along a different path to the destination [32]. Unfortunately, local rerouting may introduce CBDs and cause deadlocks even if the original network is CBD-free [24]. Consider the Clos network in Fig.10(a). Initially, packets from ToR A to ToR E follow an up-down path $A \to B \to C \to D \to E$. When the link DE fails, packets that arrived at the switch D cannot find an alternative downstream path to E and thus are bounced back to F. Then, the path from A to E becomes $A \to B \to C \to D \to F \to G \to E$. This path contains a down-up bounce, which could introduce CBDs into Clos networks.

To avoid deadlocks in Clos networks under link/node fail-



(a) Clos topology.



(b) FC's topology.

Figure 10: Rerouting under link failures.

ures, Tagger [24] adds a tag to all the packets, increases the tag on the bounce and puts packets with different tags into different lossless queues. This approach should also work for FC because we can treat FC as a virtual multi-layered network. Nevertheless, better approach may exist for FC. In Clos networks, every top-layer switch has a unique path to every ToR switch and thus every packet affected by a downstream link/node failure has to be bounced back to another top-layer switch. In contrast, every packet affected by a link/node failure in FC can freely choose any virtual layer as long as there is a link to forward this packet, because every virtual switch in the same column (see Fig. 10(b)) belongs to the same physical switch. For example, there is a flow from A to G in Fig. 10(b) and the original path is $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$. When the link EF fails, the affected packets can be rerouted to $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow H \rightarrow G$. This new path is still an up-down path and thus tagging is not required. (Admittedly, if the rerouted path contains a down-up bounce, we still need to update the packet tags.) Based on the above analysis, we suspect that FC could be more efficient in handling link/node failures than a Clos network. We will explore this further in our future work.

7.2 Handling Route Reconfiguration

Route reconfiguration is common in data centers, which could happen when 1) new flows join/leave the network; 2) DCN topology changes; 3) Traffic Engineering is enabled; 4) an SDN controller reoptimizes routing paths after link/node failures. FC's design makes it easy to handle route reconfiguration. FC performs virtual up-down routing. As long as the virtual layers remain unchanged (i.e., which ToR port belongs to which virtual layer), the combined set of the original paths and the post-reconfiguration paths is CBD-free. This could dramatically simplify the workflow of route reconfiguration, because any transient state during route reconfiguration is guaranteed to be deadlock-free.

In rare cases, e.g., after data center expansion, we may need to change the virtual layers because the original number of layers may not be able to support a larger-scale network. In this case, there could be a CBD in a transient state during route reconfiguration. Existing solutions on deadlock-free route reconfiguration [11, 24, 33, 39] can be applied here.

7.3 The Scalability of Routing Tables

Expander graphs, including FC, Jellyfish [46], Xpander [50], FatClique [54], etc., face a common scalability challenge in the switch routing tables. Unlike Clos, expander graphs cannot easily aggregate IP addresses in the switch routing tables due to the increased routing complexity. To resolve this challenge, one potential solution is to design a hierachically routing strategy, e.g., divide ToRs into groups based on their IP prefixes and then perform intra-group and inter-group routing separately. This approach could increase the chance of IP aggregation in the switch routing tables, but may also hurt path diversity and load balancing efficiency. We will explore this tradeoff further in our future work.

8 Conclusion

We present FC, a topology-routing co-designed methodology to eliminate PFC-induced deadlocks, for cost-effective and safe deployment of RoCEv2 over expander networks. Motivated by the fact that the up-down routing paths of multi-layered Clos networks are CBD-free, we design FC's topology to exhibit a virtual layered structure, and propose an edge-disjoint virtual up-down routing for FC that is guaranteed to be CBD-free. We evaluate FC against several competitors using throughput analysis, testbed implementation and packet-level simulation. Our evaluation results demonstrate that 1) FC is deadlock-free while ECMP/KSP may trigger deadlocks; 2) FC significantly reduces average hop count and improves network throughput over the state-of-art EDST-based routing strategy; 3) FC attains higher throughput than Clos networks built using the same number of switches under all-to-all and uniform random patterns. These properties make FC a promising design for deadlock prevention in expander graphs.

Acknowledgement: This work was supported by the NSF China (No. 61902246, 62272292 and 61960206002). We also thank our shepherd Brent Stephens and the NSDI reviewers.

References

[1] Mincostflow solver of ortools. https://developers.google.com/optimization/reference/graph/min_cost_flow.

[2] 32×400Gbps Switch Price. https://www.fs.com/products/158704.html.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[4] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen, and H. Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *SIGCOMM*, 2020.

[5] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *SC*, 2014.

[6] M. Besta, M. Schneider, M. Konieczny, K. Cynk, E. Henriksson, S. D. Girolamo, A. Singla, and T. Hoefler. Fatpaths: Routing in supercomputers and data centers when shortest paths fall short. In *SC*, 2020.

[7] B. Bollobás. The isoperimetric number of random regular graphs. *European Journal of combinatorics*, 9(3):241–244, 1988.

[8] CALIENT Technologies. https://www.calient.net/resources/#documents.

[9] P. Cao, S. Zhao, M. Y. The, Y. Liu, and X. Wang. Trod: Evolving from electrical data center to optical data center. In *ICNP*, 2021.

[10] Copper Cable Price. https://www.fs.com/products/149316.html.

[11] J.-J. Crespo, J. L. Sánchez, F. J. Alfaro-Cortés, J. Flich, and J. Duato. Upr: deadlock-free dynamic network reconfguration by exploiting channel dependency graph compatibility. *The Journal of Supercomputing*, 77:12826–12856, 2021.

[12] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. 1988.

[13] Fiber Price. https://www.fiber-mart.com/12-fibers-singlemode-smf-12-strands-flat-mtp-breakout-cable-lcscfcst-flat-fiber-cable-lszhriser-p-16935.html.

[14] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, et al. When cloud storage meets rdma. In *NSDI*, 2021.

[15] M. Gerla and L. Kleinrock. Flow control: A comparative survey. *IEEE Transactions on Communications*, 28(4):553–574, 1980.

[16] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[17] A. V. Goldberg and M. Kharitonov. On implementing scaling push-relabel algorithms. *Network Flows and Matching: First DIMACS Implementation Challenge*, 1993.

[18] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson. Backpressure flow control. In *NSDI*, 2022.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[20] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM*, 2016.

[21] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, 2017.

[22] HPCC. https://github.com/alibaba-edu/High-Precision-Congestion-Control.

[23] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016.

[24] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *CoNEXT*, 2017.

[25] IEEE. https://1.ieee802.org/dcb/802-1qbb/.

[26] S. A. Jyothi, A. Singla, P. B. Godfrey, and A. Kolla. Measuring and understanding throughput of network topologies. In *SC*, 2016.

[27] M. Karol, S. J. Golestani, and D. Lee. Prevention of deadlocks and livelocks in lossless backpressured packet networks. *IEEE/ACM Transactions on Networking*, 11(6):923–934, 2003.

[28] S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *SIGCOMM*, 2017.

[29] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *ISCA*, 2007.

[30] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, et al. Hpcc: High precision congestion control. In *SIGCOMM*, 2019.

[31] Linux Rdma. https://github.com/linux-rdma/perftest.

[32] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *NSDI*, 2013.

[33] O. Lysne, T. M. Pinkston, and J. Duato. A methodology for developing deadlock-free dynamic network reconfiguration processes. part ii. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):428–443, 2005.

[34] P. López, J. M. Martínez, and J. Duato. A very efficient distributed deadlock detection mechanism for wormhole networks. In *HPCA*, 1998.

[35] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting network support for rdma. In *SIGCOMM*, 2018.

[36] P. Namyar, S. Supittayapornpong, M. Zhang, M. Yu, and R. Govindan. A throughput-centric view of the performance of datacenter topologies. In *SIGCOMM*, 2021.

[37] Optical Transceiver Price. https://www.fs.com/products/128242.html.

[38] E. Palmer. On the spanning tree packing number of a graph: A survey. *Discrete Mathematics*, 2001.

[39] T. M. Pinkston, R. Pang, and J. Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependability. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):780–794, 2003.

[40] L. Poutievski, O. Mashayekhi, J. Ong, A. Singh, M. Tariq, R. Wang, J. Zhang, V. Beauregard, P. Conner, S. Gribble, et al. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM*, 2022.

[41] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. Technical report, RFC 5040, October, 2007.

[42] J. Roskind. *Application of Edge Disjoint Trees to Failure Recovery in Data Communication Networks*. PhD thesis, PhD thesis, Department of Electrical Engineering and Computer Science, 1983.

[43] J. Roskind and R. E. Tarjan. A note on finding minimum-cost edge-disjoint spanning trees. *Mathematics of Operations Research*, 10(4):701–708, 1985.

[44] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[45] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. 2015.

[46] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

[47] B. Stephens and A. L. Cox. Deadlock-free local fast failover for arbitrary data center networks. In *INFOCOM*, 2016.

[48] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical dcb for improved data center networks. In *INFOCOM*, 2014.

[49] M. Y. Teh, S. Zhao, P. Cao, and K. Bergman. Enabling quasi-static reconfigurable networks with robust topology engineering. *IEEE/ACM Transactions on Networking*, 2022.

[50] A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira. Xpander: Towards optimal-performance datacenters. In *CoNEXT*, 2016.

[51] X. Wu and E. T. Ng. Detecting and resolving pfc deadlocks with itsy entirely in the data plane. In *INFOCOM*, 2022.

[52] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: Automating datacenter network failure mitigation. In *SIGCOMM*, 2012.

[53] J. Y. Yen. Finding the k shortest loopless paths in a network. *management Science*, 17(11):712–716, 1971.

[54] M. Zhang, R. N. Mysore, S. Supittayapornpong, and R. Govindan. Understanding lifecycle management complexity of datacenter topologies. In *NSDI*, 2019.

[55] S. Zhao, P. Cao, and X. Wang. Understanding the performance guarantee of physical topology design for optical circuit switched data centers. In *SIGMETRICS*, 2021.

[56] S. Zhao, R. Wang, J. Zhou, J. Ong, J. C. Mogul, and A. Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *NSDI*, 2019.

[57] Y. Zhu, Y. Zhu, H. Eran, D. Firestone, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, M. Zhang, and J. Padhye. Congestion control for large-scale rdma deployments. In *SIGCOMM*, 2015.

A Appendix

A.1 Finding Edge-Disjoint Paths Using Min-Cost Max-Flow

Definition 2 *(Min-Cost Max-Flow Problem) Given a flow network $G(V,E)$ with*

- *$u(v,w)$, upper bound on flow from node v to node w;*
- *$c(v,w)$, cost of a unit of flow on (v,w),*

and a source-destination pair (s,t), $[f(v,w)]_{(v,w)\in E}$ is called a flow assignment from s to t if the following constraints are met:

1. *Capacity constraints: $0 \le f(v,w) \le u(v,w)$;*
2. *Flow conservation constraints: $\sum_u f(u,v) = \sum_w f(v,w)$ for any node $v \neq s,t$ and $\sum_w f(s,w) = \sum_u f(u,t) = F$. Here F is called the total amount of flow from s to t.*

The objective of the min-cost max-flow problem is to find a flow assignment $[f(v,w)]_{(v,w)\in E}$ with the maximum flow that minimizes

$$\sum_{(v,w)} c(v,w)\cdot f(v,w).$$

Note that the constant parameters $u(v,w)$ are all positive and $c(v,w)$ can be either positive or negative. In addition, the min-cost max-flow problem has a very nice property that guarantees integer solutions:

Theorem 6 *(Integral Flow Theorem) Given a min-cost max-flow problem, if $u(v,w)$'s are all integers, then there exists an integer solution, i.e., $f(v,w)$'s are all integers, such that $[f(v,w)]_{(v,w)\in E}$ attains the maximum flow with minimum cost.*

In fact, when we solve a min-cost max-flow problem with integer bounds using the Scaling Push-Relabel algorithm [1, 17], the resulting optimal solution is guaranteed to be an integer solution.

Finding Edge-Disjoint Paths: As a consequence of Theorem 6, we can find the maximum number of edge-disjoint paths from s to t using min-cost max-flow. Specifically, let E_0 be the set of links that can be used at most once (see the solid links in Fig. 4(c)), and $E \setminus E_0$ be the set of links that can be used multiple times (see the dashed links in Fig. 4(c)). If we set the upper bound as $u(v,w) = 1$ for all the links $(v,w) \in E_0$ and set the upper bound as $u(v,w) = \infty$ for all the links $(v,w) \in E \setminus E_0$, then the resulting min-cost max-flow solution $[f(v,w)]_{(v,w)\in E}$ can be decomposed into F (F is the maximum flow) paths where links in E_0 can be used at most once. The F paths can be found by performing Depth First Search F times (see Algorithm 1). Note that when we perform DFS in line 5 of Algorithm 1, we will never encounter a cycle. Otherwise, by removing this cycle we could obtain another flow assignment with lower cost. Having this observation could slightly simplify the DFS implementation. We do not need to track the set of visited nodes during the DFS search.

Algorithm 1: Find Edge-Disjoint Paths in the Directed Virtual Up-Down Graph

Input : A directed virtual up-down graph (see Fig. 4(c)) and a source-destination pair (s,t).
Output : Maximum number of edge-disjoint up-down paths from s to t.

1 Let E_0 be the set of solid lines in the directed virtual up-down graph. Construct a flow graph by setting the link capacity and the link cost as 1 for all links in E_0, and setting the link capacity as ∞ and the link cost as ε (an infinitesimal value) for all links not in E_0.
2 Solve the min-cost max-flow problem. Let $[f(v,w)]_{(v,w)\in E}$ be the optimal solution and let F be the maximum flow from s to t.
3 Use $\mathcal{P}$ to store the set of paths, and initialize $\mathcal{P} = \emptyset$.
4 **for** *i in {1,2,...,F}* **do**
5 Use Depth First Search to find a path P from s to t such that $f(e) \ge 1$ for every edge e in P.
6 Store P in $\mathcal{P}$.
7 For every edge e in P, decrement $f(e)$ by one.
8 **end**
9 Return $\mathcal{P}$.

A.2 A Sufficient and Necessary Condition for CBD-Free Routing

We first introduce the concept of *link dependency graph*.

Definition 3 *Given a network $G(V,E)$ and a path set $\mathcal{P} = \{P_1, P_2, ..., P_K\}$, a link dependency graph $G'(V',E')$ can be constructed as follows:*

1. *V' is the set of directed links used by at least one path $P \in \mathcal{P}$;*
2. *For any $e_1, e_2 \in V'$, there is a directed link from e_1 to e_2 in E' if and only if e_1 is the next hop of e_2 in one path $P \in \mathcal{P}$.*

Then, the following theorem offers a sufficient and necessary condition for a set of paths to be CBD-free.

Theorem 7 *Given a network $G(V,E)$, a path set $\mathcal{P} = \{P_1, P_2, ..., P_K\}$ is CBD free if and only if the corresponding link dependency graph $G'(V',E')$ contains no loops.*

Proof 2 ***Necessity $\Rightarrow$:*** *If the path set $\mathcal{P}$ is CBD free, we prove that $G'(V',E')$ contains no loops. We prove this by contradiction. Suppose that $G'(V',E')$ contains a loop $v_1' \to v_2' \to ... \to v_s' \to v_1'$. Let e_i be the link in $G(V,E)$ that corresponds to v_i'. Since e_1 is the next hop of e_2 in a path, if e_1 is paused, e_2 will be paused. Based on the same argument, $e_3, ..., e_s$ will be paused. Since e_s is the next hop of e_1 in a path, the pause of e_s will in turn pause e_1. Then, a CBD is formed, which contradicts the assumption that the path set $\mathcal{P}$ is CBD-free.*

Sufficiency $\Leftarrow$***:*** *If $G'(V',E')$ contains no loops, we prove that the path set $\mathcal{P}$ is CBD free. We again prove this by contradiction. Suppose that $\mathcal{P}$ contains a CBD. Then, there must exist a sequence of links $e_1, e_2, ..., e_s$ such that e_i is the next hop of e_{i+1} in a path and e_s is the next hop of e_1 in a path. Then, the corresponding vertices of $e_1, e_2, ..., e_s$ in $G'(V',E')$ forms a loop, which contradicts to the assumption that $G'(V',E')$ contains no loop.*

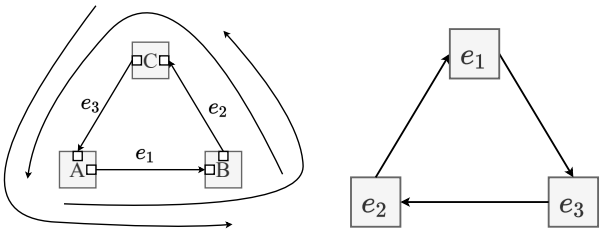
According to Theorem 7, we design Algorithm 2 to check if a set of paths is deadlock-free.

Algorithm 2: Check if a set of paths is deadlock-free

Input : A set of paths $\mathcal{P} = \{P_1, P_2, ..., P_K\}$ and a network $G(V,E)$.

Output : Whether $\mathcal{P}$ is deadlock-free.

1 Construct a link dependency graph $G'(V',E')$ based on Definition 3.
2 Use deep first search to check if $G'(V',E')$ has a loop.
3 Return true if G' has no loop; return false otherwise.



(a) An example network. (b) Link dependency graph.

Figure 11: Deadlock detection with a link dependency graph.

We use the example in Fig. 11 to illustrate the idea of the deadlock detection algorithm. Given a path set $\mathcal{P} = \{A \to B \to C, B \to C \to A, C \to A \to B\}$, we can construct a link dependency graph with three vertices: $e_1(A \to B), e_2(B \to C), e_3(C \to A)$. It is easy to see that this link dependency graph contains a loop. Thus, the path set $\mathcal{P}$ contains a CBD.

Using Theorem 7, we can prove that up-down routing is CBD-free in a multi-layered network.

Theorem 8 *In a multi-layered network, the path set generated by up-down routing is CBD-free.*

Proof 3 *For all the links in a multi-layered network, we can define a partial order as follows. A link e_1 is considered smaller than another link e_2 if either of the following three conditions is met:*

1. *e_1 is an up link while e_2 is a down link;*
2. *e_1, e_2 are down links and e_1 is at a higher layer than e_2;*
3. *e_1, e_2 are up links and e_1 is at a lower layer than e_2.*

Then, when we construct a link dependency graph based on up-down paths, it is easy to verify the following fact: if there is a directed link from e_1 to e_2, we must have $e_2 < e_1$. Therefore, the link dependency graph cannot contain a loop. As a result, the path set generated by up-down routing is CBD-free.

A.2.1 Proof of Lemma 4

Proof 4 *Since the path set $\mathcal{P}$ is CBD free in $G(V,E)$, the corresponding link dependency graph $G'(V',E')$ must contain no loop. In this case, we could construct a new graph $G''(V',E'')$ by adding a link from $v_1 \in V'$ to $v_k \in V'$ whenever there exists a sequence of node $v_2, v_3, ..., v_{k-1} \in V'$ such that $(v_i, v_{i+1}) \in E'$ for every $i = 1, 2, ..., k-1$. It is easy to check that $G''(V',E'')$ is also loop-free.*

Now we consider the contraction process. Let $\hat{G}'(\hat{V}',\hat{E}')$ be the link dependency graph of $(\hat{G}(\hat{V},\hat{E}),\hat{\mathcal{P}})$. We can prove that $\hat{G}'(\hat{V}',\hat{E}')$ is a subgraph of $G''(V',E'')$. First, in $\hat{G}(\hat{V},\hat{E})$, the edges within each vertex set V_i (i=1,2,...,m) are removed. Thus, $\hat{V}' \subseteq V'$. Second, for any edge $(e_1,e_2) \in \hat{E}'$, there must be a path $\hat{P} \in \hat{\mathcal{P}}$, such that e_1 is the next hop of e_2 in $\hat{P}$. Note that $\hat{P}$ is obtained by contracting a path $P \in \mathcal{P}$. We must have e_1 as a down-streaming hop (not necessarily next hop) of e_2 in P. Based on the construction of $G''(V',E'')$, we know that $(e_1,e_2) \in V''$. Therefore, $\hat{E}' \subseteq V''$. Based on the above analysis, we immediately know that $\hat{G}'(\hat{V}',\hat{E}')$ is a subgraph of $G''(V',E'')$. Since $G''(V',E'')$ is loop-free, $\hat{G}'(\hat{V}',\hat{E}')$ must also be loop-free. Then, according to Theorem 7, we must have that the path set $\hat{\mathcal{P}}$ is CBD free in the topology $\hat{G}(\hat{V},\hat{E})$.

A.3 Generating a Clos Network with H Hosts Using N p-Port Switches

Given N p-port switches and H hosts, we study how to construct a Clos network with the maximum throughput.

We first consider a 2-layered Clos Network. For each switch, let h be the number of ports connected to hosts. Then, the total number of switches in the first layer (i.e., the ToR layer) is $\lceil H/h \rceil$. As long as $\lceil H/h \rceil \le p$, we can put $p-h$ switches in the second layer and create a complete bipartite graph between the ToR switches and the switches in the second layer. In total, $\lceil H/h \rceil + p - h$ switches are used. To maximize throughput, we only need to find the smallest h by solving the following optimization problem:

$$\boxed{\min\ h \text{ such that } \lceil H/h \rceil \le p, \lceil H/h \rceil + p - h \le N.} \quad (5)$$

In many cases, it may not be feasible to construct a 2-layered Clos network or a 2-layered Clos network may not be throughput optimal. Hence, we also need to study how to construct a multi-layered Clos network.

We adopt a trial-and-error approach to find the throughput optimal L-layered Clos network ($L = 3, 4, ...$). Starting from $h = 1$, we try if it is possible to construct an L-layered Clos

Number of Servers	Servers per ToR		Number of Switches		Number of OCSs	Copper / Fiber Cable (km)		Number of Transceivers		Network Cost (Million $)		Bisection Bandwidth	
	FC	Clos	FC	Clos	FC	FC	Clos	FC	Clos	FC	Clos	FC	Clos
2400	8	16	300	406	24	4.8 / 158.3	4.8 / 40.6	7200	10560	27.58	29.76	1292	1280
	12	22	200	220	20	4.8 / 75.1	4.8 / 15.2	4000	4480	17.89	15.71	684	560
	16	25	150	166	16	4.8 / 40.2	4.8 / 8.7	2400	2688	12.93	11.62	396	336
4800	8	16	600	860	48	9.6 / 424.7	9.6 / 107.7	14400	19456	55.19	61.52	2526	2432
	12	21	400	482	40	9.6 / 202.4	9.6 / 48.7	8000	10560	35.80	34.70	1358	1320
	16	25	300	332	16	9.6 / 110.1	9.6 / 22.9	4800	5376	24.90	23.23	772	672
7200	8	16	900	1170	72	14.4 / 760.4	14.4 / 201.5	21600	29696	82.80	85.45	3786	3712
	12	21	600	761	40	14.4 / 361.2	14.4 / 88.4	12000	15488	52.50	54.13	2018	1936
	16	25	450	526	32	14.4 / 196.6	14.4 / 40.6	7200	8046	37.84	36.50	1156	1008
24000	8	16	3000	5500	240	48.0 / 4513.5	101.3 / 1393.5	72000	97008	276.29	383.53	12716	12288
	12	22	2000	2885	140	48.0 / 2047.1	71.6 / 493.8	40000	44298	175.53	199.58	6788	6400
	16	25	1500	2052	80	48.0 / 1102.0	62.3 / 268.8	24000	26880	124.60	140.70	3826	3584
48000	8	16	6000	12152	456	96.0 / 13571.0	227.0 / 5065.5	144000	192512	551.85	850.55	25614	24576
	12	21	4000	6691	260	96.0 / 5762.6	156.4 / 2071.1	80000	100958	350.11	465.72	13612	12672
	16	25	3000	4104	160	96.0 / 3002.8	124.7 / 940.8	48000	53760	249.32	282.75	7674	7168
72000	8	16	9000	21300	696	144.0 / 26774.9	406.1 / 11090.5	216000	288768	829.49	1471.83	38638	36864
	12	21	6000	10018	380	144.0 / 10830.7	234.1 / 4480.7	120000	151360	524.91	702.92	20626	19712
	16	25	4500	6828	240	144.0 / 5462.5	201.3 / 2021.8	72000	80640	374.11	468.66	11714	10752

Table 4: Cost Analysis: FC vs. Clos.

Switch [2]	OCS [8]	2m Copper Cable [10]	Fiber Cable [13]									Transceiver [37]		
			2m	5m	10m	15m	20m	30m	50m	100m	(100 + 50x)m	100m	500m	2000m
$ 59099	$ 60000	$ 189	$ 4.29	$ 4.71	$ 5.29	$ 5.71	$ 6.38	$ 7.38	$ 9.46	$ 16.54	$ (16.54 + 6.3x)	$ 499	$ 799	$ 1099

Table 5: Unit Price of Different Network Components.

network using at most N switches. If it is possible, we obtain the optimal h for the L-layered Clos network; otherwise, we increase h by one and retry the construction.

Starting from $L = 2$, we could use the above approach to find the best $h(L)$ for every L ($h(L) = \infty$ if it is not feasible to construct an L-layered Clos network). $h(L)$ may decrease at the beginning, but will eventually increase with respect to L. Whenever we see $h(L) < h(L+1)$, we can stop and return the minimum value of h, denoted by h^*. With h^*, the optimal throughput is $(p-h^*)/h^*$. When $h^* \leq \lfloor p/2 \rfloor$, the optimal throughput becomes larger than 1. In this case, the DCN offers abundant capacity while the access links between servers and ToRs become the bottleneck.

A.4 Network Cost Analysis

We offer a rough estimate about the total network cost for FC and Clos in this section. The network cost includes the electrical switch cost, the OCS cost and the cabling cost. Given an FC and a Clos with the same number of servers, we vary the number of servers per ToR switch and compute the number of required electrical switches and OCSs (only FC uses OCSs). To compute the cabling cost, we assume that intra-rack connections use direct attach copper cables, and inter-rack connections use optical fibers. An optical fiber requires an optical transceiver to connect to an electrical switch. The number of optical transceivers is easy to compute, which is equal to the total number of connected electrical switch ports (some switch ports may be unused) minus the total number of hosts. In contrast, the copper/fiber cable length depends on the detailed network layouts.

A.4.1 FC's Layout

In order to estimate the cable length, we make the following assumptions about FC's layout. On a data center floor, all the servers, switches and OCSs are hosted in racks. We use 2d coordinates (x,y) to represent a rack location.

- The i-th ToR switch is located at $((-1)^{\lfloor i/N_r \rfloor}(\lfloor i/(2N_r) \rfloor + 1), i\%N_r)$, where N_r is the number of racks per column;
- A rack can host four OCSs, and the i-th OCS is located at $(0, \lfloor i/4 \rfloor)$.

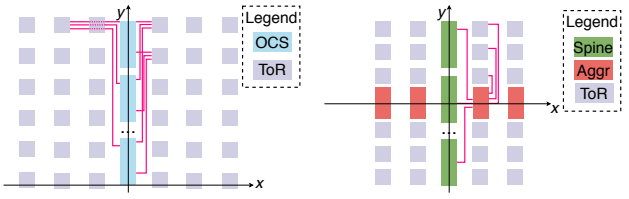
Fig. 12(a) shows FC's layout. For FC, the server-ToR connections use 2-meter copper cables and the ToR-OCS connections use fiber cables. We use Manhattan distance to compute the cable length between two racks. We vary N_r so that the total fiber cable length is minimized.

A.4.2 Clos's Layout

We focus on 3-layered and 4-layered Clos networks below. Both the 3-layered and 4-layered Clos networks follow the ToR-Aggregation-Spine architecture. In a 3-layered Clos,

Number of Switches	Number of Servers	k	Port Count of Virtual Switches	η	Cabling Constriant	Average Number of Paths	Average Path Length	Minimum Number of Paths
500	12000	4	[7, 13, 13, 7]	4	N	16.08	4.57	12.00
					Y	16.10	4.57	12.00
1000	24000	4	[7, 13, 13, 7]	7	N	14.01	4.86	9.00
					Y	14.01	4.86	9.00
2000	48000	5	[5, 10, 10, 10, 5]	13	N	15.77	5.36	11.00
					Y	15.77	5.36	11.00
3000	72000	5	[5, 10, 10, 10, 5]	19	N	14.66	5.54	10.00
					Y	14.66	5.53	10.00
5000	120000	5	[5, 10, 10, 10, 5]	32	N	13.28	5.75	9.00
					Y	13.28	5.75	9.00

Table 6: Using virtual-layered cabling has little impact on FC's routing statistics (64-port switches are used).



(a) FC cabling. (b) Clos cabling.

Figure 12: Layouts of FC and Clos. The red lines are the cable paths. To avoid visual clutter, most cable paths are omitted.

each spine is an electrical switch; in a 4-layered Clos, each spine is a 2-layered folded Clos built with electrical switches. We make the following assumptions about Clos' layout.

- The aggregation swtches in the i-th PoD are located at $((-1)^i(\lfloor i/2 \rfloor + 1), 0)$;
- The j-th ToR switch in the i-th PoD is located at $((-1)^i(\lfloor i/2 \rfloor + 1), (-1)^j(\lfloor j/2 \rfloor + 1))$;
- For a 3-layered Clos, a rack can host 24 spine switches, and the i-th spine is located at $(0, (-1)^{\lfloor i/24 \rfloor}(\lfloor (i + 24)/48 \rfloor))$. For a 4-layered Clos, we use 2 co-located racks to host a gigantic spine. Each gigantic spine is a folded Clos network, with 32 32-port switches in the first layer and 16 32-port switches in the second layer. The i-th gigantic spine is located at $(0, 2(-1)^i(\lfloor (i+1)/2 \rfloor))$ and $(0, 2(-1)^i(\lfloor (i+1)/2 \rfloor) + 1)$. Note that the intra-spine links use copper cables.

Fig. 12(b) shows Clos Network's layout. For Clos, the ToR-Aggregation and Aggregation-Spine connections use fiber cables. The Server-ToR connections use 2-meter copper cables. For 4-layered Clos, the intra-Spine connections also use 2-meter copper cables. Again, we use Manhattan distance to compute the cable length between two racks.

A.4.3 Comparison Results

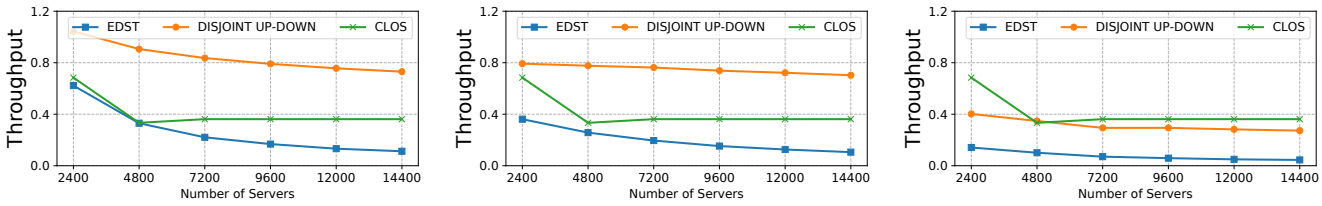
Table 4 compares the number of electrical switches/optical transceivers/OCSs and the copper/fiber cable length for FC and Clos networks. For each row, the number of servers per ToR in a Clos network has been carefully chosen such that its bisection bandwidth is equal to or slightly lower than that of FC. Generally speaking, given an FC and a Clos with the same number of hosts and similar bisection bandwidth, FC requires fewer number of electrical switches and optical transceivers, but it requires more fiber cables and additional OCSs.

Next, we offer a rough estimate on the total network cost for FC and Clos. The unit prices of different network components are summarized in Table 5. A 32×400Gbps electrical switch costs about $59000 [2] and a 320×320 optical circuit switch costs about $60000 [8]. Intra-rack connections, including server-ToR connections and intra-spine connections, use 2-meter 400Gbps copper cables, which cost about $189 [10]. The price of fiber cables increases sub-linearly with respect to the fiber length [13]. (The prices listed in [13] are the prices for 12-fiber bundles. We have divided the original price numbers by 12 in Table 5.) Hence, for each fiber cable used, we pick the shortest fiber in Table 5 that is longer than this fiber cable and use its price as the cost. The price of optical transceivers also varies depending on the transmission distance [37]. For FC, we use 2km optical transceivers because the adopted transceivers must have enough power budget to traverse an OCS. Note that traversing an OCS typically incurs about 1.5dB loss (at most 3dB) [8]. For Clos, we choose between 100m or 500m optical transceivers depending on the fiber cable length. We compare the total network cost in Table 4. When the network size is small (3-layered Clos is used), FC and Clos have similar network cost; when the network size is large (4-layered Clos is used), FC's network cost is smaller. In addition, under similar bisection bandwidth, FC uses fewer number of electrical switches and thus its network power consumption is lower (the power consumption of an OCS is only 50watts [8], which is negligible).

A.5 Additional Results

A.5.1 Impact of Cabling on FC's Routing

We generate FC's topology of different sizes & η values and evaluate if the virtual-layered cabling strategy has any impact on FC's routing. From Table 6, we cannot see clear difference



(a) Throughput of the all to all traffic matrix. (b) Throughput of uniform random traffic matrices. (c) Throughput of the near-worst traffic matrix.

Figure 13: Throughput simulation results using 64-port switches.

Number of Switches	Number of servers	k	Port Count of Virtual Switches	Routing	Average Number of Paths	Average Path Length	Average Shortest Path Length
500	12000	4	[7, 13, 13, 7]	Edge Disjoint Up-down	16.08	4.57	3.20
				EDST	20	18.34	5.26
1000	24000	4	[7, 13, 13, 7]	Edge Disjoint Up-down	14.01	4.86	3.50
				EDST	20	26.52	6.99
2000	48000	4	[7, 13, 13, 7]	Edge Disjoint Up-down	11.85	5.13	7.00
				EDST	20	33.49	9.12
3000	72000	4	[7, 13, 13, 7]	Edge Disjoint Up-down	10.63	5.30	6.00
				EDST	20	39.82	10.45

Table 7: Edge-Disjoint Virtual Up-down Routing vs. the EDST Routing (64-port Switches are Used).

when we enable/disable the cabling contraints.

A.5.2 Throughput Analysis

Clos, FC and Expander+EDST are three network architectures that are guaranteed to be deadlock-free. For networks built using 32-port switches, we demonstrate in Section 4 that

1. FC consistently outperforms Expander+EDST;
2. FC achieves higher throughput than Clos networks under all-to-all and uniform random traffic patterns.

In this section, we generate FC's topologies of different sizes using up to 600 64-port ToR switches. Each ToR switch has 40 ports connected to other switches and 24 ports connected to servers. The number of virtual layers k is chosen based on the strategy (*) in Section 3.2.3. We evaluate both FC's edge-disjoint virtual up-down routing and the EDST routing. For each FC's topology, we also compare it with a Clos network generated using roughly the same number of switches with throughput optimized. From Fig. 13, we can see that the above conclusions on FC's throughput benefits also hold for networks built using 64-port switches.

A.5.3 Routing-Path Analysis

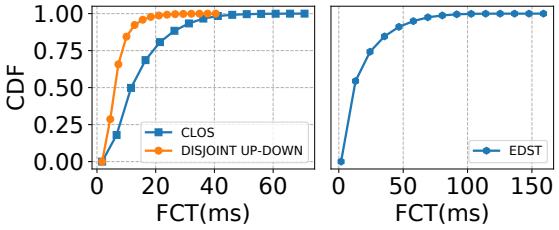
We then perform the same routing-path analysis for FC's edge-disjoint virtual up-down routing and the EDST routing. We generate FC's topologies of different sizes ($N = 500/1000/2000/3000$) using 64-port ToR switches with $s = 40$. As shown in Table 7, the average path length under FC's routing is still much shorter than that under the EDST routing.

A.5.4 More Packet-Level Simulation Results

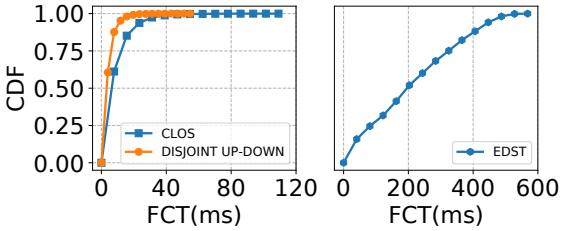
In Section 5, we perform packet-level simulation for three network setups: FC, FC+EDST, and Clos. Here, we present the detailed simulation results. We plot the CDFs for FCTs in Fig. 14. Apparently, the FCT performance under FC's routing is much better than that under the EDST routing. Hence, we mainly focus on the comparison between FC and Clos.

Under the all-to-all traffic pattern and the uniform random traffic pattern, FC achieves clearly better FCT performance than Clos because it has higher throughput. This coincides with our throughput analysis in Section 4.3.

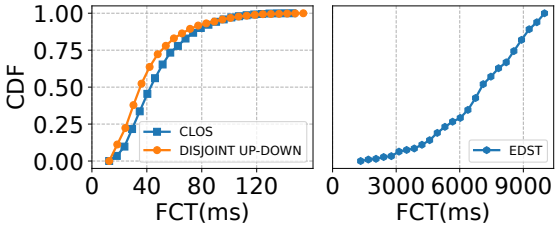
However, under the near-worst traffic pattern, we find that FC's FCT performance is just slightly worse than the Clos network's FCT performance. In contrast, our throughput analysis in Section 4.3 suggests that FC's throughput is lower than the corresponding Clos network's throughput. (In this case, FC's throughput under the near-worst pattern is about 0.5, while the Clos network's throughput is about 0.78.) The reason is that, the average hop count under FC is shorter than that under a Clos network; when the network is not congested, having a smaller average hop count compensates for the throughput gap between FC and Clos. Nevertheless, as network load increases, FC will encounter more severe congestion than Clos. We perform another simulation for the near-worst traffic pattern with network load increased from 0.3 to 0.7. (More specifically, we increase the size of each flow by $7/3$ times.) As shown in Fig. 14(d), we can see that FC's FCT performance is much worse than the Clos network's FCT performance. In fact, we see a large amount of PFC PAUSE frames in FC's network. But the good news is, there is no deadlock.



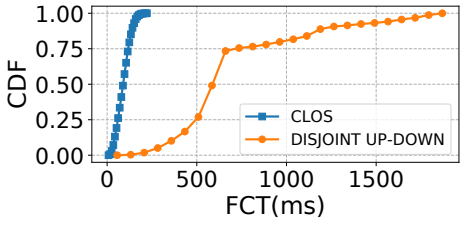
(a) All-to-all traffic pattern (Network Load= 0.3).



(b) Uniform random traffic pattern (Network Load= 0.3).



(c) Near-worst traffic pattern (Network Load= 0.3).



(d) Near-worst traffic pattern (Network Load= 0.7).

Figure 14: Compare FCTs for FC, Clos and Expander+EDST.

Scalable Tail Latency Estimation for Data Center Networks

Kevin Zhao
University of Washington

Prateesh Goyal
Microsoft Research

Mohammad Alizadeh
MIT CSAIL

Thomas E. Anderson
University of Washington

Abstract

In this paper, we consider how to provide fast estimates of flow-level tail latency performance for very large scale data center networks. Network tail latency is often a crucial metric for cloud application performance that can be affected by a wide variety of factors, including network load, inter-rack traffic skew, traffic burstiness, flow size distributions, oversubscription, and topology asymmetry. Network simulators such as ns-3 and OMNeT++ can provide accurate answers, but are very hard to parallelize, taking hours or days to answer what if questions for a single configuration at even moderate scale. Recent work with MimicNet has shown how to use machine learning to improve simulation performance, but at a cost of including a long training step per configuration, and with assumptions about workload and topology uniformity that typically do not hold in practice.

We address this gap by developing a set of techniques to provide fast performance estimates for large scale networks with general traffic matrices and topologies. A key step is to decompose the problem into a large number of parallel independent single-link simulations; we carefully combine these link-level simulations to produce accurate estimates of end-to-end flow level performance distributions for the entire network. Like MimicNet, we exploit symmetry where possible to gain additional speedups, but without relying on machine learning, so there is no training delay. On a large-scale network where ns-3 takes 11 to 27 hours to simulate five seconds of network behavior, our techniques run in one to two minutes with accuracy within 9% for tail flow completion times.

1 Introduction

Counterfactual simulation—to answer "what if" questions about the interaction of network protocols, workloads, topology, and switch behavior—has long been used by both researchers and practitioners as a way of quantifying the effect of design options and operational parameters [2, 16, 21, 23–26, 36]. As production data center networks have scaled up in bandwidth and scaled out in size [4, 29], however, network simulation has failed to keep pace. Although there is ample parallelism at a physical level in large scale data center networks, it has been difficult to realize significant speedup with packet-level network simulation [22, 30]. As packets flow through the network, the scheduling decisions at each switch affect the behavior of every flow traversing that switch, and therefore the scheduling decisions at every downstream switch, and—with congestion control—future flow behavior, in a cascading web of very fine-grained interaction. In our own experiments using ns-3 [23], for example, simulating a 384-rack, 6,144-host network on a single thread of a modern desktop CPU took 11 to 27 hours of wall-clock time to advance five seconds of simulated time. While parallel techniques for discrete event simulation exist [10], recent work has demonstrated their limited efficacy for speeding up simulations of highly interconnected data center networks [34]. As a result, packet-level network simulation today is mostly used for small scale studies.

The need for faster network simulation has spawned recent efforts to use machine learning to model how different parts of the network affect each other [32, 34]. While promising, these approaches have several limitations. MimicNet requires hours-long retraining for new workloads and network configurations, and it only accelerates simulations of uniform fat trees with uniform traffic among equally-sized clusters of machines [34]. DeepQueueNet relaxes some of MimicNet's restrictions but does not model congestion control, which can be a first-order determiner of performance [32].

This paper aims to address this gap, to develop techniques for fast approximate simulation of large scale networks with arbitrary workloads and topologies. Our work involves no training step, aiming to produce near-real time results even at scale. In addition to reducing the cost of evaluating new protocols, another goal is to provide real-time decision support for network operators, such as warnings of SLO violations if links fail [17, 20], advice on task placement of communication-intensive jobs [7], and predicting the performance impact of planned partial network outages and upgrades [8, 35].

A key observation is that we could achieve high degrees of parallelism if we could somehow disentangle the interactions between switch queues, allowing us to study the behavior of the traffic on each link in isolation. Of course, switch queues are not in reality completely disentangled. The packets for any particular flow experience a very specific set of conditions at each switch, and those conditions are affected by the presence of upstream bottlenecks which can smooth packet arrivals for competing flows at downstream switches. The congestion response for a flow depends on the combination of conditions at every switch along the path.

However, large scale data center networks are typically managed with the goal of delivering consistent high performance to applications. While congestion events do occur, they are often chaotic rather than persistent, popping up and then disappearing in different spots due to the inherent burstiness and flow size distribution of applications, rather than due to some long-term mismatch between demand and capacity in some portion of the network [33]. Further, we are often interested in *aggregate* behavior, such as the frequency of poor flow performance, rather than the behavior of each individual packet or flow.

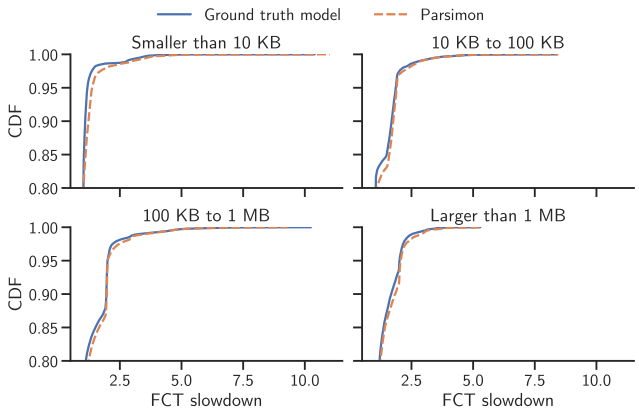


Figure 1. CDF of ns-3 versus `Parsimon` for flow completion time (FCT) slowdown across multiple flow size ranges, zoomed into the tail. While ns-3 took nearly 11 hours to produce these results, `Parsimon` took one minute and 19 seconds, end-to-end. Results were taken on a 6,144-host topology with an industry traffic matrix, 2-to-1 oversubscription, and bursty traffic.

To model aggregate behavior, our hypothesis is that we can approximate the distribution of end-to-end flow performance for a particular workload running on a large scale network by modeling the frequency and magnitude of local congestion events at each link along individual paths. A long flow will of course experience multiple congestion events during its lifetime, but most of these will occur at different points along the path *at different times*. Modeling the effect of simultaneous congestion events, and the response of the congestion algorithm to multiple simultaneous bottlenecks, is second order.

Our hypothesis is related to the concept of product-form solutions in queuing theory. For certain classes of queueing networks (e.g., Jackson [12] and BCMP networks [6]), the equilibrium distribution of queue lengths can be written in product form, i.e., the state of an individual queue is only dependent on the traffic it receives and not on the state of the rest of the network. These results generally require specific assumptions about job arrival processes (e.g., Poisson), service-time distributions (e.g., Exponential), and queueing/routing disciplines (e.g., FIFO or processor-sharing queues), and there has been much theoretical work on identifying classes of queueing networks that admit product-form solutions [13]. Although data center networks do not strictly conform to these conditions and the dynamics of each individual queue can be quite complex (e.g., due to congestion control), our hypothesis is that product-form solutions are approximately true in most realistic settings, and therefore we can analyze individual queues in isolation and combine the results to approximate end-to-end network behavior.

We built `Parsimon` to directly test this hypothesis. First, we deconstruct the network topology into a large number of simple and fast simulations where each can be run entirely in parallel by a single hyperthread. Each simulation aims to collect the distribution of delays that flows of a particular size would experience through a single link, assuming that the rest of the network is benign. We then combine these simulated delay distributions to produce predictions of the end-to-end delay distribution, again for flows of a given size. At each step, we make conservative assumptions for how delays should be computed and combined. In many settings, researchers and operators are interested in keeping tail behavior well-managed, making a conservative assumption more appropriate than an optimistic one. Finally, `Parsimon` clusters links with common traffic characteristics, eliminating much of the overhead of simulating parallel links in the core of the network as well as edge links used by replicated or parallel applications, further improving simulation performance.

Because validation against detailed packet-level simulation at scale is so expensive, we focus our study on a single widely used transport protocol, DCTCP [2], with FIFO queues with ECN packet marking at each switch [27]. We also focus on queue dynamics rather than packet loss; most data center networks are provisioned and engineered for extremely low packet loss [28, 29]. We note that these assumptions are not fundamental to our approach. We show `Parsimon` generalizes to two other transport protocols, DCQCN [36] and the delay-based TIMELY [19]. Validation of other transport protocols [3, 14, 16, 21], switch queueing disciplines [1, 9, 11, 21], and packet loss remains future work. We note that modern data center transport layer protocols are adept at quickly adapting to the presence and absence of congestion, and so we caution our results may not extend to older transport protocols where convergence time is a large factor.

Parsimon speeds up simulations by reasoning about links independently, which enables massive parallelization, but at a cost in accuracy. As we will see in §3.6, anything that creates standing congestion both at the core and at the edge, or when cross traffic is correlated across multiple hops, will result in less accurate estimates. While our methods are designed to favor overestimating rather than underestimating tail latencies, this property is only evaluated experimentally (§5). In general there is no formal guarantee, since factors like congestion control can in theory behave in arbitrary ways that render less appropriate the approximation of considering links independently. We assume that we can simulate for long enough for the network to reach equilibrium; studies of short term transient behavior should not use our approach. We do not provide predictions at the level of an individual flow, but we are able to show that `Parsimon` is accurate for sub-classes of traffic for mixed workloads. We do not attempt to model end host scheduling delay of packet processing, even though that may have a large impact on network performance [14, 15]; we leave addressing that to future work.

To assess accuracy, we compare distributions of flow completion time (FCT) slowdown, defined as the observed FCT divided by the best achievable FCT on an unloaded network, and we say a flow is complete when all of its bytes have been delivered to its destination. Fig. 1 shows a sample of our results for

the 6,144 host network mentioned above, running a published industry traffic matrix [28] and flow size distribution [21], and with standard settings for burstiness and over-provisioning. We describe the details of this and other experiments later in the paper. Depicted are FCT slowdown distributions binned by flow size. While ns-3 took nearly 11 hours on this configuration, Parsimon was able to match flow-size specific performance of ns-3 in 79 seconds (a 492 times speedup) on a single 32-way multicore server with an error of 9% at the 99th percentile. Given a small cluster of simulation servers, we estimate a completion time of 21 seconds using our approach.

In our evaluation, we scan the parameter space to identify circumstances where our approximations are less accurate. Link clustering improves performance but hurts accuracy somewhat; this tradeoff can be avoided by using more simulation cores. Without clustering, accuracy suffers when there is high utilization of links in the core (above 50%), there are high levels of oversubscription, and a large fraction of network traffic is due to flows that finish within a single round trip. Generally, a combination of factors is required for poor accuracy. In 85% of the configurations we test, the error relative to ns-3 is under 10%.

Parsimon source code and evaluation scripts are publicly available at https://github.com/netiken.

2 Parsimon Overview

This paper describes a set of methods to quickly and scalably estimate distributions of flow performance in data center networks. These techniques are implemented in a prototype called Parsimon, designed to provide the following:

- **Fast, scalable estimates.** We aim to supply estimates two to three orders of magnitude faster than full-fidelity simulation. Given enough cores, execution time should remain bounded regardless of network size.
- **Tight latency bounds, including tail performance.** Our approximations bias slightly towards overestimation, but still provide close estimates even for the 95th or 99th percentile of the distribution for a given flow length.
- **Minimal restrictions on topology and workload.** Our methods are largely independent of both topology and workload, although some combinations of topology and workload will have lower accuracy.

Fig. 2 illustrates the intuition behind its core method, and Fig. 3 depicts its workflow. The user supplies 1) a description of the topology, as a set of nodes and links, and 2) the workload, as a set of flows and routes. In our implementation, we generate the flow list by sampling from the traffic matrix and the flow size distribution, with inter-arrival times determined by a burstiness parameter. Once inputs are supplied, Parsimon proceeds in several steps:

Decomposition. To start, flows are assigned to each link they traverse, e.g., for a fat tree using ECMP. Then, for each link l, Parsimon generates a custom backend simulation with a topology selected to determine—as accurately as possible—the *contribution of* l to the end-to-end flow completion times (FCTs) of the flows passing through it. Each of these backend simulations can run in parallel.

Clustering. Depending on the size of the topology, there may be tens or hundreds of thousands (or more) of link-level simulations to perform. Fortunately, data center topologies exhibit notable symmetries, and industry has reported that the same is true for many of their workloads [28]. Parsimon can optionally cluster links with similar workloads together. Only one representative from each cluster need be simulated; the rest of the link-level simulations are pruned. Clustering is discussed in more detail in §4.2.

Simulation. The next step is to simulate all cluster representatives in parallel. The decomposition step resulted in a topology and a workload for each link-level simulation, and we can use any simulation backend. Our prototype supports two: ns-3 and a custom high-performance link-level simulator (§4.1). This allows us to directly validate our link-level simulator against ns-3. However, other efficient models, such as fluid flow [18] or machine learned models could be used here instead, for different tradeoffs of performance and accuracy. Each link-level simulation produces a distribution of the delay contributed by that link to the flow completion time (FCT), bucketed by flow size. Note this is not the link's propagation delay—we calculate that contribution directly from the topology. These distributions—described in the next section (§3)—are organized according to the original input topology, as depicted in Fig. 2. Recall that only one representative from each cluster is simulated; every other link is populated with the distributions of its cluster representative.

Aggregation. The last step is to aggregate the link-level results into estimates for entire paths through the network. These estimates are also represented as delay distributions. Conceptually, Parsimon obtains a delay distribution for a path by convolving together the appropriate distributions from each of the path's component links. Since there are multiple distributions per link and potentially many paths through the network, we do not compute convolutions up-front. Instead, convolution is done on-demand via Monte Carlo sampling; a by-product is that we can efficiently produce estimates for individual source-destination pairs, virtual networks, or classes of service (§A). To make a single point prediction for a flow taking some path through the network, Parsimon uses the flow size to find the appropriate distribution for each link, samples a value from each of them, and combines them together. This process is repeated for each flow.

At a bird's-eye view, Parsimon's method is simple: to accelerate FCT estimates, we estimate the effect of each link independently and in parallel. Then to make predictions about the whole network, we combine the results. However in our experience, the accuracy of the method hinges tightly on the

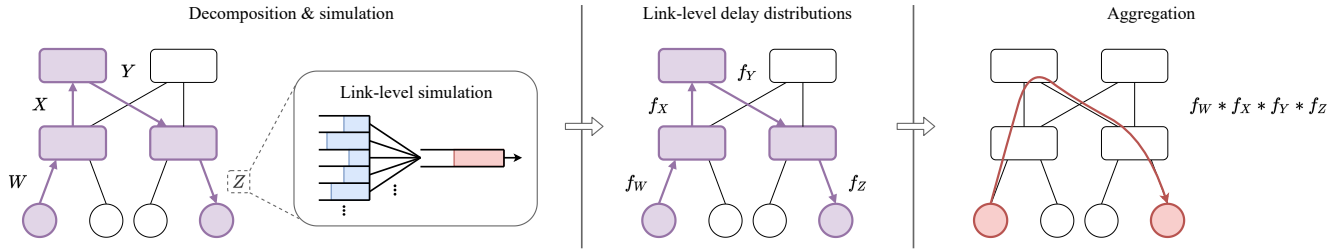


Figure 2. Overview of Parsimon. First, for any path, Parsimon estimates the contribution of each component link to delays in flow completion times, represented as a probability distribution. Parsimon then combines delays along the path using Monte Carlo simulation (see §3). Further, for added performance, link-level simulations are optimized and redundant simulations (due to e.g., ECMP or symmetries in workload patterns) are pruned (see §4).

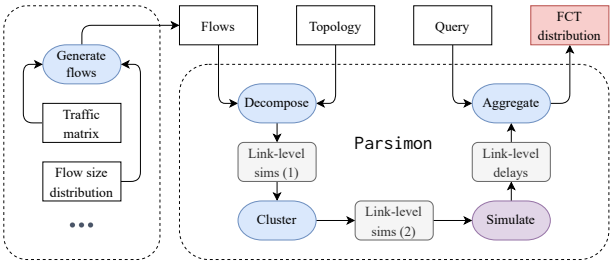


Figure 3. An illustration of Parsimon's workflow. All inputs and outputs are shown in the top row. Rectangular boxes are inputs and outputs, rounded boxes are intermediate artifacts, and ovals are Parsimon's actions.

quality of the link-level estimates and subsequent aggregation. For example, when generating the backend simulations, we have observed that failure to adequately capture pertinent features of the network severely degrades the quality of Parsimon's estimates. Similarly, link-level results must be processed and aggregated with care to preserve accuracy across all flow sizes. §3 describes these techniques in detail.

3 Key Methods: Decompose and Aggregate

Together, the methods for decomposition and aggregation are what enables Parsimon's scaling, and while we later engage additional techniques for further speed-up, they are a byproduct of—and not independent from—these more essential methods. Decisions made during this step are also the central determiners of accuracy. This section describes these processes in detail: how link-level topologies are generated, how the link-level data are post-processed and stored, and finally how they are aggregated to produce end-to-end estimates.

3.1 Generating Link-Level Workloads

To start, Parsimon associates each link with the flows passing through it. Since links are bidirectional, there are two sets of flows—and consequently two link-level simulations—per link. Parsimon populates links with flows using flows' routes. Then for each link and in each direction, the associated flows constitute the input workload to the link-level simulation. The sizes and arrival times of the flows pass though unmodified.

3.2 Generating Link-Level Topologies

Once the link-level workloads are in place, we generate the link-level topologies. In this step, we think of each link as contributing some amount of delay to end-to-end FCTs. Any given flow will accrue these delays at each hop, depending on—for example—how much bandwidth is available and how much queueing is present. Highly-loaded links are expected to contribute more delay, while rarely utilized links will contribute relatively little.

For each link and in each direction, we generate a topology and perform a simulation using just the flows traversing that link. Once the simulation is finished, the delay caused by the link for a given flow is computed by taking the observed FCT and removing the ideal FCT for that flow size. (For a flow of size s traversing a link of speed C and propagation delay l, the ideal FCT is $s/C + l$.) This intuitively captures all delays incurred due to queueing, congestion control, bandwidth sharing, and so on at the target link.

In generating a per-link topology, our goal is to isolate and measure the expected delay contribution of the target link. A simple but inefficient strategy would be to use the original topology, but with only the traffic traversing the target link, without any cross traffic. This would be relatively accurate at measuring the delay contributed by the target link, albeit a bit conservative. Upstream cross traffic congestion will slightly smooth out downstream congestion at the target link, and so removing cross traffic would make the queue distribution at the target link slightly worse than in reality.

Although relatively accurate and parallelizable, simulating every link on the original network topology would still be inefficient, as packet-level simulation cost is roughly proportional to the number of packets simulated times the number of hops each packet takes through the network. Because we run the link simulation separately in each direction on every packet that passes through that link, this would inflate the aggregate computational cost of the simulation by a multiplicative factor of roughly half the average network path length—a significant factor for large-scale networks. Instead, we want to simulate only a small constant number of hops per target link.

An extreme alternative would be to simulate only the target switch queue. This is inaccurate for two reasons. First, we

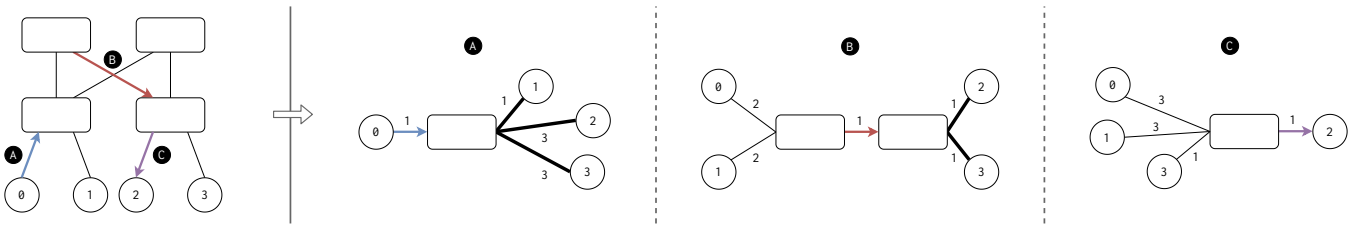


Figure 4. An illustration of how Parsimon generates link-level topologies. Simulations are unidirectional, and a different topology is used for (A) first-hop links, (B) switch-to-switch links, and (C) last-hop links. For illustration purposes, each link in the original topology has a propagation delay of one. To the left is the original topology; to the right are the corresponding link-level topologies, with new propagation delays annotated. Bold lines denote links whose bandwidths have been artificially increased during topology generation.

need to preserve end-to-end round trip delays, as these affect the speed of the congestion control adaptation to congestion or its absence; hosts closer to the target adapt faster than those farther away. Second, we need to preserve the spacing of packets induced by the original topology—a large flow does not immediately dump all of its data into the queue for the target link; instead, those packets arrive spaced apart by the edge link capacity. Ignoring this effect would lead to larger queues and more delay at the simulated link than would occur at that link in the original network.

Thus, we construct a topology for each link-level simulation that reflects a performance-accuracy tradeoff, attempting to capture the most important effects for computing the delay contributed by the target link. Fig. 4 shows how topologies are minimized. The generated topology takes one of three shapes, depending on the location and direction of the target link: (i) a first-hop up-link from a host to a ToR, (ii) a switch-to-switch link in the middle of the network, or (iii) a last-hop downlink from a ToR to a host.

Suppose the traffic through the target link originates from sources S and terminates in destinations T. In case A of Fig. 4, we connect the target link directly to each host in T via a dedicated link. If the target link is a switch-to-switch link (case B), we remove intermediate hops and connect the hosts in S directly to the input, and the output directly to the hosts in T. Lastly, if the target link is a last hop (case C), then the hosts in S are connected directly to the input. Rewriting the topology in this manner ensures that packets can traverse at most three hops, regardless of the size of the original topology.

Modeling round-trip delay. Next, we set the link delays in each constructed topology to match the round trip delays in the original network. For example, in case A of Fig. 4, the round-trip time between host 0 and host 2 is 8 in both the original topology and the generated topology, even though Parsimon has removed intermediate hops between the switch and host 2. Fig. 4 is meant as illustrative; as with ns-3, Parsimon can model arbitrary round-trip delays.

In data center networks, congestion controllers play a large role in determining the extent to which longer flows yield throughput to benefit the latency of short flows. Most algorithms such as DCTCP [2], DCQCN [36], and TIMELY [19] are *end-to-end* in the sense that sources adjust their sending rates based on feedback echoed from destinations [11]. With an end-to-end control loop, a source must wait an entire round-trip time (RTT) before being able to adapt its sending rate based on congestion feedback, resulting in longer queue lengths with higher RTTs. Thus, correctly modeling RTTs is essential to correctly modeling queue dynamics.

Selecting link bandwidths. In some cases, we artificially increase the bandwidth of downstream links to ensure that they do not artificially add congestion. We say such links are *inflated*. For example, in cases A and B of Fig. 4, the bandwidths of the last-hop links are inflated. We want any queueing to be due to the target link and not the downstream link. By inflating downstream links, we remove store and forward delay (a small packet following a large packet would otherwise need to queue for the downstream link); it also addresses the case where core links are fatter than downstream links. Queueing at the downstream link itself is accounted for in case C. By contrast, we do *not* inflate first-hop links in cases B and C, as this would enable a long flow to arrive at the target link at a higher rate than it would in practice.

A cluster of sources sending simultaneously through an oversubscribed top-of-rack (ToR) switch in the original network will be throttled beyond what is implied by the edge link capacity. To improve simulation speed, we ignore this effect and are therefore slightly conservative in our estimates for oversubscribed networks.

Correcting for ACK traffic. Since Parsimon only simulates one direction at a time, we must account for the load induced by acknowledgments due to traffic in the reverse direction. This is usually small, but can be significant at high load and where average packet size is small. Instead of modeling ACK traffic in detail, we apply a simple rule, mechanically reducing the forward bandwidth on each simulated link by the average volume consumed by ACKs for flows in the opposite direction over the course of the simulation. This correction is applied to all links but is most necessary for the target link. Note that Parsimon does not account for extra delay caused by ACK jitter on the reverse path; this could be an issue when applying our ideas to networks with bandwidth asymmetry between forward and reverse paths [5].

3.3 Post-Processing Link-Level Results

Each link-level simulation produces an FCT for each flow in the link-level workload, and these FCTs are used to compute delays. Recall from §3.2 that the delay is just the observed FCT minus the ideal FCT on an unloaded network. For each flow, we could, theoretically, estimate the end-to-end delay as some function of the delay contributed by each link for that flow. We discuss how that function works in `Parsimon`, along with its sources of bias, later in this section.

First, we address a different issue. Recall that we cluster similar links together (§4.2) so that we only simulate the flows through a single representative link for each cluster of links. Thus, to compute the end-to-end delay for a particular flow, we take a sample from the delay distributions at each hop in the path, or from the hop's standin representative.

In post-processing the link-level results and constructing these distributions, our primary objective is to support accurate estimates for *all flow sizes*. It is not enough to produce the correct FCT distribution across the entire workload; we must also accurately estimate the FCT distribution for short flows containing just a few packets as well as for long flows that last for hundreds of round trips. This extra requirement necessitates some post-processing before distributions can be constructed. Here we describe how this is done.

Packet-normalized delay. Maintaining accuracy across all flow sizes would not be possible if we used delays directly. For example, long flows, which may experience variations in their bandwidth share over time, will almost always experience more absolute delay than short flows.

As a start, we can address this by normalizing delays by flow size: after computing the delay for a particular flow, we can then divide the delay by the flow's size in packets. We call the resulting metric the *packet-normalized delay*, and it has the intuitive interpretation of summarizing the flow's average delay per packet. Link-level distributions are constructed from packet-normalized delays rather than absolute delays. We normalize by the number of packets instead of the number of bytes because flows are discretized into—and therefore delays are incurred by—packets. Further, normalizing by the number of bytes loses accuracy for small flows, especially those smaller than the maximum packet size. For example, a 10 byte packet would be delayed by the same amount as would a 100 byte packet if it arrived in the switch queue just behind a jumbo (9 KB) frame [31].

Bucketing distributions. Even with packet-normalized delays, we should still expect long flows to have different delay distributions than short flows. The FCT of a long flow is mainly determined by the throughput it achieves, while the FCT of a short flow depends on how much queueing it encounters. Further, congestion control algorithms trade the throughput of long flows for the latency of shorter ones to varying degree. An aggressive congestion control algorithm could try to keep queues near-empty [16], resulting in smaller short-flow delay and larger long-flow delay.

To ensure that estimates for different flow sizes are accurate, it is necessary to sample each packet-normalized delay from the appropriate distribution. We bucket the distribution of packet-normalized delays by flow size. Buckets need to contain enough samples to form statistically meaningful distributions, but they should also be small enough so that the values come from flows with similar delay characteristics (i.e., similarly-sized flows).

`Parsimon` uses a simple bucketing algorithm. In brief, we start with a packet-normalized delay per flow, and we sort them according to flow size. Then, starting with the shortest flow, we begin populating buckets. For each bucket b, let maxf_b and minf_b be the maximum and minimum flow sizes associated with b, respectively, and let n_b be the number of elements in b. Each bucket b apart from the last one is locally subject to two constraints

$$n_b \geq B \qquad \text{and} \qquad \text{maxf}_b \geq x * \text{minf}_b,$$

for some choice of B and x. Globally, `Parsimon` also ensures buckets are contiguous and non-overlapping. For any bucket, once the two local constraints are satisfied, `Parsimon` begins populating the next bucket, and the final bucket is assigned whatever elements remain.

In practice, we find $B = 100$ and $x = 2$ works well. Data center workloads have heavy-tailed flow size distributions in which short flows arrive much more frequently than long ones. With these parameters, the first buckets will have size boundaries that are approximately powers of two, and as flows get larger, buckets will cover larger and larger ranges. This is the desired behavior. Intuitively, a queueing-sensitive 1 KB flow should not be grouped with a throughput-sensitive 1 GB flow, but a 1 GB flow can be grouped with a 10 GB flow provided the distribution of throughput is stable on long timescales. Accuracy across different flow sizes at finer or coarser resolution can be achieved by modulating x. We examined sensitivity to the number of buckets by decreasing x for selected experiments and found no meaningful change in the predicted distributions.

In summary, each link-level simulation produces FCTs, and these FCTs are used to construct bucketed distributions of packet-normalized delay. Since different links have different workloads, bucketing is performed on a per-link basis. This means that the links in any given path are likely to have different bucket sizes with different flow size ranges. In the next subsection (§3.4) we describe how the data are aggregated.

3.4 Aggregating Link-Level Estimates

For any given range of flow sizes, the final distribution of (packet-normalized) delay for any path through the network can be estimated by selecting an appropriate distribution from each component link and then performing an n-ary convolution. However, the efficiency of this step must be considered. Since there are multiple distributions per link and potentially

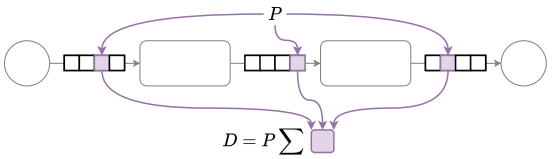


Figure 5. An illustration of how Parsimon aggregates link-level results into a path-level point estimate. Parsimon samples a packet-normalized delay (§3.3) from each link along the path, and combines these to estimate the end-to-end absolute delay D.

many paths through the network, performing all convolutions up front and storing one path-level distribution per path, per flow-size range would be costly in space and in time.

To avoid these costs, Parsimon uses an on-demand sampling strategy to perform the convolution. Recall that the simulation step resulted in bucketed distributions of packet-normalized delay per link, organized in a graph isomorphic to the original topology. Parsimon makes this graph a queryable object that is capable of supporting point estimates. Given a size, a source, and a destination, Parsimon computes a path from the source to the destination and uses the size to select a distribution per-link. Then, one packet-normalized delay is sampled from each distribution and the results are subsequently combined into a point estimate. Suppose there are n hops and let $D_1^*, D_2^*, \ldots, D_n^*$ be the sampled packet-normalized delays. Then, the end-to-end absolute delay D is computed as

$$P\sum_{i=1}^{n} D_i^* \;=\; \sum_{i=1}^{n} D_i^* P \;=\; \sum_{i=1}^{n} D_i \;=\; D,$$

where P is the input flow size in packets and D_i is the absolute delay for hop i. Fig. 5 illustrates this process. Finally, to obtain a distribution of end-to-end delay estimates, we need only sample enough point estimates for the desired flow size range and source destination pairs.

3.5 Primary Source of Speedup

Parsimon speeds up large network simulations by considering the effect of each link in isolation, allowing it to scale in the size of the simulated network and the number of processing cores. Although the link is the unit of decomposition, Parsimon's scaling ability is determined not by the total the number of links, but rather by the *fraction of total packets traversing any link*. In other words, Parsimon's speed-up depends on the number of busy links and how well the load is balanced among them. This explains why Parsimon is most suited for large data center networks, where the total workload comprises many source destination pairs with many paths between them. If a network traffic is heavily skewed such that most of the workload traverses only a few paths, the amount of speedup will be limited.

3.6 Primary Sources of Error

To balance accuracy and performance, Parsimon makes a number of approximations, with some having more of an effect on accuracy than others. Here we catalog some of the main sources of error, describing 1) how we expect the errors to manifest and 2) what modifications, if any, could be made to address them.

Bottleneck fan-in. To simulate a given target link in the network, Parsimon constructs a topology that connects all of the source nodes feeding traffic directly into that target. In practice, of course, there would be multiple stages of fan-in, and that fan-in would tend to spread out any burst of arriving flows due to upstream bandwidth capacity constraints. Any target link would experience slightly less queueing and less congestion in reality than in Parsimon. Of course, Parsimon also simulates the upstream link; because it is closer to the sources, its traffic and queueing behavior would be a closer model to what would happen in a full network-wide simulation.

Because Parsimon sums the delay contributed by each hop along a flow's path, the lack of fan-in will tend to slightly overestimate the delays caused by downstream links. Put another way, any delay induced by fan-in constraints is counted twice—once when we simulate the upstream link and again when we simulate the downstream link. In our evaluation, accuracy is slightly lower for networks with higher degrees of oversubscription, as we would expect. We could potentially remove this inaccuracy by including the upstream fan-in as part of the topology for each link simulation. Since simulation time is proportional to the number of hops, this would decrease individual link simulation efficiency by a small but significant factor.

Lack of traffic smoothing. Similarly, any cross-traffic that shares a portion of a path with traffic destined for the target link will tend to smooth out traffic before it reaches the target. Parsimon does not include any cross-traffic in its per-link simulation, making it slightly overestimate the queueing delay at the target link. Assuming the simulation is stable—that the arrival rate does not exceed the service rate for any link—the target link will experience the correct long-term average rate, but without as much smoothing as would happen in practice. We see evidence of this effect in our evaluation, where error is slightly larger for workloads with a predominance of short flows which would benefit more from smoothing. Of course, correctly modeling the effect of cross-traffic on the traffic arriving at a downstream link would be difficult to accomplish without reverting to a full network simulation.

Link-level independence. A more fundamental approximation is that link-level simulations are treated independently. This technique enables wholesale parallelization, but its accuracy depends on the amount of correlation between the traffic intensities on the various hops along the path. The more correlated the traffic, the more error Parsimon's method produces.

Since Parsimon produces estimates by convolving delay distributions (adding independent random variables), full accuracy requires the mutual independence of delays among the links in every path. Consider a single-packet flow that traverses two hops, both with load l. If the delays along the two hops are independent, the probability that the flow will encounter *no queueing* is simply $(1-l)^2$. However, if both

hops tend to have queueing at the same time (i.e., if the traffic intensities and therefore the delays are correlated), then that probability is closer to $1 - l$. Since Parsimon does not distinguish between these two scenarios, the difference is not reflected in its estimates.

In very large networks with thousands of hosts and paths, and with realistic workloads, we expect the effects of correlation to be small. A basic result of queueing theory is that under some circumstances it is possible to analyze queues independently, even when the output of one queue connects to the input of another, so that queue behaviors are obviously correlated. One view of our work is that we are empirically observing that data center networks approximately admit product-form solutions for their equilibrium state queue distributions under realistic workloads.

However, some networks use PFC [36] to reduce packet loss due to go-back N error handling in some RDMA network interface cards. Because PFC suffers from head-of-line blocking, PFC can cause correlated congestion across multiple links, and so Parsimon would not be a good choice for modeling such networks. If correlation is a problem, we could potentially measure the degree of correlation and apply a correcting factor during the convolution step, but we leave that for future work.

One bottleneck at a time. Estimating the performance of long flows comes with an additional difficulty which is also exacerbated by correlated delays. While a single packet flow can only reside in one queue at a time, a long flow can be backlogged on multiple links *at the same time.* Depending on the specific congestion control mechanism, the throttling back of a long flow (the delay it experiences) is typically *not* the sum of the delays it would experience on individual links (as Parsimon approximates), but rather only the delay caused by the true (instantaneous) bottleneck. Since Parsimon sums all delays, it will overestimate the end-to-end delay for the long flow that encounters simultaneous cross-traffic congestion at multiple points along its path. In summary, Parsimon is more accurate when the congestion is episodic and temporary, appearing at different links at different times, and less accurate when congestion is persistent across multiple edge and core links of a given path.

Congestion on any link (and therefore simultaneous congestion on multiple links) becomes more common with higher network load, and we see this effect in our evaluation. We can potentially correct for this bias by using a more complex function for combining link delays when overall network utilization is high. Because network operators are often willing to over-provision their network hardware to reduce application tail latency, this is rare in practice. For example, some recent end-to-end congestion protocols, such as Homa [21], simply assume that network congestion predominantly occurs at the last hop of each path. We do not make such an assumption; we handle congestion equally wherever it might occur. However, we do assume that congestion events are not persistent and network wide.

Our approximations are biased toward producing overestimates rather than underestimates, because we expect network operators to be more sensitive to over-promising tail behavior, even if that comes at the cost of being too conservative with respect to capacity planning. Additional analyses on the errors induced by these approximations can be found in the appendix (§C).

4 Complementary Methods

The previous section described how we decompose a single large network simulation into many small, independent ones that can be executed in parallel and later combined. This section describes additional optimizations that reduce, cluster, and prune these link-level simulations for better computational efficiency. These reduce the number of cores needed to simulate a given network within some time bound, or equivalently, the execution time on a single server machine.

4.1 Fast Link-Level Simulation

By far the largest computational cost in Parsimon are the link-level simulations. Initially we used ns-3 as our link-level backend. However, as a general-purpose simulator, ns-3 is designed to support arbitrary protocols with arbitrary extensions, all the way down to hardware models. This is more flexible but means that every packet in ns-3 generates events at every host, queue, and link—as well as throughout the hosts' modeled network stacks.

Instead, we implemented a custom and minimal simulator optimized for high fidelity single link simulation. This backend only models the workload, topology, queueing, and congestion control. For congestion control, our prototype implements DCTCP's core algorithm [2] in a few tens of lines of code. For example, we do not need to model the mechanism for carrying ECN bits from switches back to endpoints. Switching to a custom simulator speeds up the individual link simulations by roughly an order of magnitude, with negligible loss of accuracy. Reducing the simulation time of the worst case (most congested) link also reduces the critical path dramatically. If more simulation features are needed, Parsimon can use ns-3 at the cost of using more cores.

4.2 Clustering and Pruning Simulations

Lastly, we recall that Parsimon's decomposition results in two simulations per link: one in each direction (§3.1). On a large-scale 6,144-host topology we use for evaluation, there are over 9,000 links, and therefore over 18,000 simulations generated. Fortunately, data center topologies commonly induce symmetries that render some of these simulations redundant. For example, up-links in the same ECMP grouping can be assumed to have the same characteristics and traffic patterns. Furthermore, the workloads themselves may also induce symmetries due to communication patterns and load balancing [28].

We can take advantage of these symmetries by clustering links that carry similar traffic and only simulating one representative from each cluster. Then, in each cluster, all links

Algorithm 1 Greedy link clustering

```
unclustered ← AllLinks                          ▷ links here are unidirectional
clusters ← []                                   ▷ list of list of links
while not Empty(unclustered) do
    members ← []                                ▷ new cluster
    representative ← PopFirst(unclustered)
    Push(members, representative)               ▷ with initial member
    for candidate in unclustered do             ▷ find other members
        rfeature ← Feature(representative)
        cfeature ← Feature(candidate)
        if IsCloseEnough(rfeature, cfeature) then
            Push(members, candidate)            ▷ new member
            Remove(unclustered, candidate)
    Push(clusters, members)
return clusters
```

inherit the delay distribution produced by the representative link. Parsimon's clustering requirement is quite specific, which limits the range of popular clustering algorithms that can be used. Let $l_1, l_2 \in L$ be any two link-level simulations, and let $d : L \times L \to \mathbb{R}$ be a distance function. Ideally,

$$l_1 \text{ and } l_2 \text{ are clustered together} \iff d(l_1, l_2) < \epsilon,$$

where ϵ is some bound. The left-to-right direction preserves accuracy; the right-to-left supports efficiency. Most centroid-based and density-based clustering algorithms aren't designed to provide the left-to-right property. Instead, Parsimon uses Alg. 1. This algorithm greedily clusters simulations together, using a distance function that predicts which links will have similar delay profiles. In our prototype, we check that the link flow size and inter-arrival time distributions—as well as their load levels—are close. We find this provides a reasonable tradeoff between efficiency and accuracy, but users can turn off the optimization at the cost of using more cores. Further details about the clustering can be found in the appendix (§D).

5 Evaluation

Parsimon's goal is to quickly estimate tail latencies for a variety of large data center networks and workloads. In evaluating Parsimon, we would like to assess 1) Parsimon's accuracy and performance at the scale of thousands of hosts, and 2) how accuracy is affected by a wide range of variables over the workload and the topology.

Our strategy is as follows. Using workloads extracted from industry datasets, we start with a 384-rack, 6144-host topology to evaluate Parsimon's speed and accuracy in one scenario at scale. Then, to evaluate nearly 200 other topology and workload scenarios, we downsample the workload so that it can run on a smaller 256-host topology. This allows us to run enough ns-3 simulations quickly enough to perform a detailed sensitivity analysis.

To more clearly illustrate sources of error in Parsimon, we also construct and evaluate Parsimon on synthetic workloads on a small-scale parking lot topology in Appendix §C.

Variant	Clustering?	Link-level backend
Parsimon	No	custom
Parsimon/C	Yes	custom
Parsimon/ns-3	No	ns-3
Parsimon/inf	—	custom

Table 1. The Parsimon variants under consideration. Parsimon/inf is a variant that assumes infinite cores and memory.

5.1 General Setup

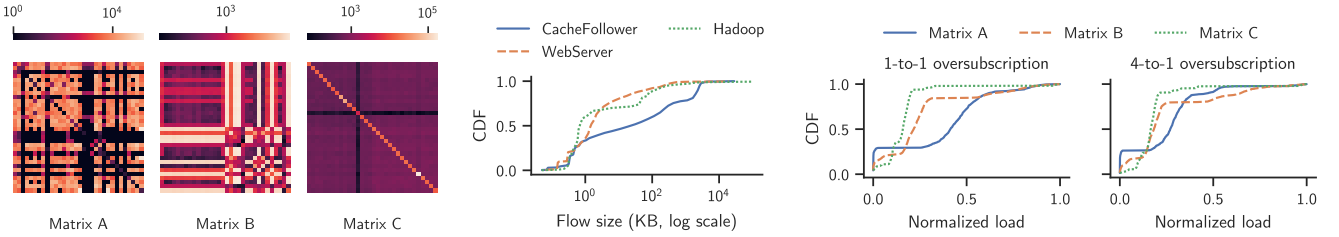
Each scenario we consider has six components: 1) a topology size, 2) an oversubscription factor, 3) a traffic matrix, 4) a flow size distribution, 5) a burstiness level, and 6) a maximum load level. Here, we briefly describe how these are specified and configured. We also discuss which Parsimon variants we will assess and how we establish a baseline.

Topology and oversubscription. To mimic an industry topology, our topologies are modeled after Meta's data center fabric [4]. In brief, there are three layers of switches: hosts connected to a top-of-rack switch (ToR) with 10 Gbps links constitute a *rack*, racks connected to each other via fabric switches with 40 Gbps links constitute a *pod*, and pods connected to each other via spine switches with 40 Gbps links constitute a *cluster*. Spine switches are organized in *planes*. We can modulate the size of a topology (corresponding to a cluster) by adjusting the number of pods, the number of racks per pod, and the number of hosts per rack, and we can modulate the oversubscription factor by adjusting the number of spines per plane.

Traffic matrices. The traffic matrices are extracted from the datasets accompanying Roy *et al.*'s study of Meta's data center network [28]. The data only allow us to construct reliable rack-to-rack matrices. When sampling workloads, we use the matrices to generate rack-to-rack traffic, but once a rack is chosen, we select its hosts uniformly at random. This may bear semblance to reality: according to Roy *et al.*, Meta's racks typically only contain servers in the same role, and load balancing is used pervasively. We use traffic matrices from three different clusters: a database cluster (matrix A), a web server cluster (matrix B), and a Hadoop cluster (matrix C). Fig. 6a shows 32-rack samples of the matrices.

Flow sizes and burstiness. We use three flow size distributions, estimated from published data in Roy *et al.*'s study [28]. These are reproduced in Fig. 6b. For inter-arrival times, we use the log-normal distribution to model bursty traffic, and we modulate the burstiness by adjusting the log-normal shape parameter σ. For low burstiness, we select $\sigma = 1$, and for high burstiness, we choose $\sigma = 2$.

Maximum load level. When setting a load level, we ensure that the offered rate is less than the link capacity for each link by specifying the maximum load level that any link can have. Note that a given maximum load level may result in different link load distributions, depending on the traffic matrix and the topology. Fig. 6c shows the distribution of normalized link



(a) Traffic matrices (32-rack sample) **(b)** Flow size distributions **(c)** Normalized link load distributions

Figure 6. In the evaluation, we model workloads using data from Roy *et al.*'s study of Meta's data center network [28]. The traffic matrices in Fig. 6a are extracted from the accompanying dataset, and the flow size distributions in Fig. 6b are estimated from the published data. Lastly, for a given topology, the distribution of link loads depends on 1) the traffic matrix and 2) the degree of oversubscription. Fig. 6c shows the link loads induced by the matrices in Fig. 6a on two 32-rack topologies with different overprovisioning. The x-axis is normalized to the maximum link load.

Estimator	Time	Speed-up
ns-3	10h 48m 26s	—
Parsimon	4m 13s	154×
Parsimon/C	1m 19s	492×
Parsimon/inf	21s	1864×

Table 2. Running times and speed-up of Parsimon variants for five seconds of simulated time on a large oversubscribed network with thousands of hosts. We find that Parsimon estimates latencies orders of magnitude faster than does ns-3. If there is ample opportunity for clustering or if there are infinite compute resources, speed-up is substantially further increased. Measurements were taken on a 32-core machine.

loads on a 32-rack topology with the traffic matrices in Fig. 6a and two different oversubscription factors. When describing how loaded a topology is, we will usually specify the average load of the top 10% most loaded links.

Parsimon variants and baseline. To establish a baseline for Parsimon's accuracy and performance, we use ns-3 with the optimized build profile. We also consider several Parsimon variants, summarized in Table 1. By default, Parsimon uses the custom link-level backend (§4.1) with clustering turned off. This expresses a lower bound on Parsimon's expected speed-up given a particular machine. Parsimon/C adds clustering to the default variant using the methods described at the end of §4.2, and Parsimon/ns-3 replaces the default's custom backend with ns-3. Lastly, Parsimon/inf provides an estimate of Parsimon's performance given infinite cores and infinite memory, computed by adding the run time of the longest link-level simulation to the fixed costs of network setup and convolution sampling. This represents an upper bound on the Parsimon's achievable performance. All performance measurements are taken on a 32-core AMD Ryzen Threadripper 3970X.

5.2 Analysis on a Large-Scale Network

Here we evaluate Parsimon's accuracy and performance on a 384-rack, 6144-host topology. The topology has eight pods, 48 racks per pod, and 16 hosts per rack, with 2-to-1 oversubscription. For the workload, we use matrix B, the WebServer flow size distribution, and high burstiness ($\sigma = 2$). We set a maximum link load of about 50%, which gives the 100 most loaded links an average load of 32%, and the top 10% most loaded links an average load of about 15%. We configure all simulations to run for five seconds of simulated time. To establish a baseline, we first run the scenario in ns-3, then we run the scenario in Parsimon and Parsimon/C (see Table 1). Due to memory constraints we omit Parsimon/ns-3 here, but we include its analysis at smaller scale in §5.3.

Fig. 7 shows the accuracy of Parsimon relative to ns-3 across four flow size bins. We find that across all bins, both variants accurately estimate tail latencies. If we consider all flow sizes together, we find that Parsimon and Parsimon/C overestimate the p99 FCT slowdown by 8.8% and 7.5%, respectively.

Table 2 shows the running time and speed-up for each estimator, which includes topology generation and convolution sampling overheads where applicable. While ns-3 took nearly 11 hours, Parsimon without clustering took four minutes and 13 seconds, for a speed-up of 154×. If we turn clustering on by using Parsimon/C, the running time is further reduced to one minute and 19 seconds, for a speed-up of 492×. [1] In this case, only 25% of links were simulated; the rest were pruned. Lastly, Parsimon/inf estimates Parsimon's best possible performance given infinite compute resources. The longest-running single-link simulation took 11 seconds, and with the additional 10 seconds required for network setup and convolution sampling, the fastest projected running time is 21 seconds.

We chose an oversubscribed topology to slightly disadvantage Parsimon's method, as oversubscription can lower Parsimon's accuracy. §5.3 analyzes the effect of oversubscription in more detail. We also ran the above experiment on a topology without oversubscription, which for the same maximum load setting increased the top 10% average link load from 15% to 25%. We found Parsimon's p99 accuracy improved from 9% to about 7%, while Parsimon/C's accuracy remained

[1] We advise caution both in interpreting this number and in generalizing it to scenarios at large. While our workloads are modeled after industry data, they are still synthetic. There may be more or less opportunity to cluster and prune link-level simulations, depending on the structure of real workloads and the quality of the clustering algorithm.

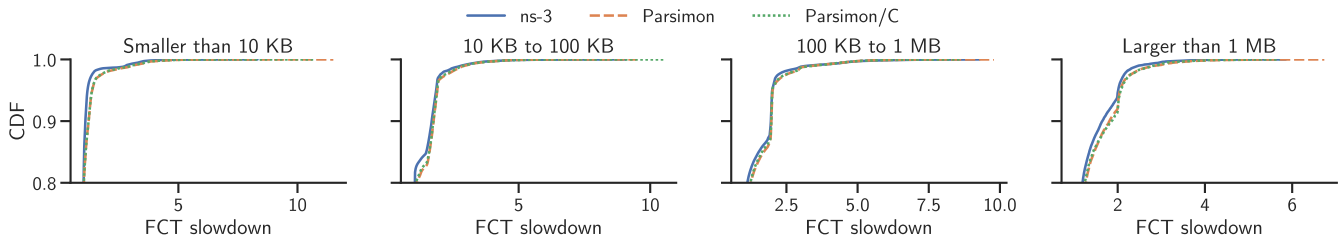


Figure 7. CDFs of FCT slowdown estimated by ns-3 and two Parsimon variants (note the y-axis). On a large network with 6,144 hosts, an industry traffic matrix (matrix B), and 2-to-1 oversubscription in the core, Parsimon's latency estimates are similar to those produced by full-fidelity simulation. Table 2 shows the performance of each estimator.

Parameter	Sample space
Oversubscription	1-to-1, 2-to-1, 4-to-1
Traffic matrix	Matrix A, Matrix B, Matrix C
Flow size distribution	CacheFollower, WebServer, Hadoop
Burstiness	Low ($\sigma = 1$), High ($\sigma = 2$)
Max load	26% to 83% (continuous range)

Table 3. The sample space for the sensitivity analysis in §5.3.

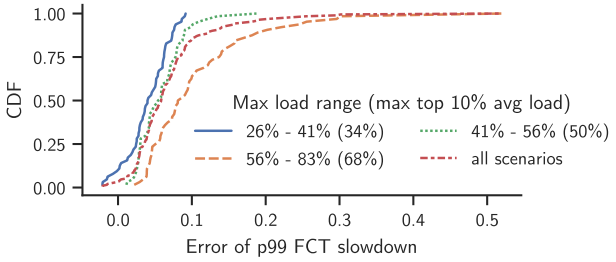


Figure 8. CDFs of p99 error between Parsimon and ns-3 across all scenarios drawn from the sample space in Table 3. The distributions are binned by maximum load. In parentheses, we give the maximum value for the top 10% average load in each bin. Under common conditions of low to moderate load, Parsimon's estimates for the p99 FCT slowdown are reliably within 10% of the ground truth.

approximately the same. However, because aggregate load increased, ns-3 took 27 hours for five seconds of simulated time, and speed-ups for Parsimon, Parsimon/C and Parsimon/inf were 152×, 872×, and 3487×, respectively. Parsimon/C benefited from the increased number of links in each ECMP grouping, allowing it to prune 85% of the link-level simulations.

5.3 Sensitivity Analysis at Small Scale

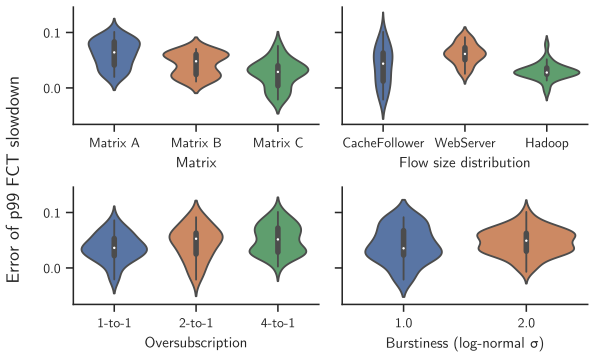
Next we turn our attention to how different aspects of workloads and topologies affect Parsimon's accuracy. To be able to simulate enough scenarios in ns-3 for a sensitivity analysis, we downsample the topologies and traffic matrices to 32 racks. The resulting topologies have two pods, 16 racks per pod, and eight hosts per rack, and the number of spines per plane varies to accommodate different oversubscription factors.

Our approach is as follows. First, we construct a sample space over the parameters defining the workload and the topology (aside from the number of servers, which is fixed). The sample space is shown in Table 3. Then, we sample 192 scenarios uniformly at random, and we run ns-3 and the default Parsimon variant on each of them for several seconds of simulated time. Next, for each scenario, we take the p99 FCT slowdown estimated by both ns-3 and Parsimon, and we compute the error between them. If these values are n and p respectively, then the error is $(p - n)/n$. Negative values indicate that Parsimon produced an underestimate.

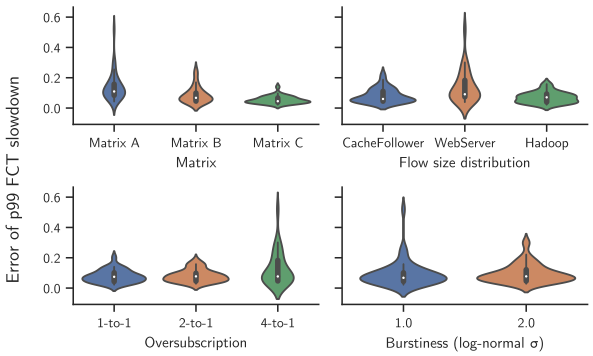
Since we have one error value per scenario, the errors give rise to distributions of error associated with the original sample space. Now what remains is to determine how the workload and topology parameters affect error distributions. To start, recall from the discussion in §3.6 that the magnitude of error is expected to be load-dependent, with higher errors typically manifesting at higher loads, so we begin by examining the effect of the maximum load setting on Parsimon's accuracy.

Maximum load. Fig. 8 shows the error distributions binned by maximum load. Among all scenarios, Parsimon's p99 estimates are within 10% of ns-3's estimates 85% of the time At high load, we observe larger overestimates of up to 52% in the worst case. In the most highly-loaded group of scenarios—with maximum link loads between 56% and 83%—Parsimon is within 10% of ns-3 62% of the time, with an average error of about 11%. However, this includes scenarios where 10% of the links have an average load of up to 68%, which is much higher than what is reported in the literature. For example, Roy *et al.* report that in Meta's data center network, 99% of host links are less than 10% loaded, and the top 5% of core links have loads between 23% and 46% [28]. Among scenarios where the maximum link load is between 26% and 41%, Parsimon is within 10% of ns-3 100% of the time. If we further include scenarios with maximum link loads between 41% and 56%, that number falls to 96%. Finally, while Parsimon's techniques tend to overestimate latencies, in 3% of the scenarios, Parsimon underestimates p99 slowdown by up to 2%.

Other parameters. We next turn to the effects of all other workload and topology parameters. We start by only considering scenarios where the maximum link load is less than or equal to 50%; this will tell us whether any of the parameters



(a) Max load ≤ 50%



(b) Max load > 50%

Figure 9. Distributions of p99 error between `Parsimon` and ns-3, faceted by different workload and topology parameters. For each distribution we show the median, the quartiles, and a rotated kernel density estimation. We consider the low-load regime (Fig. 9a) and the high-load regime (Fig. 9b) separately. At low load, the workload and topology parameters only have a modest effect on `Parsimon`'s accuracy, but at high load, the conditions leading to the largest errors come into view: high load, high oversubscription, with very short flows. Note the different y-axes between the two load regimes.

Error	Max load	Matrix	Sizes	Oversub	σ
51.9%	77.6%	A	WebServer	4-to-1	1
30.1%	67.3%	A	WebServer	4-to-1	2
29.6%	67.0%	A	WebServer	4-to-1	2
25.6%	65.9%	A	WebServer	4-to-1	1
24.6%	73.2%	B	WebServer	4-to-1	1

Table 4. The five scenarios with the highest error values from the sensitivity analysis in §5.3.

have a large effect on accuracy in the low-load regime. Fig. 9a shows the median error and error distributions as a violin plot for low-load scenarios grouped by traffic matrix, flow size distribution, oversubscription, and burstiness. Overall, changes to these parameters appear only to have a modest effect. The choice of traffic matrix has the clearest trend, but load is a confounder here: recall from Fig. 6c that different traffic matrices yield different link load distributions for the same maximum load setting.

When we look at the high load regime in Fig. 9b, a clear picture comes into view. We see much longer tails in error distributions for matrix A, the WebServer flow size distribution, and 4-to-1 oversubscription. Together with Fig. 9a, this suggests that none of these settings has a strong effect on its own, but *coupled together in the high load regime,* they have a pronounced effect on `Parsimon`'s accuracy. Matrix A induces higher average load and has more cross-rack traffic, making it more likely for its flows to encounter multiple simultaneous bottlenecks. The WebServer flow size distribution is dominated by short flows (Fig. 6b), a third of which are smaller than 1 KB and 80% of which are smaller than 10 KB. Because more of the traffic completes within a single round trip, there is more ephemeral congestion and bandwidth smoothing can have a larger impact.

Finally, oversubscription has an effect at high load: if we removed all scenarios with 4-to-1 oversubscription, the maximum error would only be 20% rather than 52%, even at high load. In addition to the double counting of delays described in §3.6, oversubscription can also increase correlations in link delays. To achieve 4-to-1 oversubscription in topologies as small as these, there are only four spine switches per plane forwarding traffic between groups of 16 racks, leaving relatively few paths through the core. Fewer paths can result in higher degrees of correlation—especially with matrix A, whose traffic is primarily inter-rack (Fig. 6a). Finally, this setting combined with the short flows from the WebServer distributions gives rise to errors of up to 52%.

Table 4 lists the scenarios with the top five highest error values. Four have matrix A, all have the WebServer distribution, and all five have 4-to-1 oversubscription. In this group, the average maximum load is 70.2%. Since we expect the combination of all-to-all workload, heavily oversubscribed topology, and persistently high core utilization to occur relatively infrequently, the data suggest that `Parsimon` maintains good accuracy under common conditions.

Mixed Workloads. We also use the small topology to study the `Parsimon` prediction error for subsets of traffic in heterogeneous workloads in Appendix §A.

5.4 Analysis of One Configuration

We pick one representative scenario to examine in more detail, to test if our approach is robust to alternate definitions of tail latency, congestion control protocol, workload, and topology. To pick a scenario whose accuracy is somewhat worse than the average case, we rank-order all scenarios by error and select the one at the 85th percentile. This has matrix A, the Hadoop flow size distribution, low burstiness, 2-to-1 oversubscription, and a maximum load of 68% (with a top 10% average load of 56%).

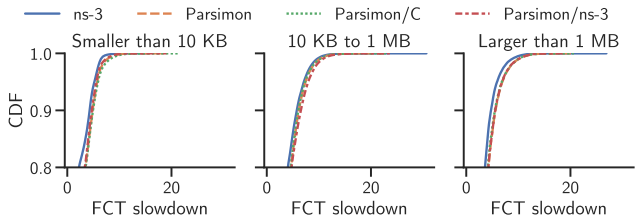


Figure 10. CDFs of FCT slowdown estimated by ns-3 and Parsimon for the scenario whose error is at the 85th percentile of the p99 error distribution. Note the y-axis. Even though the accuracy here is worse than in the common case, Parsimon's estimates remain close across most of the tail. Also shown is Parsimon/ns-3.

Protocol	Max load	Error in p99 FCT slowdown		
		< 10 KB	10 KB - 1 MB	> 1 MB
DCTCP	45%	1.4%	9.2%	15.9%
TIMELY	45%	4.0%	17.9%	13.7%
DCQCN	45%	5.9%	11.6%	12.8%
DCTCP	56%	2.8%	9.2%	14.6%
TIMELY	56%	8.1%	20.0%	11.3%
DCQCN	56%	7.6%	14.6%	12.2%
DCTCP	67%	13.8%	11.3%	13.6%
TIMELY	67%	13.3%	18.2%	5.0%
DCQCN	67%	18.0%	15.2%	13.6%

Table 5. Prediction error of Parsimon/ns-3 for estimated p99 FCT slowdown with three different congestion control protocols for the sample configuration at different load levels and for different request sizes.

Tail distribution. Operators may differ in their definitions of tail latency, e.g., focusing on the 90th or 99.9th percentile, rather than just the 99th FCT slowdown. Fig. 10 shows the tail of the cumulative distribution of FCT slowdown for different flow sizes for the selected configuration, for ns-3 and each of the Parsimon variants. The prediction error is similar across the tail of the distribution for this scenario, with little accuracy difference between any of the variants.

Transport protocols. We use the sample scenario to test the generality of Parsimon to two additional congestion control protocols, DCQCN [36] and TIMELY [19]. DCQCN is designed for RDMA traffic, while TIMELY uses network delay, rather than ECN signals, to detect congestion. To focus on prediction error for our approximation methods, we use the pre-existing ns-3 implementation of the protocols as the Parsimon link level simulator for this part of the evaluation. Note that Parsimon and Parsimon/ns-3 exhibit a few percent difference in p99 error for DCTCP for this configuration. Because the prediction error for different congestion control protocols may depend on the amount of congestion, we also run the experiment at varying load levels.

Table 5 shows the prediction error for Parsimon/ns-3 relative to ns-3 in the estimated p99 FCT slowdown at three load levels for the three transport protocols, aggregated by request size. For this configuration, Parsimon is most accurate for small flows and low to moderate maximum link utilization, and that is true for all three congestion control protocols. DCTCP has somewhat lower error for small and medium size flows at low to moderate utilization. Relative error is higher for larger transfers and higher maximum link utilization, with no clear pattern in the error for different protocols.

Simulated link failures. We also use the sample configuration to examine the prediction accuracy for topologies with simulated link failures in Appendix §B.

6 Conclusion

In this paper, we propose and evaluate a new method for computing a conservative estimate of flow-level tail latency for large scale data center networks, given an arbitrary traffic matrix, topology, flow size distribution, and inter-arrival process. Our approach decomposes the problem into a large number of individual link simulations, specially constructed to produce accurate estimates of the probability distribution of delay contributed by congestion at each link. We then mechanically combine these link-level delay distributions to produce flow-level estimates. On a large-scale network using a commercial workload, our approach outperforms ns-3 by a factor of 492 on a single multicore server with a loss of accuracy of less than 9% in the tail of the latency distribution.

Acknowledgments. We are grateful to Vincent Liu, Jeff Mogul, our shepherd Arpit Gupta, and the anonymous reviewers for their feedback and useful comments. This work was supported in part by NSF grants CNS-2006346, CNS-2006827, a Cisco Research Center Award, and a Google Research Award.

References

[1] A. G. Alcoz, A. Dietmüller, and L. Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, 2020.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[3] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, 2012.

[4] A. Andreyev. Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[5] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The Effects of Asymmetry on TCP Performance. *Mobile*

Networks and Applications, 4(3):219–241, 1999.

[6] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *Journal of the ACM (JACM)*, 22(2):248–260, 1975.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[8] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, 2018.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM SIGCOMM 1989 Conference*, pages 514–528, 2020.

[10] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[11] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, 2022.

[12] J. R. Jackson. Networks of Waiting Lines. *Operations Research*, 5(4):518–521, 1957.

[13] F. P. Kelly. Networks of Queues. *Advances in Applied Probability*, 8(2):416–432, 1976.

[14] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 514–528, 2020.

[15] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, 2014.

[16] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM SIGCOMM 2019 Conference*, page 44–58, 2019.

[17] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A Fault-Tolerant Engineered Network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.

[18] V. Misra, W.-B. Gong, and D. Towsley. Fluid-Based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED. In *Proceedings of the ACM SIGCOMM 2000 Conference*, pages 151–160, 2000.

[19] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, page 537–550, 2015.

[20] J. C. Mogul and J. Wilkes. Nines are Not Enough: Meaningful Metrics for Clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 136–141, 2019.

[21] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[22] D. Nicol and R. Fujimoto. Parallel Simulation Today. *Annals of Operations Research*, 53(1):249–285, 1994.

[23] ns-3 Network Simulator. https://www.nsnam.org, 2020.

[24] OpenSim. OMNeT++. https://www.omnetpp.org, 2018.

[25] OPNET Network Simulator, 2015.

[26] V. Paxson and S. Floyd. Why We Don't Know How to Simulate the Internet. In *Proceedings of the 1997 Winter Simulation Conference*, pages 1037–1044, 1997.

[27] K. Ramakrishnan and S. Floyd. A Proposal to Add Explicit Congestion Notification (ECN) to IP. Technical report, RFC 2481, January, 1999.

[28] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123–137, 2015.

[29] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, page 183–197, 2015.

[30] B. K. Szymanski, A. Saifee, A. Sastry, Y. Liu, and K. Madnani. Genesis: A System for Large-scale Parallel Network Simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS)*, 2002.

[31] R. Winter, R. Hernandez, G. Chawla, A. Faustini, C. Solder, T. Scheibe, D. Law, S. Ayandeh, B. Booth, B. Kohl, C. Lavacchia, S. Krishnamurthy, R. Karthikeyan, E. Multanen, and M. Wadekar. Ethernet Jumbo Frames. http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf, 2009.

[32] Q. Yang, X. Peng, L. Chen, L. Liu, J. Zhang, H. Xu, B. Li, and G. Zhang. DeepQueueNet: Towards Scalable and Generalized Network Performance Estimation with Packet-Level Visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 441–457, 2022.

[33] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 11 2017.

[34] Q. Zhang, K. K. Ng, C. Kazer, S. Yan, J. Sedoc, and V. Liu. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proceedings of the ACM SIGCOMM 2021 Conference*, pages 287–304, 2021.

Name	Matrix	Sizes	Max load	σ
W0	A	CacheFollower	~20%	2
W1	B	WebServer	~20%	2
W2	C	Hadoop	~20%	2

Table 6. The three workloads mixed together in §A.

[35] S. Zhao, R. Wang, J. Zhou, J. Ong, J. C. Mogul, and A. Vahdat. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 221–234, 2019.

[36] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, page 523–536, 2015.

A Mixed Workloads

Parsimon's methods are designed to estimate performance distributions rather than per-flow metrics. However, it is often useful to aggregate FCT performance estimates in different ways. For example, an operator may wish to estimate the performance of individual virtual networks or individual services. In this section, we conduct a simple experiment to assess Parsimon's ability to estimate performance for separate aggregates.

We start by mixing three different workloads—each with its own traffic matrix and flow size distribution—into one workload. Table 6 summarizes their differences. Each workload has a maximum load setting of 20% and a high burstiness setting ($\sigma = 2$), and their combination results in a maximum link load of about 50%. We run the combined workload on the small-scale topology with 2-to-1 oversubscription from §5.3, and we observe the accuracy for each workload faceted by flow size. Fig. 11 shows the cumulative distribution function (CDF) of FCT slowdown for ns-3 and Parsimon. We observe that across all workloads and flow size bins, Parsimon maintains good accuracy.

B Link Failures

One operational use case for Parsimon is to estimate counterfactual network performance in the presence of potential link failures or planned outages. In this section, we use the sample scenario from §5.4 (matrix A, the Hadoop flow size distribution, low burstiness, 2-to-1 oversubscription, and a maximum link load of 68%) to evaluate Parsimon for this use case. For this configuration, the error in estimated p99 FCT slowdown between ns-3 and Parsimon was around 10%. Since link failures increase the load on the remaining links, we should expect some decreased accuracy for Parsimon in this case. On the other hand, simulating all possible network failures in ns-3 would be prohibitively expensive.

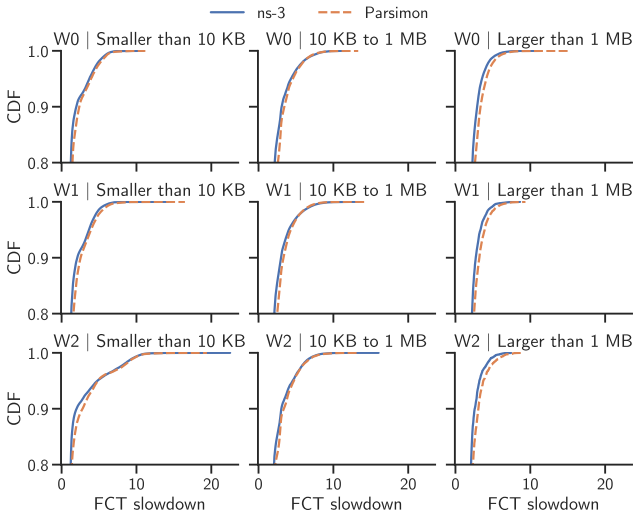
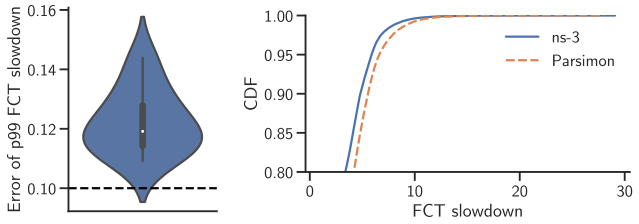


Figure 11. CDFs of FCT slowdown for ns-3 and Parsimon, bucketed by workload and flow size. Note the y-axes. When mixing workloads in a single simulation, Parsimon can accurately estimate performance distributions for individual workloads in addition to full-network aggregates.



(a) p99 errors **(b)** CDF with the max p99 error (0.144)

Figure 12. Errors between ns-3 and Parsimon in estimated FCT slowdowns when there is a link failure. Fig. 12a shows the error distribution for p99 estimates from ten trials—each with one random link failure—with the dashed line showing the error with no link failure. Fig. 12b shows the CDF of FCT slowdowns for the trial with the highest p99 error. For the small oversubscribed topology used in this experiment, a link failure modestly increases estimation error.

In selecting links to fail, we only consider links in ECMP groupings, such that the failure of one link causes traffic to be routed to the other links in the group. In Meta's data center fabric [4], this corresponds to links between fabric switches and spine switches and links between ToR switches and fabric switches. In the small 32-rack topology used here (§5.3 for details), there are 96 such links. We run ten trials, each time picking a random one of the links to fail, keeping the workload constant. We note that this setting represents a particularly bad case for Parsimon: in addition to the high link loads, the scenario uses an all-to-all communication pattern on a small and oversubscribed topology, which means each link failure in the core can have an outsized effect on other core links.

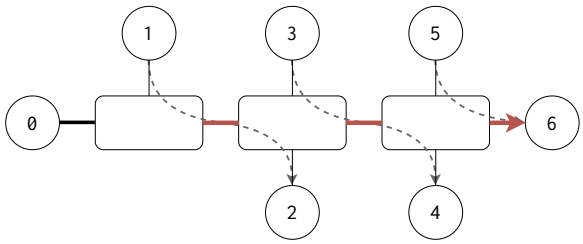


Figure 13. The parking lot topology used in §C. In this topology, zero sends to six, one sends to two, three sends to four, and five sends to six. We refer to the traffic from zero to six as *main traffic* and to all other traffic as *cross traffic*. The bolded red links contain both main traffic and cross traffic, and we call them *congested links*.

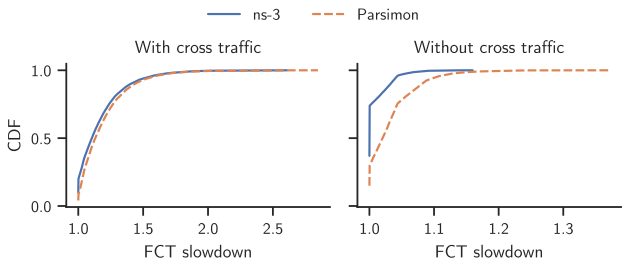


Figure 14. CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the main traffic, both with and without cross traffic. When there is cross traffic, errors arising from first-hop delays are second-order, as most delays are cause by queueing on the congested links. However, when there is no cross traffic, those errors become dominant. The graph on the right uses the same workload as the one on the left, except the cross traffic is removed. Note the different x-axes.

Fig. 12a shows the distribution of errors in p99 estimates. With a single link failure, the errors range from 11% to 14%, with a median error of 12%. Fig. 12b shows the estimated CDFs of FCT slowdown for the trial with the highest error.

C Studying Error Sources

Recall from §3.6 that `Parsimon`'s approximations induce errors in its end-to-end estimates. In this appendix, we use microbenchmarks to study the effects of some pathological cases on `Parsimon`'s accuracy. For an initial discussion on these topics, please refer to §3.6.

Throughout, we use the parking lot topology shown in Fig. 13 with 40 Gbps links. The flow of traffic through the topology is shown with arrows and described in the caption. We refer to the traffic from node zero to node six as *main traffic* and to all other traffic as *cross traffic*. The bolded red links contain both main traffic and cross traffic, and we call them *congested links*. In all experiments, we set the load of the main traffic to 25%. When there is cross traffic, its load is also 25%, yielding a total load of 50% on all three congested links. Lastly, to isolate the effects on the main path from zero to six, we measure FCT slowdown distributions only for the main traffic.

C.1 First-Hop Delays

First, consider the case where all traffic in Fig. 13 originates from node zero and is destined to node six, and recall that all links have the same capacity. In a real network, all queueing in this scenario would occur at the first hop. Subsequent hops would see traffic completely smoothed, and they would therefore contributing zero queueing delay.

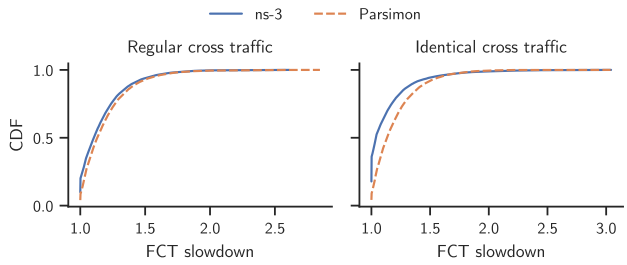
If we re-examine how link-level topologies are constructed in Fig. 4, we see that this smoothing effect is captured, since all traffic passes through edge links with the original edge-link capacities. However, for the link-level topologies in cases B and C of Fig. 4, it is possible for first-hop edge links to contribute delays that will be (erroniously) attributed to the target link. In most cases, we expect the magnitude of this error to be small. A target link will almost always have multiple sources, and only the traffic passing through the target link is simulated. Consequently, the first-hop delays in link-level simulation are expected to be small compared to delays accrued at target links.

The scenario which we first described—in which all traffic on a path originates from a single source—represents the worst case. Here, all target links (aside from the first hop) contribute no queueing delay, thus magnifying the error induced by repeatedly counting the first-hop delays for each target link. Fig. 14 shows this effect. In this experiment, the main traffic consists of one kilobyte flows, and the cross traffic consists of 10 kilobyte flows. All traffic follows a Poisson arrival process. With cross traffic, we see from the graph on the left that `Parsimon` accurately estimates the FCT slowdown distribution of the main traffic. However, when we remove the cross traffic, as done to produce the graph on the right, we see substantial error in `Parsimon`'s estimates due to the first-hop delays previously described. We note that this error exists *even when there is cross traffic*, but the error contributes so little to total delays—which are dominated by queueing at congested links—that `Parsimon` still maintains good accuracy.

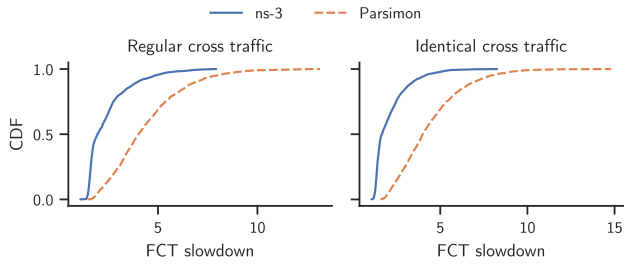
C.2 Correlated and Simultaneous Delays

Next we examine the effect of correlated and simultaneous delays on `Parsimon`'s accuracy. We begin by artificially correlating delays and examining the effect on estimated slowdown distributions. Note that if the delays along a path are positively correlated—for example, if the probability of encountering delay at hop i+1 is higher given there is delay at hop i—then we also expect to see more simultaneous delays along the path. We create these correlated delays by modulating the cross traffic. For regular unmodified cross traffic, we use the same setup as in the previous subsection (§C.1). To artificially correlate delays, we replicate the exact sequence of flows from source one on sources three and five in Fig. 13, so that all three sources of cross traffic send the same flows at the same time. This produces an extreme case of correlation.

Because short-flow and long-flow estimates have different sources of error, we separate the two cases when generating the main traffic. For short flows we use the same one kilobyte flows as before, and for long flows we generate flows that are 10 times the maximum bandwidth-delay product, or 400



(a) Short flows (1 KB), Poisson cross traffic



(b) Long flows (400 KB), Poisson cross traffic

Figure 15. CDFs of FCT slowdown estimated by ns-3 and `Parsimon` for the main traffic with regular or identical cross traffic. The main traffic consists either of short flows (Fig. 15a) or long flows (Fig. 15b). When delays are artifically correlated by replicating the same cross traffic across hosts, accuracy decreases for both short and long flows, with long flows seeing larger errors. In fact, long-flow estimates have significant error even when delays are not explicitly correlated; this is due to the simultaneous delays induced by the smooth Poisson cross traffic.

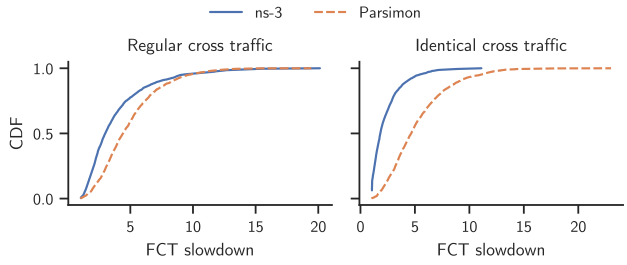


Figure 16. CDFs of FCT slowdown for the same scenario as in Fig. 15b, but with bursty cross traffic (log-normal inter-arrival times, $\sigma = 2$). When the cross traffic is bursty, long flows experience fewer simultaneous delays with regular cross traffic. This results in less error in `Parsimon`'s estimates.

kilobytes. Fig. 15 shows the effect of correlating delays on `Parsimon`'s accuracy for short and long flows.

Short-flow main traffic. In the case of short flows (Fig. 15a), a chief effect of increased correlation is to alter the probability that a flow will encounter queueing. For example, suppose a short flow traverses only two links at 50% utilization. If the delays of the two links are independent, we can estimate the probability that the flow encounters no delay (i.e., no queueing) as $50\% \times 50\% = 25\%$. However, if the delays are perfectly positively correlated, then the probability that the flow encounters no delay increases to 50%. `Parsimon` does not capture this effect because it treats all links independently; in this experiment, this manifests as slight overestimates in FCT slowdown distributions.

Long-flow main traffic. While the total delay for a short flow can be thought of as the sum of individual link delays, the same reasoning does not straightforwardly extend to long flows. Unlike a short flow, a long flow occupies multiple hops at the same time, and only the bottleneck at each instant contributes to end-to-end delay. Summing link delays is therefore only appropriate if different hops contribute significant delays at largely different times. However, `Parsimon` always aggregates individual link contributions by adding them, regardless of whether a link was the bottleneck when the delay was incurred. When we turn our attention to Fig. 15b, we see that not only is the effect of identical cross traffic more severe, but also there is significant error even with regular cross traffic. This is because the cross traffic is smooth (recall that it uses uniform flow sizes and a Poisson arrival process). Smooth traffic results in small but frequent delays at congested links, increasing the chance that long flows will experience simultaneous delays.

In Fig. 16, we duplicate the scenario in Fig. 15b, except we make the cross traffic bursty by using a log-normal inter-arrival time distribution with shape parameter $\sigma = 2$. Because the cross traffic is bursty, there is less simultaneous delay in the regular case, and the induced error is less dominant. Consequently, `Parsimon`'s estimates are closer to the ground truth in the graph on the left. Identical cross traffic still induces large and frequent simultaneous delays, so the errors remain in the graph on the right.

D Clustering Details

Here we briefly describe the distance function and the thresholding critera we use in the evaluation (§5) for clustering link-level simulations. First, recall from §4.2 that the link features we extract are 1) the average load, 2) the flow size distribution, 3) the inter-arrival time distribution. For any two links, we compute distances between their features, and we cluster the links together if the distances are under some threshold.

Distance functions. To compute a distance between link loads, we compute the error. If a and b are two link loads, error e is computed as

$$e = \frac{|a-b|}{a}.$$

To compare distributions, there are many options. We opt for a function that is 1) easily interpretable, 2) scale-independent, and 3) adequately captures differences in the tail. To compute a distance between two distributions, we extract 1,000 percentiles from each of them, and we compute a weighted mean absolute percentage error (WMAPE) between them. Suppose A and B are the sequences of extracted percentiles.

Then, WMAPE is computed as

$$\text{WMAPE} = \frac{\sum_{i=1}^{n} |A_i - B_i|}{\sum_{i=1}^{n} |A_i|}.$$

For our purpose, A_i and B_i are non-negative for all i. We note it is a bit counterintuitive for our distance functions not to commute. However, we have found that it is easy to set thresholds for these metrics, and they produce adequate clustering for the workloads under study.

Distance thresholds. Recall that we only want to cluster two links together if we expect their simulation outputs to be similar. Consequently, when setting a threshold for link loads we must consider the network and the workload being assessed. At high load, small differences in link loads can yield large differences in the tails of FCT distributions; in these cases, we typically set tighter thresholds to preserve accuracy (as usual, this is subject to a speed-accuracy trade-off). For highly-loaded networks, we commonly require $e < 0.001$ or $e < 0.002$ for links to be clustered together. Ideally, this decision would be made on a link-by-link basis, so that tighter thresholds would be set only for high-load links—doing so may allow for more liberal clustering of the low-load links contributing little delay, resulting in more pruned simulations. However, the current prototype sets a single threshold per simulation. To set a threshold between distributions, we typically require WMAPE < 0.1.

Shockwave: Fair and Efficient Cluster Scheduling for Dynamic Adaptation in Machine Learning

Pengfei Zheng[1], Rui Pan[1], Tarannum Khan[2], Shivaram Venkataraman[1] and Aditya Akella[2]
[1]University of Wisconsin-Madison, [2]The University of Texas at Austin

Abstract

Dynamic adaptation has become an essential technique in accelerating distributed machine learning (ML) training. Recent studies have shown that dynamically adjusting model structure (e.g., lottery ticket hypothesis [16]) or hyperparameters (e.g., batch size [1]) can significantly accelerate training without sacrificing accuracy. However, existing ML cluster schedulers are not designed to handle dynamic adaptation. We show that existing schemes fail to provide fairness and degrade system efficiency when the training throughput changes over time under dynamic adaptation. We design *Shockwave*, a scheduler with future planning that builds on two key ideas. First, *Shockwave* extends classic market theory from static settings to dynamic settings to co-optimize efficiency and fairness. Second, *Shockwave* utilizes stochastic dynamic programming to handle dynamic changes. We build a system for *Shockwave* and validate its performance with both trace-driven simulation and cluster experiments. Results show that for traces of ML jobs with dynamic adaptation, *Shockwave* improves makespan by 1.3× and fairness by 2× when compared with existing fair scheduling schemes.

1 Introduction

GPU-powered deep neural network (DNN) training is rapidly becoming a core workload in data centers [25, 28, 29]. Due to the sheer volume of training data and massive, ever-increasing model sizes, many DNN models cannot be trained on a single GPU device, and distributed, multi-GPU training has become the norm. The increasing demand for GPU devices motivates enterprises to consolidate their hardware resources and run their workloads in a shared GPU cluster [25]. Thus, building scheduling mechanisms that can fairly arbitrate among jobs competing for GPU resources and efficiently schedule them for high cluster utilization is important.

While there has been a plethora of work in designing schedulers for DNN workloads, they do not use a rigorous approach to co-optimize system efficiency and fairness. Systems like Gandiva [41] and Tiresias [21] optimize makespan and average JCT (Job Completion Time) with techniques such as dynamic scaling, time-slicing, and over-subscription, but do not consider fairness. Processor sharing [40] based approaches such as DRF [17] and Gavel (Weighted Max-Min Fairness) [33] provide instantaneous fair share of (dominant) resources in each scheduling round, but this can significantly undermine efficiency [20, 35]. Stride [39] scheduling-based approaches such as Gandiva-Fair [10] require cluster operators to explicitly specify an individual job's share (e.g., A 20% and B 80% of GPUs), and manually specified fixed shares can violate long-term fairness for ML jobs [29]. Finally, AlloX [28] and Themis [29] aim to provide long-term fairness by adopting a filter-based approach where within each round, a subset of jobs that are furthest from the fair share are filtered, and among the filtered jobs the ones which maximize efficiency are chosen by the scheduler. However, the filter value requires onerous hand-tuning; furthermore, even with careful tuning, using a fixed filter can lead to sub-optimal efficiency and fairness (§2).

We design *Shockwave*, a scheduler that leverages market theory to jointly optimize efficiency and fairness for ML training jobs in a systematic and principled fashion. We formulate a Fisher market [5] where every job receives an equal budget to purchase resources from a central arbiter. The arbiter then computes prices such that the market reaches an equilibrium; i.e., each job's budget is spent to maximize its performance (e.g., training throughput) and all resources are completely sold. Formulating resource allocation using market theory is powerful because achieving market equilibrium guarantees both fairness and efficiency. Each job has equal purchasing power in acquiring resources, ensuring fairness. Further, market-clearing equilibrium ensures work conservation and that each job's performance is maximized given its budget.

While economic theory has been the basis of many prior systems (e.g., DRF [17], Themis [29], and REF [43]), they all assume jobs have known static resource requests. This assumption is no longer true for elastic ML training jobs [24, 30, 36] whose resource requirements dynamically change over time; further, the changes in resource requirements depend on model update patterns, and thus they are unknown apriori.

For example, training jobs can dynamically scale their batch size by computing the gradient noise scale (GNS) [30, 31]. OpenAI has used batch size scaling (from 32 to 32M) to accelerate GPT-3 training by 500x [7] and similarly, BERT-Large training uses dynamic batch sizes (256 to 4096) to achieve a 2.5x speedup [37]. In this paper, we extend market theory to develop an efficient and fair scheduler for ML jobs with elastic resource requirements.

Existing schedulers are either agnostic or reactive to dynamic changes and our experiments show (§3) that they fail to guarantee fairness or significantly degrade efficiency. The key reason for this is that an optimal schedule or weight assignment [10] at the current instant can be suboptimal in the future, and reactively re-prioritizing jobs can be too late to compensate for the under-prioritization in the early phases. State-of-the-art schedulers that accommodate dynamism, e.g., Pollux [36] do so automatically on behalf of jobs, e.g., by automatically scaling batch sizes. We find that this can hurt training accuracy [1, 11] (§2.3); thus, our aim is to let users perform elastic changes as their algorithms demand. Achieving fair allocation under dynamism without assuming any control over said dynamism is challenging, and is not studied in existing research. We present a detailed comparison between *Shockwave* and other schedulers such as Themis [29], AFS [24] and Pollux [36] in Section 2.

To support dynamic changes in resource requirements over time, we extend the classic, static Fisher market and propose a new discrete-time, dynamic market that can ensure long-term efficiency and fairness. Using discrete time helps us capture the effects of running a market repeatedly over many rounds and a dynamic market helps us capture time-varying utility[1] for jobs. For example, consider a scenario where we are running 20 rounds of scheduling for a job. If a job's per-GPU batch size increases by 2× after 10 rounds due to GNS scaling, its utility from being allocated one GPU (u_0) will also double after 10 rounds ($u_1 = 2u_0$). A static market will assume time-invariant utility, and will compute the accrued utility over 20 rounds for the job as $20u_0$; in contrast, a dynamic market can capture the change in utility for the job over time, and can accurately compute the accrued utility over 20 epochs as $30u_0$. Accurately computing the utility can enable the dynamic market to optimize fairness and efficiency over time. We prove that our dynamic market formulation (§4.2) guarantees long-term efficiency and fairness properties such as maximized Nash social welfare over time, Pareto optimality over time, and sharing incentive.

Implementing the dynamic market formulation in real systems is challenging for two main reasons. First, the market formulation needs to know utility values in the future to compute market equilibrium. Dynamic adaptations in jobs are non-deterministically triggered, as they are dependent on gradient values that vary across models and datasets, which makes it challenging to predict utility in the future. Second, solving the dynamic market equilibrium for an (infinitely) long time horizon is difficult and impractical. It is computationally prohibitive and requires forecasting every job's future performance characteristics. Further, as jobs arrive and complete online, we need to periodically solve for the market equilibrium while maintaining low scheduling overheads.

To bridge the gap between theory and systems, *Shockwave* addresses these challenges and implements a dynamic adaptation predictor and an approximate dynamic market. First, we observe that dynamic adaptation for real-world ML workloads follows a handful of patterns, and these patterns can be predicted using Bayesian statistics. We then develop methods to integrate these predictions into our dynamic market formulation. Second, while performing round-based scheduling, we find that planning a schedule for an (infinitely) long time horizon can introduce significant overheads. To maintain low scheduling overheads, *Shockwave* only plans the schedule for a finite length window (e.g, 30-60 minutes), and we design estimators that can capture the effects on long-term fairness and long-term efficiency that arise from short-term planning. This design helps us balance the system overheads without sacrificing long-term objectives.

We evaluate *Shockwave* on a 32-GPU cluster testbed and use a simulator to study large-scale GPU clusters. Using multiple workloads derived from prior, real-world systems [33, 36], we find that *Shockwave* improves makespan by 1.3× and fairness by 2× compared to existing fair DNN cluster schedulers including Themis [29], Gavel [33], AlloX [28], etc. We further evaluate *Shockwave* on differently sized clusters. Using a simulator built with the same scheduler as in a physical cluster we find that *Shockwave* scales to schedule 900 active DNN training jobs on 256 GPUs and maintains the benefits in makespan (1.26-1.37×) and fairness (2.5-3.1×) when compared to existing schedulers. We show that our solver overhead remains low and is less than 12.5% of a two-minute-long round duration.[2] *Shockwave* is open sourced at https://github.com/uw-mad-dash/shockwave.

2 Motivation

We begin by motivating the need to design a new cluster scheduler for machine learning workloads.

Filter f	Worst FTF-ρ	SI	Avg. JCT	Makespan
Adaptive - $1/\frac{1}{3}/\frac{2}{3}$	0.83	✓	5	7
Fixed - 1/3	1.0	✓	5.7	7
Fixed - 2/3	1.1	×	5.7	7
Fixed - 1	1.1	×	6.0	7

Table 1: Themis example: using a fixed filter yields suboptimal JCT and/or fairness compared with an adaptive filter. Figure 1 visualizes the schedule for $f = 2/3$, showing the cluster and job setting, and demonstrates how a filter works.

[1] A utility function maps a job's allocated resource (e.g., GPU) to the resulting performance (e.g., throughput).

[2] The solver runs asynchronously in a separate thread and does not block the main scheduling loop.

Cluster setting: 4 GPUs. Jobs: A, B, C are three DNN training jobs with one iteration.
Serial (1-GPU) iteration times for A, B, and C are 12, 8, and 6.
The number of requested GPUs per iteration for A, B, and C are 3, 2, and 2.

GPU ID \ round ID	0	1	2	3	4	5	6
0	A	B	A	A	B	A	A
1	A	B	A	A	B	A	A
2	B	C	C	B	C	A	A
3	B	C	C	B	C		
f	2/3	2/3	2/3	2/3	2/3	2/3	2/3

Figure 1: Example - Themis [29] with a static filter ($f = 2/3$). In each round of allocation, the filter (grey color) selects $2/3$ of the jobs unfairly treated so far. The resulting FTF-ρ values for jobs (A, B, C) are (0.78, 0.83, 1.1), showing a static filter hurts fairness. As in Themis, we assume a linear slowdown when the number of allocated GPUs is less than requested.

2.1 Jointly Optimizing Fairness and Efficiency

Existing scheduling mechanisms lack a systematic approach to jointly optimize fairness and efficiency. We first formally define a fairness metric: we adopt the definition of fairness used in recent work such as Themis [29], Gavel [33], and Pollux [36]: Finish Time Fairness (FTF) $\rho(G) = \frac{t_{schedule}}{t_{egalitarian}}$; where $t_{schedule}$ represents the job finish time resulting from a policy G, and $t_{egalitarian}$ is $t_{exclusive} \cdot N$; N indicates the number of contending jobs, $t_{exclusive}$ indicates run time when running exclusively with requested resources. FTF $\rho > 1$ ($\rho <= 1$) indicates that a job is unfairly (fairly) scheduled. Note that while we focus on FTF, *Shockwave*'s underlying market formulation can be extended to support other fairness metrics. For example, by assigning different budgets to jobs we can support weighted proportional fairness [27] with the budgets encoding priorities.

Second, we define a policy as efficient if it minimizes makespan, or correspondingly, maximizes cluster utilization given a sequence of jobs.

Instantaneous fair-share sacrifices efficiency. Existing fair sharing policies, such as Processor-Sharing (PS) [40] and its multi-dimensional extension, Dominant Resource Fairness (DRF) [17], guarantee that at each instant, any job in the schedule obtains exactly a $1/N$ share of the (dominant) resources. However, restricting schedulers to instantaneous fair share can adversely degrade long-term efficiency. Previous work in Altruistic Scheduling (AS) [20] has shown that sacrificing instantaneous fair share and letting some jobs altruistically contribute resources can improve efficiency by 26% [20].

Using filters to balance fairness and efficiency is sub-optimal. Given the limitations of instantaneous fair sharing schemes, recent work [28,29] has proposed using round-based schemes that optimize instantaneous efficiency and long-term fairness. Within each round of scheduling, AlloX [28] and Themis [29] select for allocation a fixed fraction (f) of jobs that have attained the least resource share in the past. Within these filtered jobs, the scheduler tries to maximize efficiency. Across rounds, the filter compensates for jobs unfairly scheduled in the past and thus pursues fairness in the long run.

Existing studies pre-specify a fixed value for filter f across rounds, but we find that adopting a fixed filter can incur a loss in average JCT or makespan [29], and filter tuning is challenging. Table 1 uses a simple example with three jobs to show how different filters yield very different performance outcomes: fixed filter values $f = 1$ and $f = \frac{2}{3}$ violate finish time fairness ($\rho > 1$) while $f = 1/3$ leads to worse JCT. We included the full toy examples in Appendix B. Tuning the hand-crafted fairness filter is challenging without any insight into the resulting performance outcomes, and it is more difficult when the workload varies.

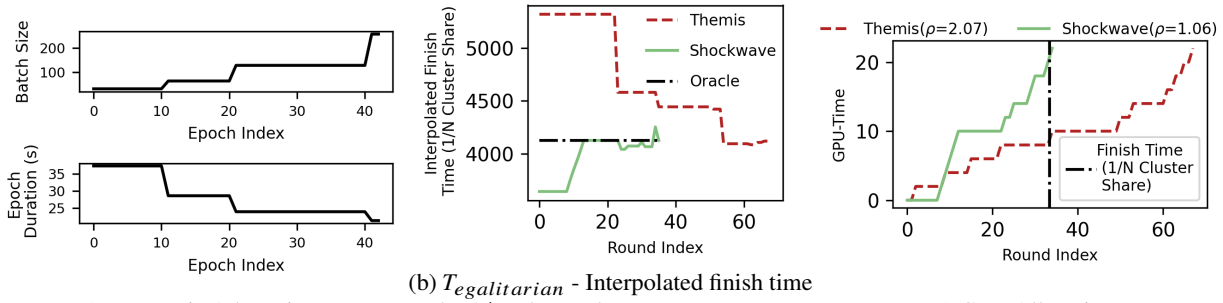
Overall, this argues for a rigorous, systematic approach that jointly and intrinsically (i.e., without knob-tuning) optimizes efficiency and fairness in resource sharing.

2.2 Handling Dynamic Batch Size Scaling

The above goal of optimizing for fairness and efficiency is made further challenging in the presence of dynamism. Dynamism can result in a number of different scenarios. For example, dynamism can result from job arrivals leading to time-variant cluster contention, and systems like AFS [24] are designed to improve JCT by adjusting shares based on job arrivals. On the other hand, dynamism can arise from training jobs that can change their training configurations dynamically. For example, if a job uses gradient noise scale (GNS) [30,31], the batch size used can change dynamically. This can affect fair scheduling because when a job switches to using a large batch size, the per epoch time will decrease, and thereby its remaining running time will also decrease (Figure 2(a)). Unlike prior systems which only handle dynamism that arise from job arrivals, *Shockwave* (and prior work in Pollux [36]) focus on handling dynamic changes in batch sizes of training jobs.

Being agnostic or reactive to dynamic adaptation breaks finish time fairness. We show that being agnostic or reactive to dynamic changes (or dynamic adaption) can yield severe unfairness. Finish time fairness (FTF) implies a soft deadline $t_{egalitarian} = t_{exclusive} \cdot N$ for job completion; the later the job finishes after the deadline, the more unfair the schedule is. Computing FTF requires computing the exclusive run time (i.e.,$t_{exclusive}$), which is straightforward for static jobs since run time roughly equals training throughput (samples per second) times the remaining amount of work (remaining number of epochs). However, for jobs that use dynamic adaptation, future training epochs could be significantly faster because of using a larger batch size. Agnostic scheduling and reactive scheduling are both unaware of future speedups and hence overestimate run time, and thus mistakenly extend the deadline $t_{egalitarian}$ leading to significant unfairness.

Figure 2b uses a job from our trace to show the difference between Themis, which uses a reactive approach, and *Shockwave*, which uses a proactive approach. The job dynamically doubles batch size three times from 32 to 256, and gradually boosts training speed by up to 1.7× (Figure 2a). Themis is notified and updates the job's throughput immediately after each batch size scaling, and recomputes the estimated finish



(a) Dynamic Adaptation

(b) $T_{egalitarian}$ - Interpolated finish time under $1/N$ cluster share

(c) GPU Allocation

Figure 2: Example - Reactive scheduling (Themis [29]) for dynamic adaptation breaks finish time fairness. Proactive scheduling (*Shockwave*) for dynamic adaptation preserves finish time fairness.

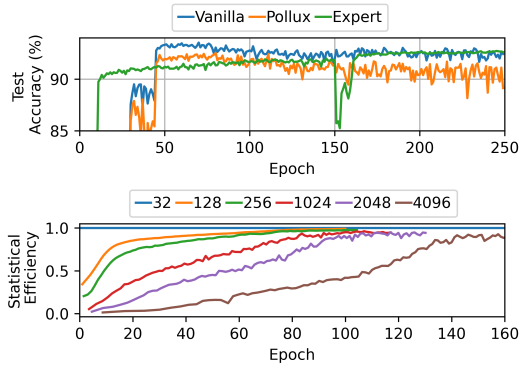


Figure 3: Comparing model accuracy (ResNet18-CIFAR-10) for vanilla training, expert-set batch size scaling, and Pollux autoscaling. The legends in the bottom figure indicate batch size.

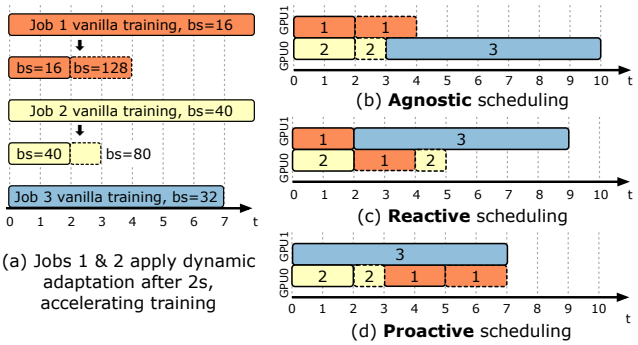


Figure 4: Being agnostic and/or reactive to dynamic adaptation undermines efficiency while proactive scheduling minimizes makespan and maximizes efficiency.

time based on that (red dashed line in Figure 2b). Changes in the estimated finish time lead Themis to detect that the job has received less than its fair share and Themis attempts to prioritize this job in future scheduling rounds. However, the job has already suffered from under-prioritization in its early phases and misses the fairness deadline by 2.07× (Figure 2c). Agnostic scheduling is even worse and increases the job's FTF ρ to 3.07; we omit this from the figure.

Being agnostic or reactive to dynamic adaptation degrades system efficiency. Many makespan-minimizing algorithms, such as Mixed Integer Linear Programming (MILP), Longest Processing Time (LPT) [14], and JCT (Job Completion Time) minimization algorithms such as Shortest Remaining Time (SRPT) and AlloX [28], rely on the exact job run time, or the ordering of jobs' run time to derive a schedule.

However, dynamic adaption adaptively changes a job's throughput, and thus a job's run time can be reduced (when batch size is increased) or prolonged (when batch size is decreased) on the fly. This means that when making scheduling decisions, the job run time estimated using initial or current throughput is only valid for the current instant, and if it is used beyond that system efficiency can be significantly undermined. In Figure 4, the example shows that for MILP makespan minimization, being reactive to dynamic adaptation yields a 22.3% worse makespan and 28% worse cluster utilization compared to proactive scheduling. Reactive scheduling considers $J1$ and $J2$ as long-running jobs from their initial throughput and prioritizes them to minimize makespan. But due to dynamic adaptation, $J1$ and $J2$ become shorter than $J3$ in their second epoch, and it is too late to compensate and re-prioritize $J3$. Being completely agnostic to dynamic adaption is even worse, yielding a 30% worse makespan.

Overall, the above points motivate the need for a scheduler that can model future dynamic adaptation and account for this uncertainty while optimizing for both fairness and efficiency.

2.3 Supporting User-defined Dynamic Ddaptation

While dynamic adaptation with batch size scaling is a key enabler for efficient training of large-scale DNNs, improper changes to the batch size can adversely impact convergence properties and degrade model accuracy. Thus, unlike systems such as Pollux [36] which automatically modify the batch size of training jobs, we argue that schedulers should support user-defined dynamic adaptation schedules to avoid affecting training accuracy. This is mainly because no adaptive batch size scaling technique works consistently well across all datasets and optimizers. As a result, ML researchers have developed many different batch sizing scaling policies including linear scaling rule [7], Accordion [1], Gradient Noise scale (GNS) [36], SimiGrad [37], Hessian eigenspectrum [42], etc.

We next study an example of how choosing an incorrect batch size schedule can affect accuracy. In Figure 3, we consider a CIFAR-10 training job on 2 GPUs with ResNet18 with an initial batch size of 32. When using Pollux [36], the end-to-end training time reduces by 5× as Pollux scales up the batch size from 32 to 64 at epoch 1, and then up to 314 at epoch 2, then to 690 at epoch 30, and finally up to 1682 at epoch 70 till completion. However, this aggressive scaling leads to a 2-3% accuracy loss. Plotting the statistical efficiency (as defined in Pollux [36]), we find that using large batch sizes in the first 30 epochs leads to accuracy degradation. Our conversation with the authors of Pollux suggests that the accuracy degradation depends on the initial batch size used (32 in this case) and can thus vary across jobs.[3]

We also tested an expert heuristic for batch size scaling of ResNet18 training on CIFAR-10. The heuristic scales up the batch size when the gradient norm [1] has insignificant (<50%) changes, and does not scale in the initial epochs 20 epochs and the 10 epochs before and after each learning rate decay. This expert-defined scaling schedule has minimal accuracy loss and is 3× faster than vanilla training. Such expert heuristic and the associated thresholds vary across models and datasets; the above heuristic is specific to ResNet-18-CIFAR-10 training and is not easily transferable. For example, for ResNet-50-ImageNet training, the experts propose a different heuristic that scales up the batch size by a factor of ten at the 30th, 60th, and 80th epoch, respectively [38]. Thus, while prior work has developed scheduling mechanisms for specific dynamic adaptation techniques, in *Shockwave*, on the other hand, we assume no preference for any technique and respect any choice made by users regarding how to dynamically scale a training job's batch size.

In summary, we find that automatically performing dynamic adaptation runs the risk of accuracy degradation. Hence in this work, we aim to develop a scheduler that can observe and forecast future scaling events but treats dynamic adaptation as a part of the user's program that cannot be modified.

3 Overview

We next present an overview of *Shockwave*, a new scheduling framework that jointly optimizes efficiency and fairness for machine learning workloads in shared clusters.

Using Market Theory for Efficiency and Fairness In *Shockwave* we propose using market theory to provably guarantee efficiency and fairness for resource sharing. While prior schedulers [29, 44] have also leveraged market theory for fair sharing, they are built on static market models which assume that resource requests for a job don't change over time. We find that the fairness and efficiency guarantees of a static market do not hold when jobs dynamically change over time [15]. Thus, *Shockwave* extends the classic, static market to a discrete-time, dynamic market, to support efficient and fair resource sharing under dynamic adaptation.

Predicting Dynamic Adaptation Building a dynamic market alone is not enough as it presumes perfect future knowledge of jobs' dynamic adaptation behavior; that is, the market needs to know when and how much jobs' performance (e.g., training throughput) is changed by dynamic adaptation as training progresses. As ML training itself is a stochastic process, the trajectory of dynamic scaling is intrinsically uncertain. We address this problem in *Shockwave* by forecasting the trajectory of dynamic adaptation and developing methods to use these forecasts in the dynamic market.

Scalable System Implementation Solving a dynamic market and predicting dynamic adaptation introduces scheduling overhead. We build a centralized, round-based scheduler [33] and incorporate tractable approximations that can ensure the overhead remains low even as we scale the number of GPUs and cluster size. We find that *Shockwave* can maintain low overhead while scheduling every 120 seconds and scale to handle 900 active jobs running on a cluster of 256 GPUs.

4 Dynamic Market Theory Formulation

We begin by describing our theoretical formulation of a discrete-time, dynamic market and the properties it provides.

4.1 Volatile Fisher Market

Market theory provides a fundamental approach to provably guarantee efficiency and fairness in resource sharing. The equilibrium of a Fisher Market [5], which is solved by maximizing Nash Social Welfare (NSW) [8], is a strong condition that implies all fairness guarantees used in prior systems. It is known that Fisher market equilibrium (under equal endowment) implies Pareto Optimality (PO), Envy-freeness (EF), and Proportionality (PR), which are fairness properties adopted by existing systems like DRF [17].

We define efficiency in terms of the utility of a job, where utility is a function that maps allocated resources to the resulting job progress (e.g., throughput improvement if we allocate more resources). The market equilibrium for a Fisher market has also been shown to maximize efficiency [6]. Thus, we explore the applicability of Fisher markets for DL jobs.

From static market to dynamic markets: Volatile Fisher Market. Classic Fisher Market assumes static, time-invariant utility for jobs, and a recent study [15] shows that efficiency and fairness guarantees can be violated for dynamic, time-variant utilities. Prior work [2, 3] on dynamic markets has also studied settings where goods (resources) arrive online, while our market considers a different setting where buyers in the market have time-variant utilities over goods.

To perform efficient fair sharing under dynamic adaptation, we extend the static Fisher market to a discrete-time, dynamic market. We name this new market **Volatile Fisher Market (VFM)**. We prove that maximizing Nash Social Welfare Over Time (i.e., $\mathsf{NSW}_{\mathsf{OT}}$ in Equation 1) solves the market

[3] We also found that the statistical efficiency metric in Pollux can be incorrect for Neural-MF models [22]. We include details of this experiment in Appendix A.

equilibrium of VFM and establishes long-term efficiency and fairness properties, such as Proportionality Over Time, i.e., $\mathsf{PR}_{\mathsf{OT}}$, which has strong implications for finish time fairness and sharing incentive. We leave the formulation and related proofs of VFM in Appendix C-D, and provide a succinct description below.

VFM operates at discrete time intervals $t = 1, \ldots, T$. At each time instant, a central seller (the scheduler) sells resources (e.g., GPUs and/or CPUs) to buyers (jobs). All resources are volatile. That is, resources bought by a job at time t' cannot be carried over to the future time steps $t > t'$. To model dynamic adaptation, the utility for any job i is a sequence of time-variant functions u_{it} $(t = 1, \ldots, T)$. For example, a job might have a utility u_0 when the batch size is 16 and its utility could double $u_1 = 2u_0$ when the batch size doubles at $t = 1$. Since jobs' utilities can change over time, this creates dynamic changes in demands over time, and thus, resource price, which is achieved at equilibrium, is also time-variant. We assume that each job is endowed with an initial budget to spend across rounds. The budget for a job reflects its purchasing power and different budgets can reflect scheduling priority.

Given the resource demands, budget, and utility for each job, at every time instant, the VFM solves for an allocation and assignment of prices that can lead to market equilibrium. We define the market to have reached an equilibrium when two conditions are satisfied. **(a) Optimal Spending**: Each job's utility accrued over time, i.e, $\sum_{t=1}^{T} u_{it}$, is maximized under its budget. **(b) Work-conserving**: There are no leftover resources if the price for the resources is non-zero.

4.2 Equilibrium Properties

The market equilibrium achieved by VFM has a number of powerful properties that we define below. Proofs for them are in Appendix C-E.

Cluster-level performance. The equilibrium of VFM maximizes Nash Social Welfare Over Time($\mathsf{NSW}_{\mathsf{OT}}$) which is an indicator of cluster-level performance.

$$\mathsf{NSW}_{\mathsf{OT}}(U_1(\boldsymbol{X_1}), \ldots, U_N(\boldsymbol{X_N})) = \prod_i U_i(\boldsymbol{X_i})^{\frac{B_i}{\sum_i B_i}}, \qquad U_i(\boldsymbol{X_i}) = \sum_t \boldsymbol{u}_{it}(x_{it}) \tag{1}$$

Let $U_i(X_i) = \sum_t u_{it}(\boldsymbol{x}_{it})$ represent the utility (e.g., epoch progress) for a job i accrued over rounds $t = 1, \ldots, T$, $\boldsymbol{X_i}$ represent the sequence of allocations $\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{it}, \ldots, \boldsymbol{x}_{iT}$ received for individual rounds, and B_i represent the budget provided for the job. Maximized $\mathsf{NSW}_{\mathsf{OT}}$ guarantees that the (weighted) geometric mean of job progress is maximized for a T-round-long time horizon. In effect, this property guarantees that the overall cluster-wide utility is maximized across all jobs, thus leading to improved utilization.

Pareto Optimality over time. We prove that maximized $\mathsf{NSW}_{\mathsf{OT}}$ also implies Pareto Optimality Over Time ($\mathsf{PO}_{\mathsf{OT}}$), which guarantees resource allocation efficiency. Specifically, $\mathsf{PO}_{\mathsf{OT}}$ ensures that each job has no surplus resources at each instant; i.e., we cannot increase one job's training progress without depriving that of another job.

Finish Time Fairness (FTF) over time. We also show that maximized $\mathsf{NSW}_{\mathsf{OT}}$ minimizes the product of FTF across all jobs and that this directly leads to sharing incentive (assuming budgets are equal). A formal statement is in Corollary 4.0.1 and the proof is in Appendix E.

Corollary 4.0.1. *The equilibrium of the Volatile Fisher Market with linear or Leontief utility at each instant (a) minimizes the product of FTF (ρ) across all jobs (i.e., $\prod_i \rho_i$); (b) when the budgets assigned to jobs are equal, the equilibrium provably guarantees Sharing Incentive (SI), i.e., all jobs' FTF ρ are no greater than 1, i.e., $\rho_i \leq 1,\ \forall i$.*

Thus, our formulation of volatile Fisher markets can capture time-varying utility for jobs while providing a number of powerful guarantees in terms of fairness and efficiency.

4.3 Handling Uncertainty

The Volatile Fisher Market model described above assumes perfect knowledge of the future. That is, the model requires knowing at which time point t will the throughput change due to dynamic adaptation. However, dynamic adaptation in jobs is non-deterministically triggered, as they are dependent on stochastic gradient values and can thus vary across models and datasets. To handle this, in §5, we develop methods to predict dynamic adaptation in jobs. But given that the predictions are random variables, we further extend our above formulation to derive a VFM that can handle uncertainty in future request demands. We show that this extension guarantees Maximized Nash Social Welfare Over Time in Expectation ($\mathsf{MNSW}_{\mathsf{OTE}}$). Details are provided in Appendix F.

5 Predicting Dynamic Adaptation

In this section, we develop techniques to predict the dynamic adaptation of the batch size that occurs in elastic ML training workloads. Our key insight in developing a predictor is to leverage our knowledge about techniques that are used for batch size scaling [1, 31] and thereby restrict the search space of possible batch size changes. We next define how changes in batch size over time can be viewed as trajectories and then describe how we can use Bayesian statistics to predict which trajectories are most likely.

Dynamic adaption, regimes, and randomness. We define a regime of training R as a tuple $R = (c, f)$; where c indicates the job configuration (e.g., batch size) used in the regime and f represents the duration (as a fraction of the total epochs) that this regime lasts. For example, if a 100-epoch-long DNN training job starts with batch size 32 (denoted BS_{32}) for epoch 1-20, then the first regime is denoted as $c1 = BS_{32}, f1 = 0.2$. We define a trajectory as a sequence of regimes. For example, if the same job scales up to BS_{64} for epoch 21-80, and finally scales down to BS_{32} for epoch 81-100, then its trajectory is represented as $(c1 = BS_{32}, f1 = 0.2) \rightarrow (c2 = BS_{64}, f2 = 0.6) \rightarrow (c3 = BS_{32}, f3 = 0.2)$. Thus, given a new job, each

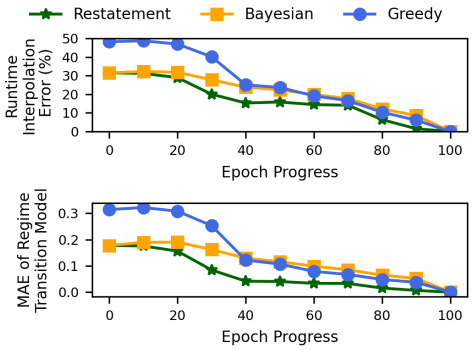


Figure 5: *Shockwave* Dynamic Adaptation Modeling Error.

regime c_i, f_i is a tuple of random variables.

Leveraging domain knowledge. We leverage domain knowledge about techniques used for batch size scaling to constrain the random variables. Techniques for scaling batch size have deterministic patterns. (a) **Accordion [1]** only alternates between two configurations c_1 for small batch size and c_2 for large batch size. When gradient values change slowly (below a threshold) during training, Accordion scales up the batch size from c_1 to c_2, and when gradient values change rapidly, Accordion scales from c_2 back to c_1. (b) **GNS [31]** only scales up the batch size up to the pre-specified limit and never scales down. Existing studies show that the gradient noises tend to grow throughout training, implying that GNS will gradually scale up the batch size and never scale down [30, 31]. We use a simple model of GNS scaling where, as the gradient noise grows above a relative threshold, the batch size doubles.

We choose Accordion and GNS as representative batch size scaling patterns as they have been used in prior systems like KungFu [30] and Pollux [36]. Further, their scaling decisions are completely determined by gradient states (i.e., gradient norms and noises), which encode the stochasticity induced in back-propagation algorithms. Most other dynamic batch sizing policies also adapt to gradient states and we plan to add support for more policies in the future.

Prior for regime transition. Given that batch size scaling rules have deterministic configuration transitions, the only random variable is a regime's duration. For a job with K regimes, we define a probabilistic model $P(f_1, \ldots, f_K)$ to represent the probability that regime k (k=1,...,K) lasts for f_k fraction of epochs. We also note that the sum of all regimes' epoch fractions needs to sum up to 1 [4]. Given this formulation, we use an approach based on Bayesian statistics to predict regime duration. At a high level, our approach is to define a prior distribution of regime duration and then update the posterior distribution in real time as training progresses. The key challenge here is in determining how we can update the posterior as training progresses.

The restatement posterior update rule. Given our problem formulation, we adopt the commonly used Dirichlet prior $Dir(n1, \ldots, n_K)$. A standard Bayesian posterior update rule assumes the epoch samples of individual regimes are independently and randomly drawn as training progresses. But this does not hold in practice. Epochs of the k-th regime can only emerge if the $k-1$-th regime finishes. To deal with the temporal-dependence issue, we design a simple update rule, named the restatement rule, for posterior updates. The restatement rule only updates the prior's parameters that correspond to completed epochs, while continuing to believe that the ongoing and future regimes will evenly split the remaining epochs. Specifically, suppose a user specifies that at a maximum, K regimes can exist, the prior is set as $Dir(N/K, \ldots, N/K)$ for the K potential regimes. When the k-th (k=1,...,$K-1$) regime finishes, suppose the observed epochs for past regimes $1, \ldots, k$ are $m_1, \ldots, m_{k-1}$, we update the posterior distribution to $Dir(m_1, \ldots, m_k, S_k, \ldots, S_k)$, where $S_k = (N - \sum_{k=1}^{K} m_k)/(K-k)$. We compare the Bayesian update rule with the restatement rule in Figure 5 and find the restatement rule has a lower interpolation error; the interpolation error is averaged over 200 jobs randomly drawn from the Gavel workload trace (Section 8.1), each with a batch size scaling schedule imposed by Accordion or GNS.

Predicting job remaining time. Given the predictions from the Bayesian model, we next predict the remaining runtime for a job. This is necessary for estimating finish time fairness. We sum up individual regimes' expected duration to calculate total job runtime. Total job time minus cumulative run time in the past (i.e., T_j) gives the remaining time.

Computational tractability. Finally, as each job can comprise of many possible regime trajectories, at the cluster level, the trajectory space for all jobs is combinatorially large. To avoid space explosion, the scheduler only considers a single regime transition trajectory for each job, which is the mean (expectation) of its posterior distribution model [4].

Evaluating prediction accuracy. Figure 5 shows the online prediction (i.e., mean of posterior distribution) accuracy for regime transition and job run time. We compare *Shockwave*'s restatement rule with two baselines. The first is a standard Bayesian posterior update rule; the second is a greedy approach that forecasts future job run time only using the most up-to-date job throughput, which is used by all reactive schedulers. The evaluation includes 200 Accordion and GNS jobs with real dynamic adaptation trajectories. *Shockwave*'s restatement rule converges to the oracle job run time and the oracle dynamic adaptation trajectory faster than the baselines. Throughout the training, the error in modeling the duration of each regime is on average 6%, which results on average an 84% accuracy in run time prediction. In summary, we see that our proposed predictor for elastic training jobs is able to accurately capture the total run time without prior training and by only observing job progress across epochs. We next discuss how our predictor can be integrated with the market formulation.

[4] We don't make any stationary assumptions about the distribution.

6 *Shockwave* Design

Overview. Figure 6 presents the overall system design of *Shockwave*. When a new job arrives (1), the Bayesian predictor will construct a prior model for the job's batch size scaling schedule and the job is added to the active pool.

As a job makes progress, upon epoch completion or when the job triggers a dynamic batch size scaling (2), the job's Dirichlet posterior model is updated using the restatement rule (3). The posterior model then forecasts the future batch size schedule for this job and delivers it to the scheduler solver. Further, the posterior model predicts the job's (remaining) run time under dynamic batch size scaling and delivers this to the long-term efficiency and fairness estimator.

We design two estimators: a long-term efficiency estimator (4) that can estimate the makespan (time to finish all active jobs) and a long-term fairness estimator (5) that can estimate FTFs for all active jobs. The schedule solver converts the predicted batch size schedules into the utility for each job and synthesizes a generalized Nash social welfare function (6) that uses jobs' FTF estimate as weights and the makespan estimate as a regularizer. Finally, the output of the solver is a schedule for the next T rounds, and this schedule is used by the cluster manager to launch jobs (more details in §7).

We next discuss some design details of how the generalized Nash social welfare function is derived from its inputs. We also present an overview of how the efficiency and fairness estimators work. We include a more detailed explanation in Appendix G.

6.1 Schedule Solver

Output. The solver plans the schedule for a configurable number of future rounds T (the default is 20 two-minute-long rounds). Thus, the output is a $N \times T$ binary matrix X, where N is the total number of active jobs available for scheduling. $\mathsf{X}[j,t] = 1$ ($\mathsf{X}[j,t] = 0$) represents scheduling (descheduling) job J_j in round t.

Objective. The inputs to the schedule solver include the batch size schedule for all jobs, which can be used to derive their utility (epoch progress) UTIL_j, the estimated FTFs $\hat{\rho}_j$, and the estimated makespan H.

The objective of the solver is to maximize the generalized Nash social welfare as shown in Equation 2. $\sum_t \mathsf{UTIL}_j(\mathsf{X}[j,t])$ represents the summed utility of all active jobs. The utility increases when a job is scheduled for more rounds within the planning window, and the sum of the logarithm of utilities, across all jobs, represents the Nash social welfare. We use the k-th (default: 5) power of FTF values $\hat{\rho}_j$ as weights to prioritize jobs that are at risk of violating FTF (e.g., jobs that have been waiting in the queue for a long time). Finally, we add a regularization term that penalizes schedules (in the planning window) that could potentially increase the makespan estimate $H(X)$. Coefficient λ (default: 1e−3) controls the magnitude of the regularizer, Z_0 is a normalization factor that renders the regularizer insensitive to the scale of

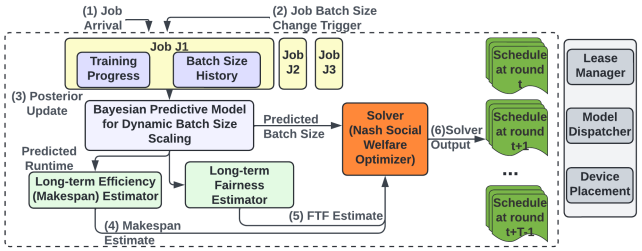


Figure 6: Design of *Shockwave* showing how the different components interact with each other to derive a schedule.

$H(X)$, and M is the total number of GPUs in the cluster:

$$\underset{\mathsf{X}}{\text{Maximize}} \frac{\sum_{j=1}^{N} \widehat{\rho}(j)^k log \sum_t \mathsf{UTIL}_j(\mathsf{X}[j,t])}{NM} - \frac{\lambda \mathsf{H}(\mathsf{X})}{Z_0} \quad (2)$$

We tune the hyperparameters over a large range and find that *Shockwave* performs consistently well around the default hyperparameter values (k in [1,10] and λ in [1e−4, 1e−2]). Exceedingly large or small hyperparameters make the regularization term dominate the Nash social welfare term (or vice versa) and push *Shockwave* away from the Pareto frontier of fairness and efficiency, while the default values strike a balance between them.

Similar to prior work [29,33,36], the solver recomputes the program in Equation 2 either when the planned rounds elapse, or when jobs arrive or complete. If dynamic adaptation is predicted to occur within the planning window, the scheduler needs to incorporate dynamic changes in jobs' throughputs when computing the utility. To account for dynamic changes, we decompose a job's schedule into regimes, where each regime has a fixed batch size and throughput. The generalized Nash social welfare (Equation 2) can then be implemented at a regime level, where the utility of a job equals the summed utility over all regimes.

6.2 Long-term Fairness and Efficiency Estimators

Finish time fairness estimator. We estimate job J_j's finish time fairness (FTF) $\widehat{\rho}(j)$ as its predicted job completion time (the sum of attained service time, waiting time, and the predicted remaining run time), divided by its predicted total job run time. Note that a job's predicted runtime is related to its predicted batch size scaling schedule. *Shockwave* plugs in FTF ρs of jobs into social welfare function (see Equation 2) as weights. The weights in the social welfare function act as the budgets assigned to jobs in the volatile Fisher market. If a job is predicted to be unfairly scheduled (large FTF ρ) in long term, VFM correspondingly assigns a higher budget for it and proactively prioritizes the job in the planning window.

Makespan estimator. The efficiency estimator estimates the makespan to complete all active jobs and penalizes schedules in the planning window that increase the makespan. However, it is challenging to estimate the makespan for all active jobs at a given instant. Thus, in practice, *Shockwave* uses a lower bound [12] of the makespan as a proxy and penalizes

Model	Task	Dataset	Batch Size(s)
ResNet-50	Image Classification	ImageNet	16 - 128
ResNet-18	Image Classification	CIFAR-10	16 - 256
LSTM	Language Modeling	Wikitext-2	5 - 80
Transformer	Language Translation	Multi30k (DE-EN)	16 - 256
Recoder Autoencoder	Recommendation	ML-20M	512 - 8192

Table 2: Workloads used in the evaluation.

increasing the lower bound. More details are in Appendix G.

7 Implementation

Scheduler and worker. *Shockwave* scheduler and worker implement time-sharing of cluster resources with round-based scheduling. Each round is a fixed interval (default: 2 minutes). In each round, the scheduler selects a set of jobs from the active job pool to run. The lease manager translates the schedule to job leases and notifies the workers to launch, suspend, or resume jobs. Each worker binds to a single GPU device.

We adopt a simple job placement engine along with Gavel. The placement engine tries to tightly pack jobs' workers over the machines to minimize fragmentation, and it also tries to place scheduled jobs on their previously executed machines to maximize job locality.

Scheduler solver, lease manager, and model dispatcher. If a job does not run in round T, but is scheduled for round $T+1$, the scheduler will notify the lease manager to create a new lease for it, and dispatch the job to GPU workers before the next round starts. The assigned workers will launch the job when the next round begins. If a job is actively running in round T and the scheduler continues to schedule it for round $T+1$, the lease manager will send a lease extension signal to the job's workers. This job will stay running on the same workers in round $T+1$. If a job is actively running in round T, but the scheduler decides to suspend it in round $T+1$, the job's workers will stop it since its lease will not be renewed.

Shockwave also penalizes frequent restarts as it adds overheads in dispatching models and datasets to workers. The schedule solver prefers to schedule jobs to continuous rounds in the window and penalizes scattering the job's execution across rounds. Furthermore, the underlying device placement engine prefers mapping a job to its previously allocated workers to reduce restarts.

Dynamic adaptation support. When a training job triggers dynamic adaption (i.e, batch size scaling), it notifies the scheduler solver of the occurrence of the event. The cluster manager can configure *Shockwave*'s responsiveness to dynamic scaling. The reactive mode requires *Shockwave* to invalidate its current schedule and immediately trigger resolving in response to dynamic adaptation. The lazy mode continues the original schedule and postpones resolving until the next rescheduling interval. *Shockwave* is by default configured in reactive mode.

Prototype. *Shockwave* is implemented in Python atop ML cluster manager Gavel [33]. We integrate *Shockwave* into Gavel by implementing a schedule solver, meta-data collector, and schedule translator, which translates *Shockwave*'s produced schedule to job leases. Furthermore, *Shockwave* provides an interface for users to monitor gradients and trigger batch size scaling. Scaling requests are sent to the scheduler with gRPC. The schedule solver is implemented with Gurobi [34]. *Shockwave* uses Linux NFS to store model checkpoints. Our checkpointing overhead is less than 3%.

8 Evaluation

We next evaluate *Shockwave* using ML job traces derived from real-world clusters and compare *Shockwave* to state-of-the-art deep learning schedulers.

8.1 Experiment Setup

Testbed. We conduct experiments using a 32-GPU, 8-node cluster on TACC [9]. Each node has 4 NVIDIA Quadro RTX 5000 GPUs (16GB GRAM), 2 Intel Xeon E5-2620 v4 "Broadwell" CPUs, and 128GB DDR4 RAM. The network bandwidth is 200 GB/s inter-switch and 100 GB/s inter-node.

Workload. *Shockwave* 's evaluation uses two separate workloads to reinforce its practical applicability. These traces include diversity in job sizes, model types, and arrival patterns. Unless otherwise specified, we use Gavel's workload generator [33] to construct synthetic distributed training workloads. Job information is detailed in Table 2. The jobs used in this paper range from 0.2 to 5 hours long, with 1, 2, 4, or 8 workers for distributed training, and the arrival of jobs follows a Poisson arrival process with an inter-arrival rate λ ranging from 0.1 to 0.2 [33]. We use a mix of job durations also derived from prior work [36]. We categorized jobs based on total GPU-time, and similar to prior work, we set the probability of generating Small (0.2-8 GPU-hours), Medium (8-16 GPU-hours), Large (16-72 GPU-hours), and Extra Large (>72 GPU-hours) jobs to be 0.72, 0.2, 0.05, 0.03, respectively. Each job is configured with one of the three modes: Static, Accordion [1], or GNS [31]. We increase the total batch size by increasing the per-GPU batch size while preserving the number of workers. In addition to traces generated by Gavel, we also evaluate a production trace of real job duration and arrival timestamps used by Pollux [36] in Appendix J. We also tune the hyperparameters k and λ with the range discussed in Section 6.1.

8.2 Baseline Schedulers

We compare *Shockwave* to six schedulers: **OSSP (Open Shop Scheduling) [18], AlloX [28], Themis [29], Gavel [33], MSS (Max-Sum-Throughput) [33], Gandiva-Fair [10], and Pollux [36]**. All baselines, except Pollux, do not change the number of workers, whereas Pollux dynamically tunes the number of workers (and batch size) to adapt to varied resource avail-

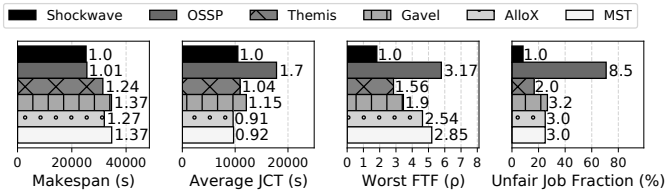


Figure 7: [Physical] Evaluating *Shockwave*'s scheduling efficiency and fairness in a 32-GPU physical cluster. The annotated number beside each bar is the relative value compared to *Shockwave*.

ability. To perform a fair comparison against the scheduling policies of most baselines, in our *Shockwave* prototype, we only perform time-sharing and maintain a fixed number of workers through a job's lifetime, even though the *Shockwave* market formulation can be easily re-parameterized to support worker scaling. Nevertheless, we compare our "constrained" version of *Shockwave* to Pollux in §8.7 and show significant fairness gains and matching efficiency.

Efficiency baseline: makespan. OSSP minimizes makespan using MILP. As minimizing makespan usually translates to maximizing cluster utilization, OSSP provides a baseline for efficiency, but with no guarantee of fairness.

Efficiency baseline: throughput. Max-Sum-Throughput (MST) maximizes the cluster-level throughput at each instant, which is the sum of throughput across all training jobs. MST is an instantaneous efficiency baseline.

Fairness baseline. Gavel [33] implements Max-Min-Fairness [32], an algorithm that performs fair sharing of cluster resources within each allocation round.

Fairness and responsiveness baseline. AlloX [28] minimizes average job completion time with maximal bipartite matching and provides a baseline for responsiveness. Pollux [36] maximizes cluster-wide goodput and uses the $p-norm$ of individual jobs' training goodput for improved responsiveness, while tuning p to penalize unfair allocations.

Fairness and efficiency baseline. Themis [29] uses Partial Allocation [13] for efficient and fair allocation. We use the default filter value for Themis. We also compare against Gandiva-Fair [10], a framework that uses lottery scheduling to guarantee a proportionally fair share of resources and efficiency by being work-conserving.

Performance metrics. We quantify efficiency using makespan and utilization. We measure fairness using two metrics: The first is the fraction of unfairly scheduled jobs, i.e., the fraction of jobs with FTF $\rho > 1.0$; The second is the worst-case FTF ρ, which is the worst-case slowdown due to unfair scheduling. The smaller the unfair fraction and worst FTF ρ are, the better a scheduler is at preserving sharing incentive. We quantify responsiveness using average JCT.

8.3 Evaluating Efficiency and Fairness

We first study the benefits of *Shockwave* using experiments on the physical TACC cluster.

[Cluster - 32 GPUs, 120 Jobs] Efficiency. (cf., Figure 7) *Shockwave* is more efficient than existing fair schedulers with a makespan on average 1.3× less than Themis, Gavel, and AlloX. Compared to our efficiency baselines that have no fairness constraints, *Shockwave* achieves a 37% improvement in makespan over MST and produces a similar makespan as OSSP. Analyzing cluster utilization data we also find that *Shockwave* outperforms Themis, Gavel, and AlloX in cluster utilization by 28% on average.

[Cluster - 32 GPUs, 120 Jobs] Finish time fairness (cf., Figure 7). *Shockwave* is fairer than existing fair schedulers. *Shockwave*'s worst-case FTF (Finish Time Fairness) ρ is 1.82, outperforming Themis, Gavel, and AlloX by 2× on average. OSSP and MST are not fair schedulers, and severely break finish time fairness, the worst-case FTF ρ of which reach 5.79 and 5.2. In addition, *Shockwave* keeps the fraction of unfairly scheduled jobs (i.e., the fraction of jobs with FTF ρ>1) low, outperforming Themis, Gavel, and AlloX by 2.7× on average. OSSP and MST unfairly schedule jobs, and their fraction of jobs that have FTF rho larger than 1 are 70.8% and 25%.

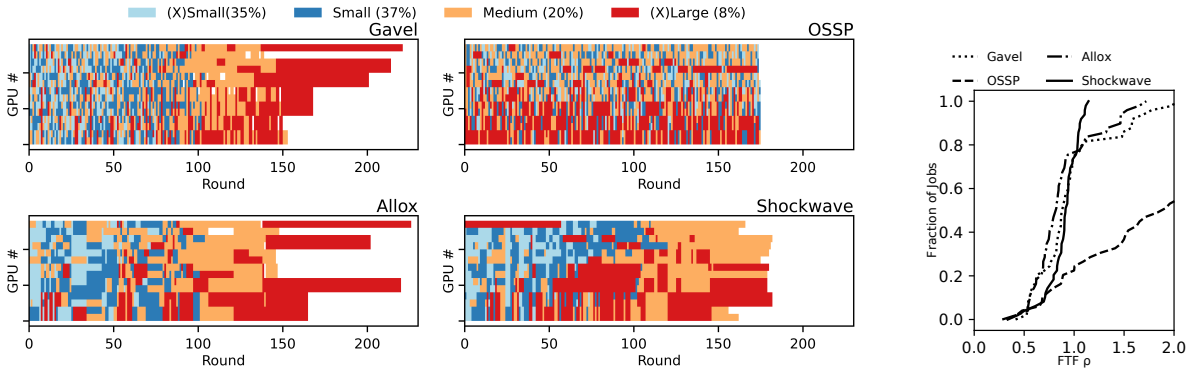
[Cluster - 32 GPUs, 120 Jobs] Average job completion time (cf., Figure 7). *Shockwave* does not sacrifice system responsiveness in exchange for improved makespan and finish time fairness. *Shockwave* produces a similar average job completion time when compared with Themis, Gavel, and MST. AlloX achieves a better average JCT by aggressively prioritizing short jobs (but at the cost of delaying long jobs), while in contrast, OSSP achieves the worst average JCT due to aggressively prioritizing long jobs for tight resource packing over time (but at the cost of delaying short jobs).

Overall, we find that by solving for the optimal efficiency-fairness trade-off, *Shockwave* can improve efficiency and fairness when compared with existing schedulers. By analyzing the scheduling decisions, we find that with *Shockwave*, jobs are opportunistically prioritized to improve long-term efficiency if such prioritization does not affect finish time fairness. Second, we find that *Shockwave*'s solver improves fairness by smart arbitrating. "Rich" jobs (i.e., jobs which have lower chances of violating FTF) yield resources to "poor" jobs which have a higher chance of violating FTF. We next take a closer look at the schedule decisions on a smaller trace to further distill the benefits of *Shockwave*.

8.4 A Closer Look at *Shockwave*'s Schedule

We compare the schedules for a batch of 50 jobs and the FTF ρ between *Shockwave* and baselines to further understand the wins in efficiency and fairness. We categorize jobs into four groups based on their sizes (GPU-time): (X)Large, Medium, Small, and (X)Small (different colors in Figure 8a).

Understanding efficiency improvement. AlloX optimizes system responsiveness (average JCT) by prioritizing small jobs. In Figure 8a, in the first 100 rounds, most of the jobs scheduled are XSmall jobs. The filter in AlloX ensures medium and large jobs do not get starved but these jobs are



(a) Visualized schedules.

(b) FTF ρ CDF

Figure 8: A closer look at how *Shockwave* prioritizes jobs of different lengths while meeting the fairness constraints.

not prioritized. Large jobs trail until round 230 and leading to worse makespan and cluster utilization.

Gavel's max-min fair scheduling prioritizes the least performant jobs. In Figure 8a, throughout the schedule, Gavel prioritizes neither small nor large jobs; jobs of all sizes evenly partition GPUs when compared with other policies. However, restricting scheduling to instantaneous fairness significantly hurts long-term efficiency, and large jobs run on a mostly idle cluster from round 170 to round 220.

Shockwave improves efficiency by opportunistically scheduling (X)Large jobs across rounds but without hurting small and medium jobs' sharing incentive (Figure8b shows the CDF of FTF). In Figure 8a, between round 0 and 50, and 50 to 120, large jobs are opportunistically scheduled by *Shockwave*, and the cluster is tightly packed over the time horizon, resulting in a low makespan. Note that *Shockwave* also preserves responsiveness since most XSmall and small jobs are completed fast (before round 50 and 110, respectively), which is comparable to AlloX.

OSSP over-prioritizes (X)Large and medium jobs throughout the timeline, but significantly delays XSmall jobs' completion (see delayed blocks at the end of the schedule). Delaying small jobs significantly breaks the sharing incentive and undermines cluster responsiveness.

Understanding fairness improvement. Figure 8b shows the FTF ρ CDFs of different policies for the batch of jobs visualized in Figure 8a. *Shockwave* improves efficiency without sacrificing the sharing incentive: the worst-case FTF ρ for the batch of jobs is 1.23, and the fraction of unfair jobs is low. In Figure 8a, AlloX and Gavel's CDF grows faster than *Shockwave*'s for $\rho <= 1$, although more than 20% of jobs have $\rho > 1$. AlloX and Gavel over-prioritize some jobs and this results in an allocation that exceeds sharing incentive. *Shockwave* avoids over-prioritization and is thus able to have more jobs meet the sharing incentive. *Shockwave* also improves fairness by predicting dynamic adaptation for a more accurate estimate of the FTF deadline. Figure 2 shows an example where *Shockwave* produces an accurate prediction of the FTF deadline and enables the job to finish on time.

Makespan (s)	Average JCT (s)	Unfair Fraction (%)
4.97%	4.62%	3.83%

Table 3: Fidelity of *Shockwave*'s simulator – difference between simulator and physical cluster.

8.5 Scaling to Large Clusters

We next use simulation to compare *Shockwave*'s and baseline algorithms' efficiency and fairness in larger-scale cluster settings. We scale both the cluster size and the number of jobs and study 64 GPUs with over 220 jobs, 128 GPUs with over 460 jobs, and 256 GPUs with over 900 jobs. We preserve the contention factor as roughly three to maintain a constant level of resource contention regardless of scale. Note that our physical cluster implementation and the simulator use the same scheduling code base and solver engine. We begin by validating our simulator's fidelity.

Simulation Fidelity

We evaluate the simulation fidelity by comparing our simulator's results with the 32 GPU physical cluster results (Table 3). We run all policies supported by our system under different workloads, and the average difference is reported in Table 3. Overall, the performance difference between a simulated and physical cluster run is around 5%.

[Simulation - 64-256 GPUs, 220-900 Jobs] Efficiency. As shown in Figure 9, *Shockwave* scales to large cluster settings and preserves the improvement in makespan over baseline algorithms. *Shockwave* achieves 1.26-1.35×, 1.3-1.34×, 1.35-1.37×, and 1.21-1.3× speedup in makespan when compared with Themis, Gavel, AlloX, and Gandiva-Fair respectively. *Shockwave* achieves a marginally worse (5%-9%) makespan compared with OSSP.

[Simulation - 64-256 GPUs, 220-900 Jobs] Finish time fairness. The worst-case FTF ρ for *Shockwave* when scaling to large clusters is on average 1.32, outperforming fair scheduling policies Themis, Gavel, AlloX, and Gandiva-Fair by 2.5×, 2.4×, 3.1×, and 3.9× respectively. In addition, *Shockwave* maintains the fraction of unfairly scheduled jobs (FTF $\rho > 1$) on average at 4%, outperforming other fair scheduling baselines by 6×.

[Simulation - 64-256 GPUs, 220-900 Jobs] Average job

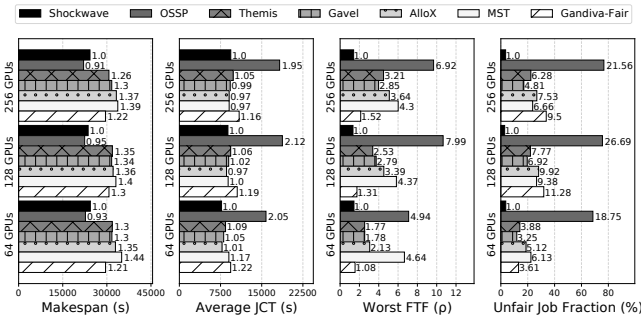


Figure 9: [Simulation] Evaluating *Shockwave*'s scheduling efficiency and fairness in differently sized large clusters.

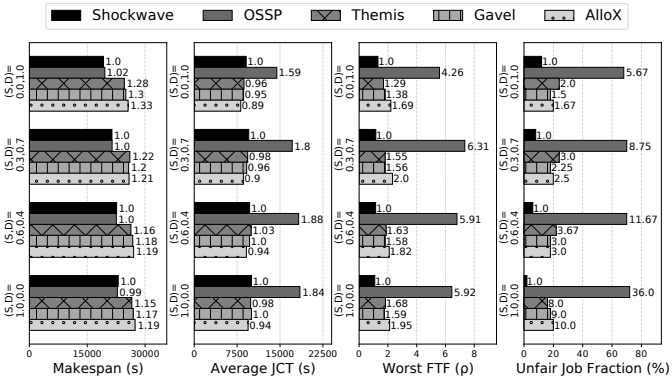


Figure 10: [Simulation] Effects of varying the mix of static and dynamic Jobs. (S, D)=(x, y) indicates x fraction of static jobs and y fraction of dynamic jobs.

completion time. At a large scale, *Shockwave* maintains similar responsiveness when compared with fair schedulers. One exception here is Gandiva-Fair which prolongs average JCT by 16-22%. Gandiva-Fair uses stride scheduling [39] where, by default, a job's number of tickets is equal to the job size (i.e., the number of workers). Thus, large jobs have a higher proportional share when compared with small jobs, and can delay small jobs, thereby degrading system responsiveness.

We study the solver overhead with large clusters in §8.9.

8.6 Benefits of Proactive Scheduling

We next compare *Shockwave* and baseline policies while varying the mix of static and dynamic jobs in simulation.

All static jobs. We first analyze the case where all jobs disable dynamic adaptation. This isolates *Shockwave*'s win due to social welfare maximization. The results (Figure 10) show that all fair scheduling policies, i.e., *Shockwave*, Themis, Gavel, and AlloX exhibit a relatively low fraction (<18%) of unfairly scheduled jobs (FTF ρ>1.0), but *Shockwave* outperforms the baseline algorithms by limiting the unfair fraction to less than 5%. *Shockwave* has on average an 18% improvement in makespan over Themis, Gavel, and AlloX, with no loss in average JCT. Overall, these results show how maximizing social welfare over time can achieve a better fairness-efficiency trade-off when compared to existing approaches.

Fairness and efficiency while being proactive. *Shockwave* sees a larger win in makespan as the fraction of dynamic jobs increases. The speedup over Gavel, Themis, and AlloX increases to 1.3× when the fraction of dynamic jobs grows from 0.4 to 1.0. Our results also show that existing schedulers that are reactive to dynamic scaling have suboptimal fairness outcomes. Both Themis and AlloX exhibit an increased unfair job fraction as the number of dynamic jobs increases. When all jobs are dynamic, Themis schedules 28% of jobs unfairly and AlloX schedules 22% of jobs unfairly, while *Shockwave* has a relatively (9%) low fraction of unfairly-scheduled jobs.

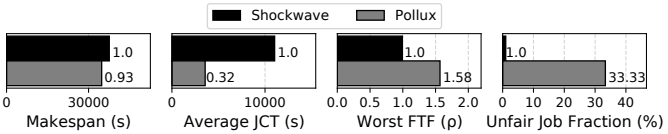


Figure 11: [Simulation] Evaluating *Shockwave*'s and Pollux's efficiency and fairness.

8.7 *Shockwave* versus Pollux

To compare the scheduling policies used by Pollux and *Shockwave*, we run both systems using the same workload trace provided by Pollux. We also first run the Pollux simulator to collect the batch size schedule observed at runtime and use that as an input to the *Shockwave* simulator. Thus, both systems see the same set of input jobs and the same batch size schedule, and hence, job processing times should match even with dynamic scaling.

JCT. From Figure 11, we see that Pollux has a 3× improvement in average JCT over *Shockwave*. Pollux can scale the number of workers of a job, which leads to reduced resource contention and improved responsiveness. In fact, we found that Pollux reduces the requested GPU hours per job by 2.4× when compared to the original trace. As our *Shockwave* prototype does not change the number of workers used by a job, it preserves the contention level in the trace (2.4× larger than Pollux) and thus exhibits inferior responsiveness. We note that as seen in Figure 7, *Shockwave* has comparable JCTs with other baselines and the Pollux paper [36] also reports a 3× speedup over the baselines.

Finish time fairness. *Shockwave* significantly outperforms Pollux w.r.t finish time fairness. This is because Pollux focuses on instantaneous fairness at each allocation but does not systematically address long-term fairness. At every round, Pollux's $p-norm$ formulation penalizes unfair allocations that lead to low instantaneous throughput for jobs but does not preserve long-term fairness over multiple allocation rounds. On the other hand, *Shockwave*'s dynamic market formulation provably guarantees long-term fairness.

Makespan. *Shockwave* benefits from optimizing for long-term efficiency and has a similar makespan as Pollux despite not changing the number of workers dynamically.

Finally, we note that as discussed in 2.3 and Appendix A.2, Pollux's approach of automatically tuning the batch size and the number of workers can lead to accuracy loss (e.g., 2% for

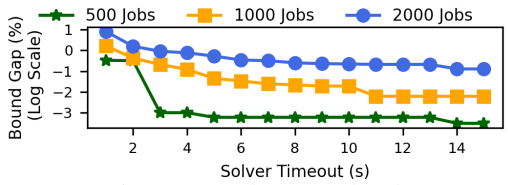


Figure 12: Solver Overhead.

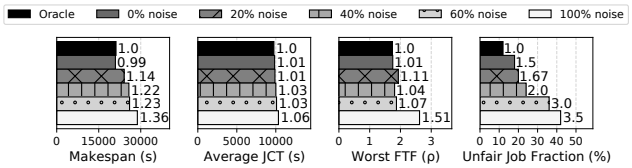


Figure 13: *Shockwave*'s scheduling efficiency and fairness under different levels of prediction errors.

ResNet18 and up to 4% for DeepSpeech [36]). We argue that Pollux's accuracy loss and poor fairness properties make it less attractive for practical deployments.

8.8 Varying Cluster Contention and Workload

We also vary the workload contention factor and compare all policies on a smaller 14-GPU physical cluster. We find that *Shockwave*'s fairness and efficiency win over the baseline schedulers increases (decreases) as cluster contention grows (drops). We include more details in Appendix I. We also compared *Shockwave* using arrival patterns from the Pollux [36] trace. Appendix J includes these results.

8.9 Solver Overhead

Shockwave uses a timeout knob (default 15s) to limit the overhead of solving our market formulation. Figure 12 uses simulation to show that on a 256 GPU cluster, the solver quality improves with diminishing returns as we increase the solver timeout from 1 second to 15 seconds. We measure solver quality using the bound gap (how far the solution found at the timeout is from the optimal). The relative bound gap at 15 seconds is small (0.03%, and 0.11%) for 500 and 1000 active jobs. The bound gap at 15 seconds for 2000 jobs increases to 0.44%. While this exceeds the criterion (0.1%) recommended by Gurobi [34], our results show a limited impact on efficiency and fairness. We note that our solver runs in a separate thread and is proactively invoked in the middle of the current round. Thus, the solver overhead is hidden when it is less than half-round duration.

8.10 Resilience to Prediction Error

Figure 13 shows *Shockwave*'s resilience to prediction errors when varying levels of random noises (i.e., ± p%) are injected into its interpolated job run time (under dynamic adaptation). The experiment settings in Figure 13 are similar to those in Figure 10 with all jobs enabled for dynamic batch size scaling (i.e., (S, D)=(0, 1.0)). First, we observe that as the injected errors grow, *Shockwave*'s worst-case FTF ρ and the fraction of unfairly scheduled jobs inflate slowly. A similar steady trend holds for *Shockwave*'s average JCT. We argue such robustness originates from the design principle of Nash social welfare, which emphasizes common ownership and fair sharing of cluster resources; the penalty is huge if skewed training progress is present in the cluster and it leads the scheduler to be conservative to jobs' interpolated schedule slacks that are predicated by the biased FTF estimates. Second, we find that *Shockwave*'s scheduling efficiency drops as the errors grow. *Shockwave* opportunistically prioritizes long-running jobs over the short ones to improve makespan. Having 100% injected noise affects *Shockwave*'s estimation of job length and lowers its scheduling efficiency by over 30%. Note that this deteriorated efficiency is still on par with the baseline schedulers (e.g., Themis, Gavel, and AlloX in Figure 10).

9 Related Work

We detail the comparison between *Shockwave* and existing schedulers (e.g., Gandiva [41], Optimus [35], DRF [17], REF [43], Themis [29], AlloX [28], Tiresia [21], Gandivar-Fair [10]) in Section 2, and spotlight *Shockwave*'s contribution from two angles. First, *Shockwave* is built on Nash social welfare, a theoretically-grounded approach to co-optimize long-term, rather than instantaneous, fairness and efficiency. Second, *Shockwave* proactively plans schedules for dynamic adaptation, while most existing schedulers only react to dynamic adaptation. Section 2 presented more details on the limitations of existing DL cluster schedulers.

AFS (Apathetic Future Share) [24] is another elastic sharing mechanism proactive to system dynamics. However, dynamic changes in AFS refer to job arrival and time-variant cluster contention, while jobs themselves do not change. *Shockwave* has a different focus: jobs' resource demands (and efficiency) dynamically change due to batch size scaling. Further, AFS primarily focuses on improving average JCT while *Shockwave* maximizes social welfare over time.

10 Conclusion

We presented *Shockwave*, a market-theory-based efficient and fair scheduling framework for DNN training workloads. We showed how existing schedulers fail to preserve fairness and degrade efficiency by being reactive to dynamic adaption. To address these challenges, we proposed a proactive approach that uses dynamic markets and Bayesian statistics for scheduling. Our experiments show that *Shockwave* can improve efficiency and fairness compared to state-of-the-art schedulers.

Acknowledgements: We would like to thank the anonymous reviewers and our shepherd Zhihao Jia for their constructive comments that helped improve our paper. We would also like to thank Zhao Zhang for helping us run experiments on TACC resources and Mosharaf Chowdhury for feedback on an earlier draft of this paper. This work was supported in part by a University of Wisconsin Fall Research Competition grant, by NSF grants CNS-2106199 and CNS-2105890 and by the CIFellows program, organized by the Computing Research Association and Computing Community Consortium.

References

[1] AGARWAL, S., WANG, H., LEE, K., VENKATARAMAN, S., AND PAPAILIOPOULOS, D. Adaptive gradient communication via critical learning regime identification. Proceedings of Machine Learning and Systems 3 (2021).

[2] ANGELOPOULOS, S., SARMA, A. D., MAGEN, A., AND VIGLAS, A. On-line algorithms for market equilibria. In International Computing and Combinatorics Conference (2005), Springer, pp. 596–607.

[3] AZAR, Y., BUCHBINDER, N., AND JAIN, K. How to allocate goods in an online market? In European Symposium on Algorithms (2010), Springer, pp. 51–62.

[4] BARABÃASI, A., ALBERT, R., AND JEONG, H. Mean-field theory for scale-free random networks. Physica A 272 (1999), 173–187.

[5] BRÂNZEI, S., CHEN, Y., DENG, X., FILOS-RATSIKAS, A., FREDERIKSEN, S., AND ZHANG, J. The fisher market game: Equilibrium and welfare. In Proceedings of the AAAI Conference on Artificial Intelligence (2014), vol. 28.

[6] BRANZEI, S., GKATZELIS, V., AND MEHTA, R. Nash social welfare approximation for strategic agents. In Proceedings of the 2017 ACM Conference on Economics and Computation (New York, NY, USA, 2017), EC '17, Association for Computing Machinery, p. 611–628.

[7] BROWN, T., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J. D., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., ET AL. Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.

[8] CARAGIANNIS, I., KUROKAWA, D., MOULIN, H., PROCACCIA, A. D., SHAH, N., AND WANG, J. The unreasonable fairness of maximum nash welfare. ACM Transactions on Economics and Computation (TEAC) 7, 3 (2019), 1–32.

[9] CAZES, J., EVANS, R. T., DUBROW, A., HUANG, L., LIU, S., AND MCLAY, R. Preparing frontera for texascale days. Computing in Science & Engineering (2021).

[10] CHAUDHARY, S., RAMJEE, R., SIVATHANU, M., KWATRA, N., AND VISWANATHA, S. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In Fifteenth European Conference on Computer Systems (EuroSys'20) (April 2020), ACM, pp. 1–16.

[11] CHIN, T.-W., DING, R., AND MARCULESCU, D. Adascale: Towards real-time video object detection using adaptive scaling. In Systems and Machine Learning Conference (2019).

[12] COFFMAN, JR, E. G., GAREY, M. R., AND JOHNSON, D. S. An application of bin-packing to multiprocessor scheduling. SIAM Journal on Computing 7, 1 (1978), 1–17.

[13] COLE, R., GKATZELIS, V., AND GOEL, G. Mechanism design for fair division: Allocating divisible items without payments. In Proceedings of the Fourteenth ACM Conference on Electronic Commerce (New York, NY, USA, 2013), EC '13, Association for Computing Machinery, p. 251–268.

[14] DELLA CROCE, F., AND SCATAMACCHIA, R. The longest processing time rule for identical parallel machines revisited. Journal of Scheduling 23, 2 (2020), 163–176.

[15] FIKIORIS, G., AGARWAL, R., AND TARDOS, É. Incentives in resource allocation under dynamic demands. arXiv preprint arXiv:2109.12401 (2021).

[16] FRANKLE, J., DZIUGAITE, G. K., ROY, D. M., AND CARBIN, M. Stabilizing the lottery ticket hypothesis. arXiv preprint arXiv:1903.01611 (2019).

[17] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (USA, 2011), NSDI'11, USENIX Association, p. 323–336.

[18] GONZALEZ, T., AND SAHNI, S. Open shop scheduling to minimize finish time. Journal of the ACM (JACM) 23, 4 (1976), 665–679.

[19] GORDON, G., AND TIBSHIRANI, R. Karush-kuhn-tucker conditions. Optimization 10, 725/36 (2012), 725.

[20] GRANDL, R., CHOWDHURY, M., AKELLA, A., AND ANANTHANARAYANAN, G. Altruistic scheduling in multi-resource clusters. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (Savannah, GA, Nov. 2016), USENIX Association, pp. 65–80.

[21] GU, J., CHOWDHURY, M., SHIN, K. G., ZHU, Y., JEON, M., QIAN, J., LIU, H., AND GUO, C. Tiresias: A {GPU} cluster manager for distributed deep learning. In 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19) (2019), pp. 485–500.

[22] HE, X., LIAO, L., ZHANG, H., NIE, L., HU, X., AND CHUA, T.-S. Neural collaborative filtering. In Proceedings of the 26th International Conference on World Wide Web (Republic and Canton of Geneva, CHE, 2017), WWW '17, International World Wide Web Conferences Steering Committee, p. 173–182.

[23] HOFFER, E., HUBARA, I., AND SOUDRY, D. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. In Proceedings of the 31st International Conference on Neural Information Processing Systems (Red Hook, NY, USA, 2017), NIPS'17, Curran Associates Inc., p. 1729–1739.

[24] HWANG, C., KIM, T., KIM, S., SHIN, J., AND PARK, K. Elastic resource sharing for distributed deep learning. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21) (2021), pp. 721–739.

[25] JEON, M., VENKATARAMAN, S., PHANISHAYEE, A., QIAN, U., XIAO, W., AND YANG, F. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (2019), USENIX ATC '19, p. 947–960.

[26] KESKAR, N. S., MUDIGERE, D., NOCEDAL, J., SMELYANSKIY, M., AND TANG, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. The International Conference on Learning Representations (ICLR) (2017).

[27] KUSHNER, H. J., AND WHITING, P. A. Convergence of proportional-fair sharing algorithms under general conditions. IEEE transactions on wireless communications 3, 4 (2004), 1250–1259.

[28] LE, T. N., SUN, X., CHOWDHURY, M., AND LIU, Z. Allox: Allocation across computing resources for hybrid cpu/gpu clusters. SIGMETRICS Perform. Eval. Rev. 46, 2 (Jan. 2019), 87–88.

[29] MAHAJAN, K., BALASUBRAMANIAN, A., SINGHVI, A., VENKATARAMAN, S., AKELLA, A., PHANISHAYEE, A., AND CHAWLA, S. Themis: Fair and efficient GPU cluster scheduling. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20) (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 289–304.

[30] MAI, L., LI, G., WAGENLÄNDER, M., FERTAKIS, K., BRABETE, A.-O., AND PIETZUCH, P. Kungfu: Making training in distributed machine learning adaptive. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20) (Nov. 2020), USENIX Association, pp. 937–954.

[31] MCCANDLISH, S., KAPLAN, J., AMODEI, D., AND TEAM, O. D. An empirical model of large-batch training. CoRR abs/1812.06162 (2018).

[32] NACE, D., AND PIÓRO, M. Max-min fairness and its applications to routing and load-balancing in communication networks: A tutorial. IEEE Communications Surveys & Tutorials 10, 4 (2008), 5–17.

[33] NARAYANAN, D., SANTHANAM, K., KAZHAMIAKA, F., PHANISHAYEE, A., AND ZAHARIA, M. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20) (Nov. 2020), USENIX Association, pp. 481–498.

[34] PEDROSO, J. P. Optimization with gurobi and python. INESC Porto and Universidade do Porto,, Porto, Portugal 1 (2011).

[35] PENG, Y., BAO, Y., CHEN, Y., WU, C., AND GUO, C. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In Proceedings of the Thirteenth EuroSys Conference (New York, NY, USA, 2018), EuroSys '18, Association for Computing Machinery.

[36] QIAO, A., CHOE, S. K., SUBRAMANYA, S. J., NEISWANGER, W., HO, Q., ZHANG, H., GANGER, G. R., AND XING, E. P. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21) (July 2021), USENIX Association, pp. 1–18.

[37] QIN, H., RAJBHANDARI, S., RUWASE, O., YAN, F., YANG, L., AND HE, Y. Simigrad: Fine-grained adaptive batching for large scale training using gradient similarity measurement. Advances in Neural Information Processing Systems 34 (2021).

[38] SMITH, S. L., KINDERMANS, P.-J., AND LE, Q. V. Don't decay the learning rate, increase the batch size. In International Conference on Learning Representations (2018).

[39] WALDSPURGER, C. A. Lottery and stride scheduling: Flexible proportional-share resource management. PhD thesis, Massachusetts Institute of Technology, 1995.

[40] WIERMAN, A., AND HARCHOL-BALTER, M. Classifying scheduling policies with respect to unfairness in an m/gi/1. In Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (2003), pp. 238–249.

[41] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., YANG, F., AND ZHOU, L. Gandiva: Introspective cluster scheduling for deep learning. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (USA, 2018), OSDI'18, USENIX Association, p. 595–610.

[42] YAO, Z., GHOLAMI, A., LEI, Q., KEUTZER, K., AND MAHONEY, M. W. Hessian-based analysis of large batch training and robustness to adversaries. Advances in Neural Information Processing Systems 31 (2018).

[43] ZAHEDI, S. M., AND LEE, B. C. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, Feb. 2014), p. 145–160.

[44] ZAHEDI, S. M., LLULL, Q., AND LEE, B. C. Amdahl's law in the datacenter era: A market for fair processor allocation. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA) (2018), IEEE, pp. 1–14.

A Dynamic Batch Scaling Degrades Accuracy

A.1 When does batch size scaling degrade accuracy

Improper batch size scaling can adversely affect convergence and degrade the accuracy of the trained model. This is known as **generalization gap** [23, 26] and its underlying reasons are still not well understood. We next list different analyses on when batch size scaling can affect final model quality: (a) scaling up the batch size by $k\times$ reduces the number of iterations per epoch by $k\times$, and given a pre-specified number of epochs, it reduces the overall iterations of back-propagation by $k\times$. Accuracy loss stems from a reduced number of model updates [23]. (b) Scaling up the batch size reduces the noise in the gradient estimate, but noise serves to regularize training and can navigate the optimizer away from local minima. Batch size scaling thus hurts generalization by reducing healthy gradient noises. (c) Scaling up the batch size causes training to converge to sharp minima, and the model outputs are sensitive to small perturbations in the input. This results in a poorer generalization [26].

Researchers have developed heuristics [1, 38] and adaptive batch size scaling techniques (e.g., Gradient Norm [1], GNS (Gradient Noise Scale) [31] and Heissan Eigenspectrum [42]) to mitigate generalization gap, but no single technique handles all models, datasets and optimizers. Thus, today, there are many different batch size scaling techniques in the ML community. Pollux adopts GNS while recent work points out some of the limitations in applying GNS [37] for batch size scaling.

A.2 Example: Pollux's automatic batch size scaling leads to accuracy loss in NeuMF-m1-lm training

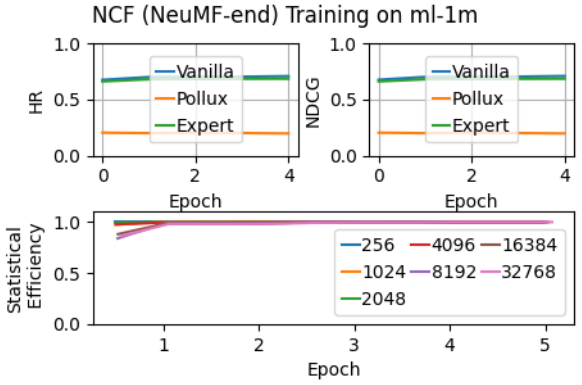


Figure 14: Comparing model accuracy (NCF-ml-1m) for vanilla training (no batch size scaling), expert-set batch size scaling, and Pollux's autoscaling. The legends in the bottom figure indicate batch size.

Figure 14 shows that statistical efficiency minimally degrades when scaling up from a batch size of 256 to 32768, and that this is true even for early training epochs. Therefore, Pollux immediately scales up the batch size from 256 to 32768 at epoch 1. However, we found that such early, aggressive scaling leads to inferior validation accuracy, i.e. lower HR (Hit Rate) and NDCG (Normalized Discounted Cumulative Gain), when compared with vanilla training where dynamic batch size scaling is disabled. An expert-set dynamic scaling schedule that scales up the batch size to 32768 at epoch 3 and this helps match the validation accuracy of vanilla training.

B Static Filters Degrade Efficiency, Fairness

GPU ID \ round ID	0	1	2	3	4	5	6
0	A	A	A	B	A	A	A
1	B	B	A	B	A	A	A
2	C	B	A	C	A	B	
3	C	C	C	C	B	B	
f	1/3	1/3	1/3	1/3	1/3	1/3	1/3

(a) Themis with $f = 1/3$.

GPU ID \ round ID	0	1	2	3	4	5	6
0	A	A	A	A	A	A	A
1	A	B	B	A	B	A	A
2	B	B	C	B	B	A	
3	C	C	C	C	C	B	
f	1.0	1.0	1.0	1.0	1.0	1.0	1.0

(b) Themis with $f = 1.0$.

GPU ID \ round ID	0	1	2	3	4	5	6
0	B	B	B	A	A	A	A
1	B	B	B	A	A	A	A
2	C	C	C	A	A	A	A
3	C	C	C	B	B		
f	2/3	2/3	2/3	1/3	1/3	1/3	1/3

(c) *Shockwave* with a dynamic filter f.

Figure 15: Visualizations of schedules produced by using different filters in Table 1. The cluster and job setting are the same as those in Figure 1.

C Volatile Fisher Market (VFM)

C.1 Market formulation

VFM is a dynamic market and continues through rounds indexed by $t = 1, \ldots, T$. Each round is a fixed time interval, e.g., 120s. Within each round, a central seller (i.e., the scheduler) sells multiple types of resources (e.g., GPUs and/or CPUs) to buyers (i.e., the contending jobs). [5] All resources are volatile. Resources bought by a job in round t cannot be carried over to future rounds. There is a dynamic price for each resource type in each round, and each job is endowed with an initial budget to spend across rounds. The amount of endowment reflects the priority of jobs. VFM assumes divisible resources [8].

(a) Buyers, Seller, and Resources. There exist N buyer jobs competing for J different types of resources. **(b) Allocation (Purchase).** Let x_{ijt} denote the allocation (purchase) of job i for resource j in round t. The resource provision is normalized to one unit. For brevity, let $\boldsymbol{x}_{it}$ denote the allocation vector $[\ldots, x_{ijt} \ldots]$ for job i in round t, and let $\boldsymbol{X}_{\boldsymbol{i}}$ denote the $J \times T$ allocation matrix for job i over rounds, with rows and columns corresponding to the resource types and round indices. **(c)**

[5] VFM supports multiple-resource allocation, but evaluation in this paper is carried out only for GPU allocation.

Budget, Price, and Payment. Job i is endowed with a budget B_i to spend over rounds. The price for resource j in round t is p_{jt}; The accrued payment over rounds for job i is $\sum_{j,t} p_{jt} x_{ijt}$. Let $\boldsymbol{P}$ denote a $J \times T$ price matrix, with $\boldsymbol{P}[j,t] = p_{jt}$. **(d) Performance (Utility) Function for Dynamic Adaptation.** We use the performance (utility) function $U_{it}(\boldsymbol{x}_{it})$ to map received resources, that is, $\boldsymbol{x}_{it}$, to the performance gain of job i (e.g., epoch progress). Note that U_{it} can be different over rounds to model time-variant performance under dynamic adaptation. We limit performance functions in the CES family [5], which are extensively used in system research.[6]

C.2 Solving Equilibrium of VFM

The market equilibrium captures the optimal allocation for the N jobs in each round, i.e. $\boldsymbol{X}_1^*, \ldots, \boldsymbol{X}_N^*$, and the optimal prices $\boldsymbol{P}^*$ for different types of resources in each round. An equilibrium is established if the following two properties are satisfied. **(a) Maximized Performance Under Budget Constraint (Optimal Spending)**: Each job's performance is maximized under budget constraints. $\boldsymbol{X}_i^* = \arg\max_{\boldsymbol{X_i}} U_i(\boldsymbol{X_i}), s.t., \sum_t \sum_j p_{jt} x_{ijt} \leq B_i, \ \forall i$. **(b) Work-conserving (Market Clearing)**: There is no leftover resource if the price for the resource type is non-zero. That is, if $p_{jt} > 0$, then $\sum_i x_{ijt} = 1, \ \forall j,t$.

Theorem C.1. *For Volatile Fisher Market with linear or Leontief (e.g., DRF [17]) utility, the solution of (3) captures the optimal allocation in the market equilibrium and the Lagrangian dual to capacity constraints (i.e., $\sum_i x_{ijt} \leq 1, \ \forall j,t$) captures the equilibrium price.*

$$\begin{aligned} &\underset{\boldsymbol{X}_1,\ldots,\boldsymbol{X}_N}{Maximize} \sum_i B_i \log U_i(X_i) \ \ s.t., \ U_i(X_i) = \sum_t u_{it}(\boldsymbol{x}_{it}), \forall i, \\ &\begin{cases} u_{it}(\boldsymbol{x}_{it}) = \sum_j u_{ijt} x_{ijt}, \forall i,t \text{ (Linear)} \\ u_{it}(\boldsymbol{x}_{it}) = min_j \frac{x_{ijt}}{a_{ijt}}, \forall i,t \text{ (Leontief)} \end{cases}, \\ &\sum_i x_{ijt} \leq 1, \ \forall j,t, \ \ x_{ijt} \geq 0, \ \forall i,j,t \end{aligned} \tag{3}$$

For VFM with linear and Leontief performance function at each instant, Theorem C.1 states that the solution of an Eisenberg-Gale [5] styled program defined in (3) captures the market equilibrium (cf., proof in Appendix D). **Note that even if the instantaneous utility is Leontief, the summed utility over time is not Leontief in general.**

D Proof of Theorem C.1

D.1 Linear Utility

Volatile Fisher Market (VFM) with linear utilities reduces to a special case of static Fisher market, if we consider volatile resource j at each different time t a unique type of resource. Upon substituting the tuple of resource and time index (j,t) with a new resource index k (i.e., $(j,t) \rightarrow k$), the Eisenberg-Gale program defined in (3) is equivalent to the program (4).

$$\begin{aligned} &\underset{\boldsymbol{x}}{Maximize} \sum_i B_i \log u_i(\boldsymbol{x_i}) \ \ s.t. \\ &u_i(\boldsymbol{x_i}) = \sum_k u_{ik} x_{ik} \\ &\sum_i x_{ik} \leq 1, \ \forall k \\ &x_{ik} \geq 0, \ \forall i,k \end{aligned} \tag{4}$$

Existing work [5] has proven that program (4) captures the market equilibrium of a static Fisher market, and thus, it also captures the market equilibrium of the equivalent VFM.

D.2 Leontief Utility

However, VFM with Leontief utilities has no direct link to the classic static Fisher market. We prove the Eisenberg-Gale (EG) program defined in (3) captures market equilibrium by characterizing the Karush–Kuhn–Tucker (KKT) [19] conditions. We rewrite (3) in a standard convex optimization form and number the constraints as follows:

$$\underset{\boldsymbol{X}_1,\ldots,\boldsymbol{X}_N}{Minimize} -\sum_i B_i \log U_i(X_i) \ \ s.t. \tag{5a}$$

$$U_i(X_i) \leq \sum_t u_{it}(\boldsymbol{x}_{it}), \ \forall i \tag{5b}$$

$$u_{it}(\boldsymbol{x}_{it}) \leq x_{ijt}/a_{ijt}, \ \forall i,j,t \tag{5c}$$

$$\sum_i x_{ijt} \leq 1, \ \forall j,t \tag{5d}$$

$$x_{ijt} \geq 0, \ \forall i,j,t \tag{5e}$$

Let $\beta_i, \lambda_{it}, p_{jt}, \eta_{ijt}$ denote the Lagrangian multipliers corresponding to constraints (5b), (5c), (5d),(5e), respectively. The Lagrangian dual function is

$$\begin{aligned} L(x, \boldsymbol{\beta}, \boldsymbol{\lambda}, \boldsymbol{p}, \boldsymbol{\eta}) = &-\sum_i B_i \log U_i(X_i) + \\ &\sum_{i,j,t} (u_{it}(\boldsymbol{x}_{it}) - \frac{x_{ijt}}{a_{ijt}}) \lambda_{ijt} \\ &+ \sum_i (U_i(\boldsymbol{X_i}) - \sum_t u_{it}(\boldsymbol{x}_{it})) \beta_i + \\ &\sum_{j,t} (\sum_i x_{ijt} - 1) p_{jt} - \sum_{i,j,t} x_{ijt} \eta_{ijt} \end{aligned}$$

First, KKT requires a first-order condition of the Lagrangian function; the gradients to all primal variables and Lagrangian multipliers should be zero. This implies that

$$\frac{\partial L}{\partial U_i(\boldsymbol{X_i})} = 0 \implies -\frac{B_i}{U_i(\boldsymbol{X_i})} + \beta_i = 0 \implies \beta_i = \frac{B_i}{U_i(\boldsymbol{X_i})}$$

$$\frac{\partial L}{\partial u_{it}(\boldsymbol{x}_{it})} = 0 \implies \sum_j \lambda_{ijt} - \beta_i = 0 \implies \beta_i = \sum_j \lambda_{ijt}$$

$$\frac{\partial L}{\partial x_{ijt}} = 0 \implies -\frac{\lambda_{ijt}}{a_{ijt}} + p_{jt} - \eta_{ijt} = 0$$

[6] Themis [29] and Gavel [33] uses linear utility, Dominant Resource Fairness (DRF) [17] uses Leontief utility, and REF (Resource Elasticity Fairness) [43] uses Cobb-Douglas utility, which are all CES utility functions.

The combination of the first two equations implies that

$$\sum_j \lambda_{ijt} = \frac{B_i}{U_i(\boldsymbol{X_i})},\ \forall i,t$$

Lagrangian multipliers are nonnegative, and thus, $\eta_i \geq 0$ implies

$$p_{jt} a_{ijt} \geq \lambda_{ijt}$$

Combining the last equation and the last inequality implies that

$$\sum_j p_{jt} a_{ijt} \geq \sum_j \lambda_{ijt} = \frac{B_i}{U_i(\boldsymbol{X_i})},\ \forall i,t$$

$$\implies U_i(\boldsymbol{X_i}) \geq \frac{B_i}{\sum_j p_{jt} a_{ijt}},\ \forall i,t$$

$$\implies U_i(\boldsymbol{X_i}) = \frac{B_i}{min_{t'}\{\sum_j p_{jt'} a_{ijt'}\}},\ \forall i$$

This states that for any job i, its overall utility is achieved by purchasing resources only in certain time periods that guarantee **MBB (Maximal Bang-Per-Buck)** [5] for the job, where $\sum_j p_{jt} a_{ijt}$ represents the unit cost to obtain one unit of utility. MBB guarantees that any job's utility accrued over time is maximized given a fixed budget B_i.

We have proved optimal spending (i.e., maximized utility under budget limit) for each job at the solution of program (D). The last step is to prove market clearing. KKT conditions also require complementary slackness:

$$p_{jt}(\sum_i x_{ijt} - 1) = 0,\ \forall j,t \implies$$

$$\text{if } p_{jt} > 0 \text{ then } \sum_i x_{ijt} = 1,\ \forall j,t$$

This implies that if a resource j is traded in time period t, then it must be exhaustively allocated to the jobs and the amount of leftover resources is zero. Therefore, we prove **MC (Market Clearing)**. An extra fact implied by the solution of the EG program (D) is that since the objective is to maximize social welfare (budget-weighted geometric mean of jobs' utilities), and thus, there should be no money left by a job in the solution of (D). This proves **BC (Budget Clearing)**. That is, the budget for all jobs will be completely burnt over periods.

In summary, proving Maximal Bang-Per-Buck, Market Clearing and Budget Clearing establishes the VFM market equilibrium produced by solving program (5).

E Proof of Theorem 4.0.1

Proof of (a): $\prod_i \rho_i = \prod_i U_i(\frac{C}{N}) \cdot \prod_i U_i(\frac{\boldsymbol{X_i}}{N})^{\frac{-B_i}{B}} = \prod_i U_i(\frac{C}{N})$ $\mathsf{NSW_{OT}}^{-1}$. Since $\prod_i U_i(C/N)$ is a constant independent of X_i, VFM equilibrium that maximizes $\mathsf{NSW_{OT}}$ equivalently minimizes the product of FTF (Finish Time Fairness) metrics over all jobs, i.e., $\prod_i \rho_i$.

Proof of (b). At VFM equilibrium, any job i has maximized utility under budget. When all job have an equal budget, job i will not prefer any other jobs j's allocation, since job i can afford to buy any other job's allocation under same budget. Formally, we get that $U_i(\boldsymbol{X}_i) \geq U_i(\boldsymbol{X}_j), \forall i,j$. Since the market clears in VFM equilibrium, it is not possible that all jobs have a strictly smaller resource share than C/N, and there must exist a job k such that its resource share is greater than or equal to C/N, then we know $U_i(\boldsymbol{X}_j) \geq U_i(\boldsymbol{X}_k) \geq U_i(\frac{C}{N})$, and thus, Finish Time Fairness if proved.

F Stochastic Dynamic Program for Efficiency and Fairness in Expectation

(a) State. Each job has a private, finite set of states. For dynamic scaling of the batch size, a state is a tuple (BatchSize, Epoch), which denotes the current batch size and the current epoch (index). Let s_{it} ($\boldsymbol{s}_t$) denote the state of job i in round t. **(b) Policy.** Let $\boldsymbol{x}_{it}$ ($\boldsymbol{x}_t$) denote the resource allocated to job i in round t. An allocation policy $\pi(\boldsymbol{s}_t, \boldsymbol{x}_t)$ indicates the probability of making allocation $\boldsymbol{x}_t$ to the jobs, conditional on job states $\boldsymbol{s}_t$, in round t. We further limit π to be a deterministic policy in this study. **(c) Transition Probability.** We model the state transition with a probability matrix $P_i(s_{it+1}|s_{it}, \boldsymbol{x}_{it})$, which indicates the probability of job i transitioning to state s_{it+1} from state s_{it+1}, under resource allocation x_{it}. Let $P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{x}_t)$ denote the transition probabilities for all the jobs. **(d) Performance (Utility) Function for Dynamic Adaptation.** Let $U_i(s_{it}, \cdot, s_{it+1})$ denote the performance gain (e.g., epoch progress) of job i when transitioning from state s_t to the next state s_{t+1}. Let $U_i(\boldsymbol{s_t}, \cdot, \boldsymbol{s}_{t+1})$ denote the performance function for all jobs.

Maximized Nash social welfare in expectation. We construct a linear program in (6) to search for an optimal policy that maximizes Nash social welfare in expectation, i.e., NSW_{OTE}. Maximized NSW_{OTE} co-optimizes efficiency and fairness in expectation sense. The first constraint in (6) defines expected cumulative utility under the policy; The second constrains the summed allocation at each period t not exceeding resource provision, and allocation should be non-negative. The third constrains valid probability transition between states. Other constraints are omitted.

$$\pi^* = \underset{\pi}{argmax} \sum_i B_i \, log\, \mathbb{E}_\pi[U_i],\ s.t. \quad (6)$$

$$\mathbb{E}_\pi[U_i] = \sum_{t=1}^{T-1} \sum_{\boldsymbol{s_t} \in S} \sum_{\boldsymbol{x_t} \in X} \sum_{\boldsymbol{s}_{t+1} \in S} [\pi(\boldsymbol{s_t}, \boldsymbol{x_t}) \cdot P(\boldsymbol{s}_{t+1}|\boldsymbol{s_t}, \boldsymbol{x_t}) \cdot U_i(\boldsymbol{s_t}, \boldsymbol{x_t}, \boldsymbol{s}_{t+1})],\ \pi(\boldsymbol{s_t}, \boldsymbol{x_t}) \geq 0 \text{ and } \pi(\boldsymbol{s_t}, \boldsymbol{x_t}) = 0 \text{ if } ||x_t||_{\ell_1} > 1 \text{ or } x_t < 0,\ \forall \boldsymbol{s_t} \in S, \forall \boldsymbol{x_t} \in X, t = 1, \ldots, T$$

$$\sum_{\boldsymbol{x_1}} \pi(\boldsymbol{s_1}, \boldsymbol{x_1}) = b(\boldsymbol{s_1}),$$

$$\sum_{\boldsymbol{x_t}} \pi(\boldsymbol{s_t}, \boldsymbol{x_t}) = \sum_{\boldsymbol{s}_{t-1}} \sum_{\boldsymbol{x}_{t-1}} P(\boldsymbol{s_t}|\boldsymbol{s}_{t-1}, \boldsymbol{x}_{t-1}) \pi(\boldsymbol{s}_{t-1}, \boldsymbol{x}_{t-1}),$$

$$t = 2, \ldots, T$$

G *Shockwave* Design Details

Shockwave plans the schedule for a configurable number (T) of future rounds (default T: 30 two-minute rounds) and re-

computes the schedule when the planned rounds elapse or when jobs arrive or complete. *Shockwave*'s solved schedule is a $N \times T$ binary matrix $\mathsf{X}[j,t]$. N is the total number of active jobs available for scheduling. $\mathsf{X}[j,t] = 1$ ($\mathsf{X}[j,t] = 0$) represents scheduling (descheduling) job J_j in round t ($t = 1, \ldots, T$). We next describe the logic in one shot of schedule solving.

Decomposing job schedules to regime schedules. If dynamic adaptation is predicted to occur within the future planning window, the scheduler must incorporate dynamic changes of jobs' throughputs when solving the schedule.

Example - A job's dynamic adaptation process has two regimes. The job is currently in the first regime at epoch 5 and the scheduler predicts that the second regime will start from epoch 15. Suppose the planning window is 30-minute long and the epoch duration for the first and second regime are 2 minutes and 1 minute, respectively. Then dynamic adaptation can start as early as the 20th minute in the window, and a 2× change in throughput should be concerned.

To support dynamic changes in job throughputs, we decompose a job's schedule into its regimes' schedules, such that each regime is a micro-job with static throughput. In our above example, epochs 5 to 14 will be one micro-job while epoch 15 onward will be the second micro-job. We build a $K \times T$-dimensional binary matrix $\mathsf{Y}_j[k,t]$ to represent the schedule of job J_j's K regimes that can fit in the planning window. $\mathsf{Y}_j[k,t] = 1$ ($\mathsf{Y}_j[k,t] = 0$) indicates scheduling (descheduling) the k-th regime of job J_j to the t-th round in the window. Note that partial order constraints are needed to preserve the sequential order between regimes.

G.1 Implementing Nash Social Welfare over Time

We compute the utility of job J_j under schedule $\mathsf{Y}_j[\cdot,\cdot]$ as:

$$\mathsf{UTIL}_j(\mathsf{Y}_j[\cdot,\cdot]) = \frac{F_j}{E_j} + \sum_{t=1}^{T}\sum_{k=1}^{K} \frac{\mathsf{Y}_j[k,t] \cdot D \cdot \mathsf{TH}(j,k)}{Q_j \cdot E_j} \quad (7)$$

Job J_j's utility equals its current epoch progress percentage (num. finished epochs F_j divided by total num. epochs E_j), plus the resulting epoch progress percentage under allocation; the latter sums up the progress percentages for the job across regimes and rounds in the window.[7]. At the cluster level, the (logarithm of) Nash social welfare over time ($\mathsf{NSW}_{\mathsf{OT}}$) integrates the utilities of individual jobs. $\mathsf{Y}[\cdot,\cdot,\cdot]$ is a three-dimensional array that includes the schedule variable for all active jobs' regimes, at all rounds, in the planning window. As stated in §4.2, maximizing $\mathsf{NSW}_{\mathsf{OT}}$ yields an equilibrium and establishes efficiency and fairness guarantees.

$$\mathsf{WELFARE}(\mathsf{Y}[\cdot,\cdot,\cdot]) = \sum_{j=1}^{N} log\, \mathsf{UTIL}_j(\mathsf{Y}_j[\cdot,\cdot]) \quad (8)$$

[7]The epoch progress at a single round t for the k-th regime equals the duration of each round (D_j) times if the regime is scheduled to the round ($\mathsf{Y}[k,t]$), then divided by epoch duration $\mathsf{Q}(j)/\mathsf{THPT}(j,k)$

G.2 Implementing Estimators for Long-Term Effects

As previously stated, maximizing social welfare for an (infinitely) long time horizon is difficult due to prohibitive computational overhead and limited predictability. Another reason is that jobs arrive and complete online, and frequent replanning is unavoidable. In practice, *Shockwave* only plans the schedule for a finite length window (e.g, 30-60 minutes), and we design estimators that can capture the long-term fairness and long-term efficiency that arise from short-term planning.

An estimator for long-term fairness.

$$\widehat{\rho}(j) = \frac{L_j + W_j + \widehat{\mathsf{R}}(\mathrm{j}) \cdot \mathsf{N}_{\mathsf{avg}}(j)}{\widehat{\mathsf{P}}(\mathrm{j}) \cdot \mathsf{N}_{\mathsf{avg}}(j)} \quad (9)$$

We estimate the finish time fairness (FTF) $\widehat{\rho}(j)$ of job J_j as its predicted job completion time (the sum of attained service time L_j, waiting time W_j, and the interpolated remaining run time $\widehat{\mathsf{R}}(\mathrm{j})N_{avg}(j)$), divided by its predicted job run time ($\widehat{\mathsf{P}}(\mathrm{j})N_{avg}(j)$).

$\widehat{\mathsf{P}}(\mathrm{j})$ ($\widehat{\mathsf{R}}(\mathrm{j})$=$\widehat{\mathsf{P}}(\mathrm{j}) - L_j$) is the total (remaining) run time under isolated resources predicted using the Bayesian posterior. Similarly to prior work [29], we linearly scale the isolated run time with a contention factor $N_{avg}(j)$ to compute the run time under contention. In this paper, we define the contention factor as, within a fixed time range, the ratio between the number of jobs requesting GPUs and the overall number of GPUs provisioned in the cluster, and a job's contention factor $N_{avg}(j)$ only accounts for the time range it is either queued or running.

Shockwave plugs in the k-th power of FTF ρs of jobs into social welfare function (see Equation 11) as weights. The weights in the social welfare function act as the budgets assigned to jobs in the volatile Fisher market. If a job is predicted to be unfairly scheduled (large FTF ρ) in the long term, VFM correspondingly assigns a higher budget for it and proactively prioritizes the job in the planning window.

An estimator for long-term efficiency. The efficiency estimator estimates the final makespan to complete all current jobs' training epochs and penalizes schedules (in the planning window) that potentially increase the makespan estimate. However, the final makespan is unknown at the current instant and, in practice, *Shockwave* penalizes increasing the lower bound of it. *Shockwave* uses the lower bound given in [12]. Let $\mathsf{R}(\mathsf{Y}_j[\cdot,\cdot])$ denote the remaining run time of job J_j from the planning window. The lower bound of makespan (for the remaining epochs) is estimated as the maximum between the sum of the remaining run time divided by the number of GPUs in the cluster (i.e., M), and the longest remaining run time among jobs. Intuitively this takes the maximum between the longest job remaining and the makespan if all remaining jobs were evenly spread out across the cluster.

$$\mathsf{H}(\mathsf{Y}[\cdot,\cdot,\cdot]) = max\{\frac{\sum_j \mathsf{R}(\mathsf{Y}_j[\cdot,\cdot])}{M}, max_j \mathsf{R}(\mathsf{Y}_j[\cdot,\cdot])\} \quad (10)$$

Finally, we plug in the long-term efficiency estimator to so-

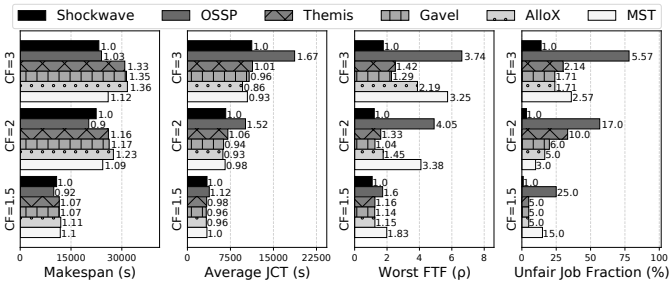


Figure 16: [Physical] Evaluating the scheduling efficiency and fairness of *Shockwave* under different contention factors (CF) in a 14-GPU physical cluster.

cial welfare maximization as a regularizer (See Equation 11). λ is a tunable coefficient that controls the degree of regularization. *Shockwave* yields similar makespan and fairness for different workloads when λ is between 1e-1 and 1e1.

G.3 An End-to-End Schedule Optimizer

Finally, Equation 11 shows the optimization problem solved by *Shockwave* in a given round. The output of the solver is schedule for each regime and the job schedule for the round can be simply translated from the regime schedule.

$$\underset{Y_1,\ldots,Y_N}{\text{Maximize}} \frac{1}{NM}\sum_{j=1}^{N}\rho(j)^k log[\,\mathsf{UTIL}_j(\mathsf{Y_j}[\cdot,\cdot])] - \frac{\lambda}{Z_0}\mathsf{H}(\mathsf{Y}_1[\cdot,\cdot],\ldots,\mathsf{Y_N}[\cdot,\cdot]) \tag{11}$$

Z_0 is a normalization coefficient, which is the sum of the interpolated run time across all jobs. More details about the constraints can be found in Appendix H.

Handling dynamic job arrival. Similar to existing schedulers, such as Themis [29], Tiresias [21], Pollux [36] and Gavel [33], *Shockwave* periodically adds newly arriving jobs to the schedule solver (Equation 11). The fairness objective in *Shockwave* (Equation 9) automatically handles selecting between newly-arrived short jobs or jobs that have been waiting in the queue for a long time, according to their pressure on breaking finish time fairness.

H Constraints Of Program 11

Program 11 requires the following constraints (details omitted). *(1) Preserving the order of regimes.* Any regime is prohibited to run before precedent regimes are complete. *(2) Work-conserving (Market Clearing).* Idle resources are not allowed when there are ready jobs. *(3) Capacity Limits.* GPUs assigned to jobs should not exceed the overall provision.

I Varying Contention Factor

We define the contention factor as, within a fixed time range, the ratio between the number of jobs requesting GPUs and the overall number of GPUs provisioned in the cluster. A larger contention factor indicates more jobs competing GPU resources at an instant. So far, we have assumed a default contention factor (three). We next vary the contention factor and compare policies on a smaller 14-GPU physical cluster.

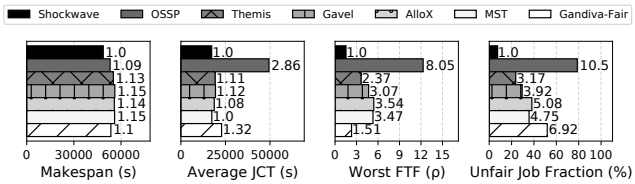


Figure 17: [Simulation] Evaluating *Shockwave*'s scheduling efficiency and fairness for Pollux trace on a 32-GPU cluster.

Shockwave's win in efficiency decreases as there is more resource slack and less contention in the cluster. *Shockwave*'s improvement in makespan over Gavel, AlloX, and Themis decreases (from 35% for contention factor 3) to 19% (8%) when the contention factor is lowered to 2 (1.5) (cf. Figure 16). A similar trend for cluster utilization is found. *Shockwave*'s improvement in cluster utilization drops to 19% (5%). Although the finish time fairness of all policies improves as the contention factor decreases, *Shockwave* still performs better than the baselines. *Shockwave* keeps the fraction of unfairly scheduled jobs (i.e., the fraction of jobs with FTF ρ>1) low when varying the contention factor. The average fraction of unfairly scheduled jobs for *Shockwave* is 8.67% when varying the contention factors, outperforming the baselines by 2.85× (see Figure 16). When the contention factor is lowered to 2, *Shockwave* maintains a worst-case FTF ρ of 1.2, outperforming Themis, Gavel, and AlloX by 1.27×. When the contention factor is further lowered to 1.5, *Shockwave* and all the baselines worst-case FTF approach 1 and the difference is insignificant.

J Varying the Cluster Trace

In previous subsections, we presented the results of *Shockwave* using synthetic traces generated by the Gavel [33] workload generator. In this subsection, we extend the evaluation using real DNN training traces provided by the Pollux [36] system. The Pollux trace provides the duration and arrival timestamps for training jobs and is extracted from a previous workload analysis [25]. Figure 17 shows the comparison between *Shockwave* and the baseline algorithms and we can see a similar trend as in previous sections. However, the win in makespan over Themis, Gavel, and AlloX drops from 30-35% to 20% on the Pollux trace. In previous synthetic traces, the duration of jobs has a greater diversity (2×) than in the Pollux trace, and thus long-running jobs have a larger impact on final makespan and cluster utilization. Therefore, opportunistically prioritizing these long-running jobs leads to greater improvement when there is more diversity among jobs.

Protego: Overload Control for Applications with Unpredictable Lock Contention

Inho Cho[1] Ahmed Saeed[2] Seo Jin Park[1] Mohammad Alizadeh[1] Adam Belay[1]
[1]MIT CSAIL [2]Georgia Tech

Abstract

Modern datacenter applications are concurrent, so they require synchronization to control access to shared data. Requests can contend for different combinations of locks, depending on application and request state. In this paper, we show that locks, especially blocking synchronization, can squander throughput and harm tail latency, even when the CPU is underutilized. Moreover, the presence of a large number of contention points, and the unpredictability in knowing which locks a request will require, make it difficult to prevent contention through overload control using traditional signals such as queueing delay and CPU utilization.

We present Protego, a system that resolves these problems with two key ideas. First, it contributes a new admission control strategy that prevents compute congestion in the presence of lock contention. The key idea is to use marginal improvements in observed throughput, rather than CPU load or latency measurements, within a credit-based admission control algorithm that regulates the rate of incoming requests to a server. Second, it introduces a new latency-aware synchronization abstraction called Active Synchronization Queue Management (ASQM) that allows applications to abort requests if delays exceed latency objectives. We apply Protego to two real-world applications, Lucene and Memcached, and show that it achieves up to 3.3× more goodput and 12.2× lower 99th percentile latency than the state-of-the-art overload control systems while avoiding congestion collapse.

1 Introduction

One of the key objectives of datacenter operators is to maximize the utilization of limited resources. While operating a server close to its capacity maximizes its throughput, it also makes it susceptible to overload due to surges in demand. Such surges can occur due to variability in request arrival patterns and sizes, and service failures. The resulting server overload can cause *receive livelock*, where the server builds up a long queue of requests that get starved because the server is busy processing new packet arrivals instead of completing pending requests [22].

The conventional solution is to use *overload control* to regulate incoming requests and shed excess load, ensuring that the server can achieve both high utilization and low latency. Existing overload control schemes focus on CPU overload [9, 34] or end-to-end response time [32]. However, we found these approaches perform poorly under lock contention, especially with blocking synchronization (e.g., mutexes) that causes a thread to yield rather than spinning on the CPU (§2). For these cases, contention leads to long queues of requests waiting to acquire a critical section, increasing tail latency and wasting CPU resources.

To better understand the challenge of managing lock contention, consider a key-value store, where the key-value pairs are grouped together based on the hashes of their keys. Access to a bucket (i.e., a group of items with the same hash) is protected by an item lock. This means that in a key-value store, the number of locks corresponds to the number of buckets. However, a `GET` request acquires only a single lock which synchronizes access to the bucket holding the data it's accessing. As a specific piece of data becomes popular, the lock protecting its bucket becomes highly contended, negatively impacting the latency of all requests attempting to access that bucket. However, it is important to note that such contention and high delay impact *some but not all* of the requests the application handles. The remainder of the requests can be accessing different buckets incurring no contention, finishing with minimal latency.

To maintain good performance under lock contention, one must reduce the load on the contended lock, and thus the latency of requests attempting to acquire it. On the other hand, this should not be done in a way that affects the throughput of requests not facing contention. The classic tension between throughput and latency is exacerbated in this case due to the unpredictability of request behavior: the locks accessed by a request can only be known after the execution of the request starts. Thus, the delay faced by different requests, that look identical when admitted to the server, can be very different depending on whether they attempt to access a contended resource or not. This renders overload signals that consider the

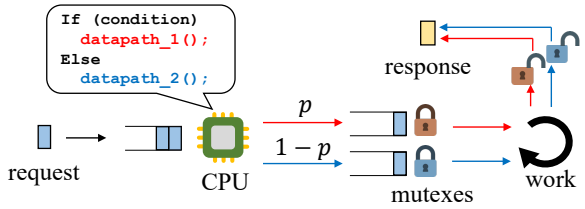


Figure 1: A simple example application with two global mutexes. With a probability p, the request takes the first data path (red arrow).

overall delay of requests ineffective. Furthermore, blocking locks can prevent the load from saturating the CPU, rendering CPU-based overload signals ineffective as well.

In this paper, we attempt to answer the following question: *how should an overload controller decide to admit a request when it can't estimate the delay the request will face?* Tackling this challenge is exacerbated by the fact that some applications have thousands of locks. Moreover, shedding load after processing a request requires cleaning up the state and resources touched by that request.

We present **Protego**, a system that provides overload control for applications that can experience lock contention (§3). Instead of using traditional overload control signals, it admits load as long as it observes throughput improvements. This approach ensures high throughput for requests not experiencing contention. However, it can exacerbate lock contention. Thus, Protego introduces new latency-aware synchronization primitives that allow applications to maintain low latency at contended critical sections, aborting requests when lock contention is too severe. As a result, Protego can offer the right load to maximize a server's throughput, even if some requests must be aborted during processing. We implemented Protego and compared it to SEDA and Breakwater, two state-of-the-art overload schemes, for three applications: Memcached, Lucene, and a synthetic application (§4). Our evaluation demonstrates that Protego outperforms SEDA and Breakwater for a wide range of workloads and applications (§5). For example, when Memcached is handling a `SET`-heavy workload, Protego achieves up to 1.6× more goodput with 5.7× lower 99th percentile latency compared to SEDA.

Protego has some limitations. It requires application-level code changes to adopt our synchronization API. Furthermore, existing overload control schemes can achieve slightly higher throughput than Protego when locks are not the bottleneck and requests are shorter than a microsecond.

Protego is an open-source software available at `https://inhocho89.github.io/protego/`.

2 Motivation

2.1 Locking Complicates Overload Control

In modern datacenter applications, RPC requests often require blocking synchronization (e.g., mutexes, semaphores, and conditional variables) to serialize access to shared data. However, blocking synchronization primitives can experience

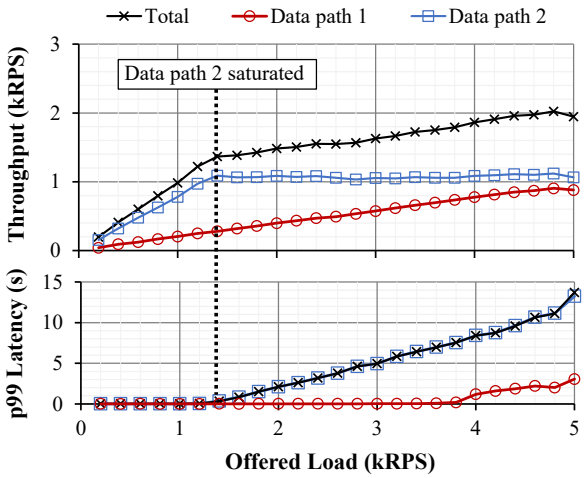
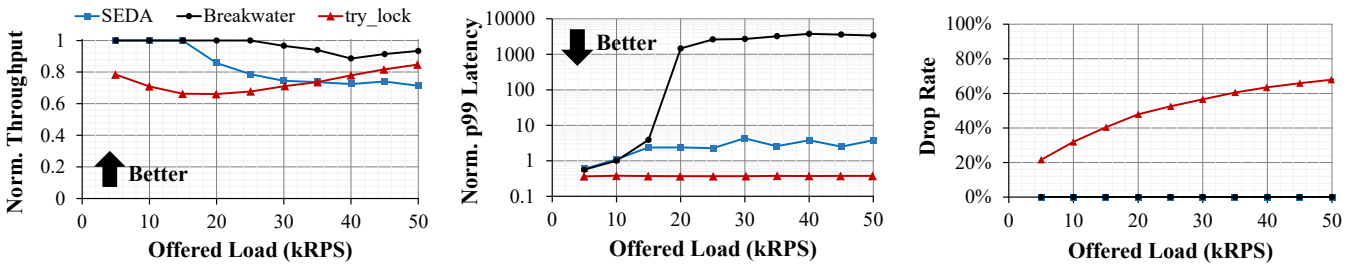


Figure 2: gRPC performance for the example application of Figure 1 ($p = 20\%$). After acquiring a mutex, requests busy-loop for a time sampled from an exponential distribution with 1 ms average. Four cores are allocated for this experiment, one for each data path and two to adsorb any system overhead, ensuring that the CPU is not bottlenecked.

contention when multiple requests attempt to access the same critical section, leading to a performance bottleneck. This is further complicated by the fact that the locks required by each request may be different depending on the request payload and the program's state. This makes it hard to know the data path a request will take before its actual execution.

The crux of this problem is that seemingly identical requests can have different execution paths at the server with different latency and throughput characteristics. This unpredictable behavior makes admission control hard, leading to the question: *which data path should admission control consider when admitting new requests?* To better understand this dilemma, consider the scenario in Figure 1. Incoming requests can take one of two paths, each protected with a different mutex. Requests can take the first data path with probability p, where $0 \leq p \leq 1$, and the second path with probability $1 - p$. We implemented this simple scenario in gRPC running on Linux. Figure 2 shows the performance of this scenario with $p = 20\%$ under various loads generated by client machines with an open-loop Poisson arrival process.

The existence of multiple data paths with different lock bottlenecks creates a dilemma. As shown in Figure 2, different datapaths are saturated at different offered load levels. Typically, clients and servers can't predict whether a request will take the datapath currently bottlenecked (data path 2 in the example). Here, the admission control dilemma emerges from the existence of multiple desirable operating points. If the operator desires low latency for all paths, then they have to sacrifice throughput, admitting only enough load to saturate the most congestion execution path (i.e., 1.2 kRPS in this example). On the other hand, if they desire high throughput, then they have to admit a high load and deal with the congested path through other means (e.g., dropping a request



(a) Normalized Throughput (b) Normalized 99%-ile latency (c) Drop Rate

Figure 3: Performance of Breakwater, SEDA, and trylock for the example application of Figure 1 ($p = 20\%$) with 100 µs average service time on Shenango. Throughput and 99th percentile latency are normalized by the performance of Protego.

after admitting it). Next, we show that no existing overload control scheme can navigate this dilemma and produce good results in such scenarios.

2.2 Problems with Existing Overload Control Schemes

Overload control attempts to operate a server near its capacity with minimal SLO violations and request drops. The basic idea behind overload control is to keep track of the load on the server using a signal, adjusting the admitted load based on that signal. Multiple signals have been proposed to improve the accuracy of admission control, including CPU utilization [30], end-to-end delay [32], and queuing delay [9, 19, 34]. However, none of these signals are useful in lock contention scenarios where the operator attempts to maximize throughput while maintaining low latency.

For example, Breakwater [9] and Swift [19] use past observations to predict the amount of queueing delay each request will face. However, in the presence of thousands of locks, it's unclear which queueing delay value (or statistic), if any, can be used to perform admission control. This is because admission control doesn't know in advance which locks requests will access, making it impossible to decide which value to react to without overestimating or underestimating overload. Note that any CPU-based metrics also fail as the CPU might not be the bottleneck in lock contention scenarios.

One possible approach to handle problematic or unpredictable lock behavior is to leverage existing primitives like `try_lock()` or `timed_mutex()`. Specifically, such primitives will allow requests to fail, avoiding latency, if the lock cannot be acquired due to congestion. However, overload control schemes that rely exclusively on request drops do not scale well due to the large overhead of packet drops. Furthermore, relying on existing primitives is not straightforward; `try_lock()` is a very aggressive overload control mechanism because it causes a request to fail on the first failed attempt to acquire a lock. On the other hand, `timed_mutex()` is too relaxed, forcing a request to wait for the full waiting time even under severe congestion conditions.

We demonstrate the limitation of existing overload control schemes, including the usage of `try_lock()`, by implementing those schemes for the scenario described in Figure 1, setting the average service time to 100 µs. However, rather than using gRPC, we leverage the existing implementation of SEDA and Breakwater [3]. Breakwater spawns a new thread per incoming request. We limit the number of spawned threads to bound the memory usage of the system. When a request is aborted, a failure message is reported to the client. The results are shown in Figure 3, comparing the throughput, tail latency, and drop rate of existing schemes, normalized by the performance of Protego.

SEDA successfully bounds the tail latency as it rate-limits clients based on the measured tail end-to-end latency. However, by considering only the tail latency, it reacts to the most congested path, leading to poor throughput as it underutilizes the uncongested path. Breakwater reacts only to queueing delay in the thread queue or the packet queue, reacting only to CPU and network overload. Thus, it doesn't perform any rate-limiting because neither the CPU nor the network is the bottleneck. Breakwater's behavior leads to high utilization and very high latency. Using `try_lock()` allows the system to achieve near-ideal latency while suffering from an extremely high drop rate and poor throughput. This is caused by `try_lock()`'s aggressiveness in dropping requests, wasting CPU and throughput even at low loads. Our proposal overcomes the shortcomings of existing systems, achieving the highest throughput while keeping the latency and drop rate low.

2.3 Challenges

Existing overload control schemes, developed for CPU overload scenarios, suffer significant performance degradation when handling lock contention. The key issue when dealing with lock contention is the unpredictability of the latency that a request will face. Particularly, the overload controller doesn't know which lock a request will require. This issue leads to the following challenges:

1. No existing overload control signal is viable. As discussed earlier, delay reflects the state of the most congested path. On the other hand, CPU utilization is not helpful when the bottleneck is not the CPU. Thus, we need a new approach to assessing the capacity of the server in order to make accurate admission control decisions.

2. Drops are inevitable to achieve high throughput. An overload controller that doesn't react to the most congested data

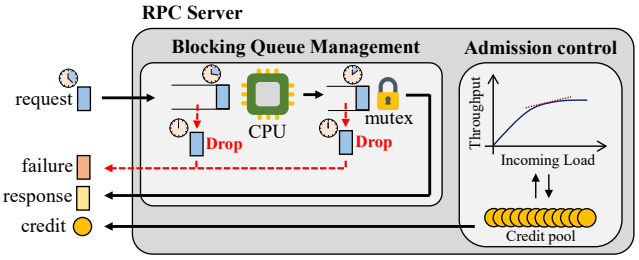


Figure 4: Protego Overview

path will incur a high delay for requests taking that path. However, it must offer enough load to keep other, less-congested paths busy. Therefore, maintaining both an acceptable SLO and high CPU utilization requires dropping requests on the most congested paths and reporting failures to the client. Early failure reporting allows the client to issue the requests to another replica while maintaining the SLO of the request.

3. Any viable solution must scale to a large number of locks. Modern programs can have thousands of data paths and synchronization primitives. An incoming request can take any of them depending on the data it carries. Thus, the admission control scheme needs to scale to a large number of locks with minimal per-lock overhead.

3 System Design

There is a fundamental tradeoff between throughput and drop rate in the presence of unpredictable synchronization. To achieve high throughput, clients should offer enough load for the server to fully utilize its uncontended data paths. Unfortunately, this permits some congestion to occur in its contended data paths. Thus, our high-level strategy is to use an admission control scheme that admits enough load to keep all data paths operating at full capacity, combined with an Active Queue Management (AQM) mechanism that drops excess load on the contended data paths. Our admission control scheme draws insight from network congestion control algorithms like PCC [12]. Specifically, Protego does not react to a specific overload signal. Rather, it observes the impact of its current admission rate on the behavior of the system, admitting more load only when it improves overall system performance.

Figure 4 illustrates an overview of Protego combined with a simple RPC server that uses a global mutex. Protego is composed of two main components: an admission controller and an AQM mechanism. The admission controller leverages a credit-based scheme, similar to the scheme used in Breakwater [9]. Protego only changes the way the number of available credits is decided, adjusting the number of credits by observing the impact of increasing the number of available credits on achieved throughput. The AQM mechanism uses *Active Synchronization Queue Management* (ASQM), a novel form of AQM that drops requests at lock acquisition time to prevent blocking on a critical section for an excessive amount of time. When a request is dropped, Protego reports this failure as quickly as possible to clients, allowing them to resend their requests to another replica.

3.1 Performance-driven Admission Control

Our goal is to develop an admission control algorithm that allows a server operator to navigate the tradeoff between throughput and drop rate. Note that the admission control algorithm should support scaling to a large number of data paths. Thus, we avoid developing an algorithm that has to take into account the state of every data path in the server.

Intuition. To better understand the intuition behind our algorithm, we go back to the setup in Figure 1. Specifically, we rerun the experiment discussed in Section 2.2. However, we use a smaller service time per request (10 µs rather than 100 µs) because these results help to make our point clearer. Moreover, we don't use any admission control scheme but rely on the AQM scheme, discussed in the next section, to keep latency bounded. The results are shown in Figure 5. The design of our admission control scheme stems from observing that as the load increases, the system operates in four different phases:

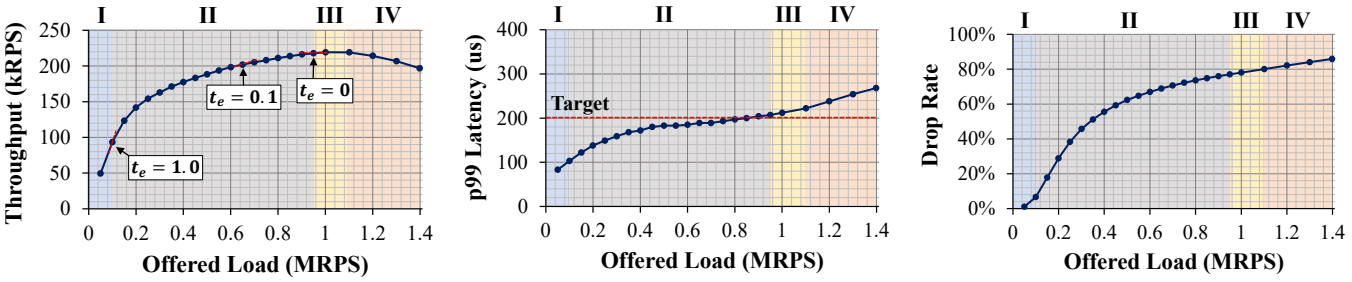
Phase I (uncongested) is the phase where none of the locks or CPUs is congested. Throughput grows linearly with load increases because the system has capacity to handle all incoming demand. Further, tail latency increases only marginally because of bursts in the queue caused by the variable request arrivals, modeled as a Poisson arrival process. With no congestion, AQM does not drop the requests.

Phase II (partially congested) is the phase where a subset of locks are contended. As load increases, throughput increases sub-linearly because the system has capacity to handle only a fraction of incoming demand (i.e., the uncongested path still has capacity). Incoming requests that take the congested path will face high queueing delay, leading AQM to start dropping requests while keeping the tail latency near the target value. To generalize, different applications will produce a different concave line like that shown in Figure 5(a), where the slope of the curve decreases as more paths become congested. The exact shape of the curve depends on the number of congested paths, and their capacities along with the load.

Phase III (congested) is the phase where all the data paths become congested. Thus, as the load increases, the throughput doesn't change. However, the increase in load increases CPU utilization because of the increase in network processing load and the increasing overhead of dropping requests. Eventually, the CPU also becomes congested, increasing tail latency.

Phase IV (congestion collapse) is the phase where the system enters a livelock state, spending more time dropping requests than processing them. During that phase, throughput degrades and latency keeps increasing.

Overview. Admission control should bound the incoming load to make the server operate in Phase II. Note that the values of latency, drop rate, and CPU utilization do not help



(a) Throughput (b) 99%-ile latency (c) Drop Rate

Figure 5: Performance of the application in Figure 1 ($p = 20\%$) with 10 µs average service with the latency bounded by ASQM

Algorithm 1 Performance-driven credit management

t_e: efficiency threshold
t_d: maximum drop threshold
C: the size of credit pool
$in_{\{last,cur\}}$: # of incoming requests in {last, current} iteration
$out_{\{last,cur\}}$: # of outgoing responses in {last, current} iteration
$drop_{cur}$: # of request drops in current iteration
a: increment step size
d: multiplicative decrement factor

repeat Every 4 * end-to-end RTT
  if $drop_{cur} > t_d \cdot in_{cur}$ **then**
    $C \leftarrow (1-d) \cdot C$
  else if $(in_{cur} - in_{last})(out_{cur} - out_{last}) > 0$ **then**
    if $|out_{cur} - out_{last}| > t_e \cdot |in_{cur} - in_{last}|$ **then**
      $C \leftarrow C + a$
    else
      $C \leftarrow (1-d) \cdot C$
    end if
  else
    $C \leftarrow (1-d) \cdot C$
  end if
  $C \leftarrow \max(C, C_{min})$
  $C \leftarrow \min(C, C_{max})$
  $in_{last} \leftarrow in_{cur}$
  $out_{last} \leftarrow out_{cur}$
until Application exits

identify the phase in which the server operates. However, by observing the slope of the throughput curve, one can identify the boundaries of Phase II. Specifically, Phase II starts when the slope of the throughput curve drops from 1 (i.e., the system can no longer handle *all* incoming requests) and ends when the slope reaches 0 (i.e., the system can no longer handle *any* additional incoming requests). A server operator that's interested in achieving a near-zero drop rate would operate the server at the leftmost edge of Phase II, where the slope of the throughput curve is slightly lower than one. On the other hand, a server operator that's interested in achieving the highest possible throughput would operate the server at the rightmost edge of Phase II, where the slope of the throughput curve is slightly higher than zero. The server operator can operate between those two points by choosing desired slope value. Additionally, the operator could specify the region of operation further by capping the maximum allowed drop rate.

We propose a performance-driven admission control algorithm with two parameters: efficiency threshold (t_e) and maximum drop rate (t_d). The efficiency threshold represents the target operating point on the throughput curve in terms of the slope of the curve at that point. Specifically, t_e takes values between zero and one, with zero representing the highest possible throughput, and one representing zero drop rate. The maximum drop rate, t_d, allows a service operator to cap the drop rate at the expense of throughput to reduce the expected number of request drops. Protego uses the maximum drop rate in addition to the efficiency threshold to determine whether to accept more incoming load. Protego judges an RPC server to be overloaded, accepting no further load, if throughput improvement with additional load is less than the efficiency threshold or if the drop rate exceeds the maximum drop rate.

Operation. A Protego server controls the number of incoming requests through the credit-based scheme we developed for Breakwater [9]. We chose a receiver-driven credit-based admission control scheme because it was shown to be robust to incast scenarios, efficiently scaling to a large number of clients while maintaining its performance [8, 9, 17, 23]. Like 1RMA [29] and Breakwater [9], Protego requires a new client to declare its intent to send requests to the server by sending an initial Request To Send (RTS) message. For Protego, this message is needed only when a new client connects to the server and is not needed for every request. The server issues credits to clients. A credit represents availability at the server to process a single request by the client that receives the credit. A client only sends a request after it receives a credit. A client disconnecting from the server has to send a Disconnect message to inform the server to stop allocating credits to it. Note that credits in Protego provide minimal commitment as the server cannot know in advance whether an incoming request will take a congested or an uncongested path. Protego determines the total number of available credits, C, before distributing them to individual clients.

The server measures its efficiency (the change in throughput divided by the change in admitted load). If measured

efficiency is less than the efficiency threshold (t_e), the server reduces the credit pool size, reducing the admitted load; otherwise, it increases the credit pool size. In particular, the server operates in iterations, each lasting a few end-to-end RTTs.[1] We measure the end-to-end RTT with the elapsed time between credit issue and the successful response return which is tracked with an 8B unique credit ID. The server keeps track of the number of admitted requests from the current iteration and the previous iteration, in_{cur} and in_{last}, respectively. It also keeps track of the current throughput and the throughput in the previous iteration, out_{cur} and out_{last}, respectively. The efficiency metric $e = (out_{cur} - out_{last})/(in_{cur} - in_{last})$ is compared to the efficiency threshold t_e. The server continuously monitors the drop count $drop_{cur}$ and decreases the admitted load if $drop_{cur}$ exceeds $t_d \cdot in_{cur}$. Protego uses additive increase / multiplicative decrease (AIMD) for credit management due to its simplicity. The details of the algorithm are shown in Algorithm 1.

3.2 Active Synchronization Queue Management (ASQM)

Protego assumes a standard queue abstraction per blocking synchronization object. However, to ensure scalability, Protego requires no coordination between queues, no per-queue parameter setting, and only minimal changes to the existing implementation of the synchronization API. Specifically, ASQM caps the total time a request is allowed to spend in a queue, assigning each request a queueuing delay budget. The value of the budget represents the maximum queueing delay a request can tolerate for the server to respond within a target latency. The queueing delay budget is computed by subtracting the 99th percentile network latency and 99th percentile service time from the *target delay* of the request, leaving the slack time that the request can afford to spend in the server.

When a request arrives at the server, Protego assigns it a queueing delay budget. Before placing the request in each queue for a contended resource, it first checks the instantaneous queueing delay of the queue and drops the request if the queueing delay is larger than the request's remaining queueing delay budget. After the request is dequeued, it deducts the queueing delay it incurred from its budget. The queueing delay is measured by computing the difference between the current timestamp and the enqueue timestamp of the oldest item in the queue. In this paper, we only consider the runnable thread queue in the CPU scheduler and the wait queues for blocking synchronization primitives. However, we believe the same idea can be applied to other queues for contended blocking interfaces such as blocking I/O.

Target delay vs. SLO. It's critical to note that the target delay used to compute the queueing delay budget is different from the RPC's Service Level Objective (SLO). The target delay is a per-server metric: a single server should finish a request or report failure within the target delay. On the other hand, an SLO is a per-request metric: a request of a specific type should finish within its SLO, taking into account that multiple attempts at multiple servers might be needed for the request to succeed. In Protego, the target delay is set by default to SLO divided by the maximum number of retries.

Handling dropped requests. Upon a request drop, the server returns a failure message immediately to the client. At the server, a request drop incurs some CPU overhead to partially process the request and generate the failure message. Further, the failure message and retransmission of the request can incur networking overhead. If the overhead of dropping requests is large, a service operator can reduce the drop rate by choosing a higher value for the efficiency threshold (t_e), sacrificing throughput. At the clients, the dropped request may be handled in various ways: retransmission to another replica, triggering failure handling operations (e.g., online banking transaction), or degrading the quality of the response (e.g., search). For systems with replication and auto-scaling, retransmission is the most common failover mechanism. For the rest of the paper, we focus on scenarios where an overloaded server has a non-overloaded replica which can serve dropped requests.

Retransmission of dropped requests introduces additional latency, inflating the overall delay faced by such requests, potentially harming their SLOs. Protego drops requests before they consume their delay budget. Thus, clients receive failure messages within the target delay. In the worst case, for each retransmission, a request will be delayed by at most the target delay (§5.3). Alternatively, if the SLO is tight, the client can send tied or hedged requests to multiple replicas to avoid the retransmission delay but incur the cost of coordination overhead and/or CPU wasted by duplicate executions [11].

3.3 System Parameters

In total, Protego has five parameters: four universal parameters whose value can be fixed across workloads, and one workload-specific parameter.

Universal parameters. The efficiency threshold and maximum drop rate parameters, t_e and t_d, do not need to change per workload. We show that the performance of Protego is not very sensitive to the choice of t_e (§5.4). We use an efficiency threshold of 10% by default. The maximum drop rate puts a cap on the allowed drop rate. Operators that want to maximize throughput should set it to 100%, which is the default value we choose in the paper.

AIMD algorithms have two parameters: an increment step size (a) and a decrement factor (d). Large values of a and d make the algorithm more aggressive in reaching the desired operating point but less stable with large fluctuations. We choose small values for a and d, preferring stability. We set a as 0.1% of the number of the client sessions and d as 2%, which leads to good performance in incast scenarios [9].

Workload-specific parameters. The target delay specifies

[1] We found that four RTTs allows for accurate measurement of all parameters while allowing for fast reaction to changes in the workload.

the maximum delay allowed in a single server. Its value is calculated as the SLO divided by the expected number of attempts that a request can make before it succeeds.

4 Implementation

We implemented Protego as a library that uses Shenango [25] and builds upon the RPC-layer implementation of Breakwater [9]. Furthermore, Protego extends Shenango's synchronization library to implement ASQM, facilitating the adoption of Protego to Shenango applications.

Performance measurement. Protego adjusts the credit pool size, once every iteration, based on five measures of efficiency and drop rate: in_{cur}, out_{cur}, $drop_{cur}$, in_{last} and out_{last}. The measures are updated (i.e., current measures are reset after their values are assigned to the last measures) after one end-to-end RTT from the time the credit pool size is updated to accurately reflect performance during an iteration. This period is selected because the incoming load changes in correspondence to the new pool size after at least one end-to-end RTT.

Dispatcher threading model. Protego assigns a queueing delay budget per request, deducting from it after a request is serviced from a queue. This operation requires accurately tracking the time a request spends in various queues, avoiding any variability that might be introduced due to the operating system or the network stack. Thus, we implement Protego with a dispatcher threading model where a dispatcher thread parses the network payloads into requests, spawning a new thread for each incoming request. This approach minimizes the delay requests face in the network stack because packets are parsed quickly by the dispatcher thread, out of the critical path of request processing. When a new thread is created by a dispatcher, it's assigned a queueing delay budget by subtracting the 99th percentile network latency and the 99th percentile service time from the target delay.

Latency-aware Active Synchronization Queue Management (ASQM) API. Protego provides the following latency-aware APIs to enable ASQM:

```
bool mutex_lock_if_uncongested(mutex_t *);
bool condvar_wait_if_uncongested(condvar_t *,
                                 mutex_t *);
```

These interfaces are similar to those of a `try_lock()`, but their behavior is different. If the queueing delay of a blocking critical section exceeds a request's queueing delay budget, it returns false without waiting. Otherwise, it returns true after successfully acquiring the lock. An application developer can leverage the existing synchronization API provided by Shenango, including `mutex_lock()` and `condvar_wait()` for parts of the program that cannot handle dropping. For example, a maintenance thread running in the background may need to acquire a lock no matter how long it has to wait.

Queueing delay measurement. Protego needs to measure instantaneous queueing delay to compare it against a request's remaining budget. We instrument the waiter queue for mutexes and conditional variables to measure the queueing delay.When a thread is enqueued to the waiter queue, Protego timestamps the request. When the blocking synchronization is queried for the queueing delay, it returns the difference between the current timestamp and the enqueue timestamp of the oldest thread in the waiter queue. Using an efficient hardware timestamp read function, Protego can measure the queueing delay of blocking synchronization with little overhead.

Identifying contended locks. In order to get the full performance benefits of Protego, developers must identify all the contended locks to replace with Protego's ASQM APIs. A developer needs to hypothesize which locks are likely to be contended based on the application-specific knowledge and run experiments to verify which locks introduce a large queueing delay with per-lock queueing delay measurements. This process requires iterating multiple times until all the contended locks are identified and their code is modified to use the Protego API. Alternatively, a developer can use high-resolution latency profilers [16] to identify contended locks.

Application modification. Enabling Protego requires replacing blocking synchronization primitives with the ones provided in the Protego API. Further, Protego allows requests to be dropped after they have been partially processed by the server, potentially modifying some states or reserving some resources. Thus, enabling Protego requires the application to perform all necessary clean-up after a request is dropped (e.g., freeing memory it allocated to the request and releasing other locks the request currently holds). However, the complexity of handling request drops can be significantly reduced by utilizing features of modern programming languages, such as RAII in C++ with smart pointers and scoped locks.

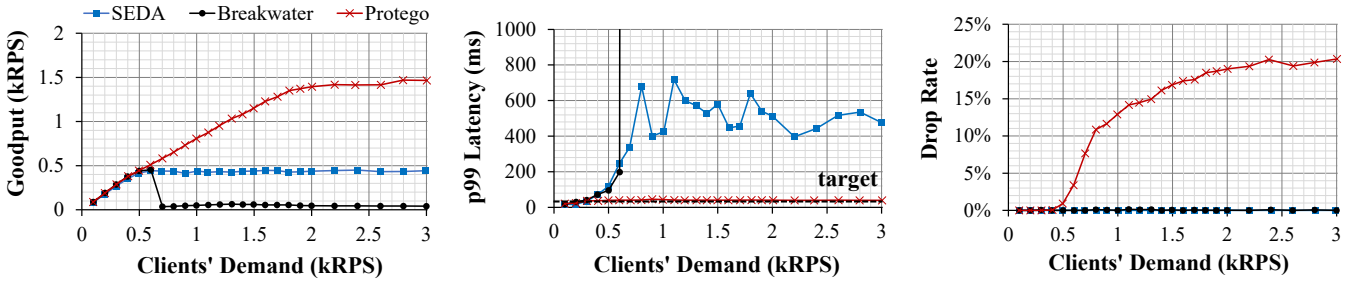
5 Evaluation

Our evaluation answers the following key questions:

1. Can Protego balance high throughput and low latency for real-world applications?
2. How much code change is required to enable Protego?
3. Does Protego maintain its benefits for different workloads?
4. Can requests maintain their SLO in the presence of drops?
5. How much does each component of Protego contribute to its overall performance?
6. How sensitive is the performance of Protego to its parameter values?
7. What are the limitations of Protego?

5.1 Evaluation Setup

Testbed: We use eleven xl170 nodes in Cloudlab [13]. Each node has a ten-core (20 hyper-threads) Intel E5-2640v4 2.4GHz CPU, 64GB ECC RAM, and a Mellanox ConnectX-4 25GbE NIC. Nodes are connected through a single Mellanox 2410 switch. The average and 99th percentile network RTT between any pair of two nodes are 10 µs and 20 µs, respectively. We use one node as an RPC server and the other ten nodes



(a) Goodput (b) 99%-ile latency (c) Drop rate

Figure 6: Performance of SEDA, Breakwater, and Protego for Lucene

as RPC clients. The server application uses up to ten hyper-threads for real-world applications and four hyper-threads for synthetic application. Each client machine simulates one hundred RPC client connections with sixteen dedicated, spinning hyper-threads. Requests are generated following an open-loop Poisson arrival process.

Workloads: We evaluate Protego using three workloads: 1) Lucene, a search application with significant lock contention overhead, 2) Memcached, a latency-sensitive in-memory key-value store that exhibits both locking bottlenecks and CPU bottlenecks, and 3) a synthetic workload with its execution time drawn from an exponential distribution.

Baseline: We compare Protego to SEDA, a latency-based overload control system, and Breakwater, a queueing delay-based one. SEDA controls the load at the server by rate-limiting the clients. Each SEDA client adjusts its request sending rate based on the 99th percentile end-to-end latency faced by requests. Breakwater controls the load at the server through a credit-based mechanism, adjusting the credit pool size based on the sum of packet queueing delay and CPU thread queueing delay. To ensure low latency, Breakwater drops a request if the queueing delay exceeds a workload-based threshold.

Evaluation metrics: To incorporate throughput, latency, and the target latency into one single metric, we compute goodput as the throughput of the requests whose latency is below the target delay. For Breakwater and Protego, we report the drop rate as the ratio of the number of dropped requests to the number of requests received by the server during an experiment. SEDA does not drop the request at the server. We run the experiments for 8 seconds and collect the data for the last 4 seconds to capture the steady-state behavior.

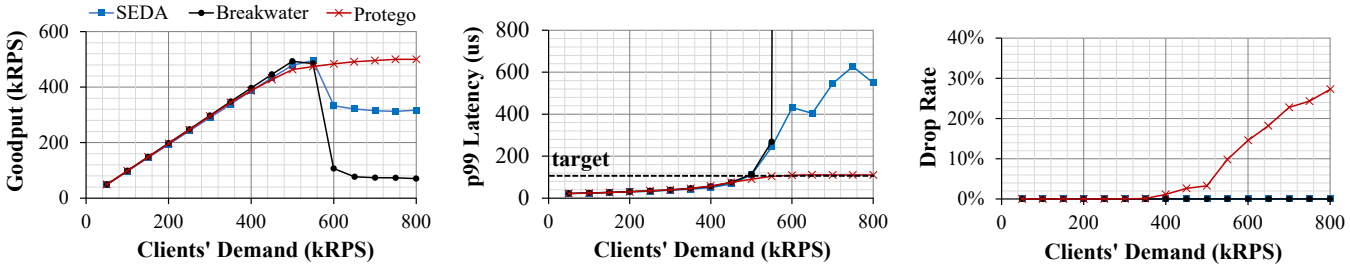
Parameter settings: We tune the parameters of all systems to allow each system to achieve its best goodput for each workload. For SEDA, we adjust *timeout* (request sending rate update interval), adj_i (rate increase factor), and adj_d (rate decrease factor). We use the default configuration from [32] for all other parameters. For Breakwater, we tune the target queueing delay and the drop threshold which we set to 40% and 80% of Protego's target delay, respectively, for all workloads. We use the default configuration from [9] for all other parameters. For Protego, we use an efficiency threshold (t_e) of 10%, a maximum drop rate (t_d) of 100%, an increment step size (a) of 1, and a decrement factor (d) of 2% for all workloads. We determine the queueing delay budget for ASQM by deducting 99th percentile service time and 99th percentile network delay (20 µs) from the target delay for each workload. We determine the target delay as the maximum value between $10\times$ the sum of average network RTT (10 µs) plus the average service time, and $2\times$ the sum of 99th percentile network RTT (20 µs) plus the 99th percentile service time. For example, for the exponential service time distribution with 10 µs average whose 99th percentile is 46 µs, we set the target delay to 200 µs because $10\cdot(10+10)=200$ µs is higher than $2\cdot(20+46)=132$ µs. The way we set the target delay is comparable to how the SLO is calculated in recent proposals [9, 10, 26]. We set the SLO as twice the target delay, assuming that a request fails at most once.

5.2 Mutex-intensive Application: Lucene

Lock contention inside Lucene: Lucene is a search engine library that maintains two main types of structures: 1) inverted indices, called `Segments`, and 2) per-term scores of all indexed documents, called `TermDocs`. Every `Segment` and `TermDocs` is protected by its own mutex. Every request performs a binary search over all `Segments` to find the documents corresponding to its search query. Then, documents are ranked based on the information found in the `TermDocs` corresponding to the identified documents.

As load increases on the server, the per-`Segment` lock becomes contended because every request needs to search over all the `Segments`. `Segments` containing more entries are more likely to be contended because it takes more time to perform a binary search over their entries. Further, if a specific document becomes popular, the per-`TermDocs` lock protecting its data becomes contended.

Application modification: We ported the C++ version of Lucene, Lucene++ [31], to Shenango and built a simple in-memory search application, where all the data is stored in memory with `RAMDirectory`. We replaced the per-`Segment` lock and per-`TermDocs` lock with Protego's latency-aware synchronization API to allow request drops. In total, we modified 40 LOC of Lucene++ after porting it to Shenango. Note that, while Lucene allows for reporting partial search results,



(a) Goodput (b) 99%-ile latency (c) Drop rate

Figure 7: Performance of SEDA, Breakwater, and Protego for Memcached with VAR workload

we don't allow that to provide a fair comparison between overload control schemes that don't drop requests. The response contains either the complete search result or a failure notification.

Workload and configuration: We populate the server with a dataset of 403,619 COVID-19-related tweets [6] in English posted between 27th and 29th November 2021. The clients generate single-term search queries. The search term (or word) is sampled from the word distribution in the data set excluding stop words like "a", "the", "and", etc. All the tweets are loaded to the server before serving clients, and tweets are not modified or deleted during an experiment. This workload yields an average processing time of 1.7 ms and a 99th percentile latency of 20 ms on a lightly-loaded server. Thus, we set the target delay to 40 ms. For SEDA, we set $timeout = 1$ s, $adj_i = 0.1$, and $adj_d = 1.3$. For Protego, we use an initial queueing delay budget of 20 ms.

Overall performance: Figure 6 shows the goodput, 99th percentile latency, and drop rate for all three overload control schemes. Note that Lucene does not suffer from any CPU congestion. Thus, Breakwater's admission control and AQM are never triggered, leading to congestion collapse as mutexes become congested with demand exceeding 600 RPS. SEDA reduces clients' request sending rate as soon as it measures high latency due to a mutex congestion, reacting to the most congested data path, which limits the system's goodput to 500 RPS. SEDA's tail latency is bounded but more than 10 times higher than the target latency because of incast. By better utilizing uncongested data paths and dropping the excess load, Protego achieves up to 3.3 times higher goodput and 17 times lower 99th percentile latency than SEDA.

5.3 Latency-critical Application: Memcached

Lock contention inside Memcached: The key-value pairs are stored in a giant hash table, composed of multiple hash buckets. Memcached has two main types of locks that may be contended. First, each hash bucket is protected by a mutex called `item_lock`, and this mutex may get contended not only by concurrent accesses (i.e., reads or rights) to the same key but also by accesses on different keys sharing the same key hash. Thus, it's difficult to predict which `item_lock` a request will need before executing it. Second, Memcached manages its memory by assigning items memory from a global pool,

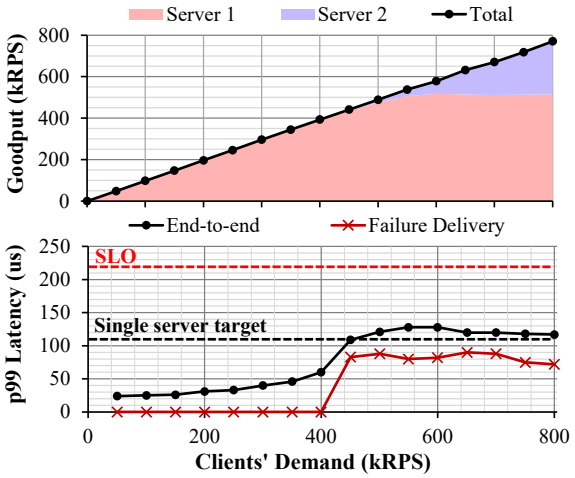
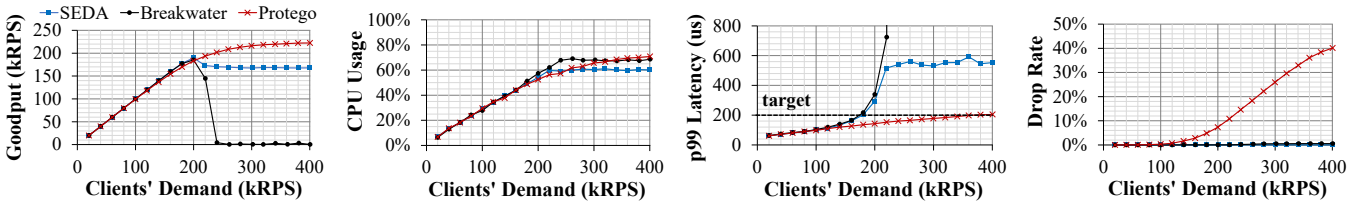


Figure 8: Service-level performance of Protego for the Memcached VAR workload with retransmission

which is protected by a global lock called `slabs_lock`. Every `SET` and `UPDATE` request must grab the `slabs_lock` to allocate memory for the new value.

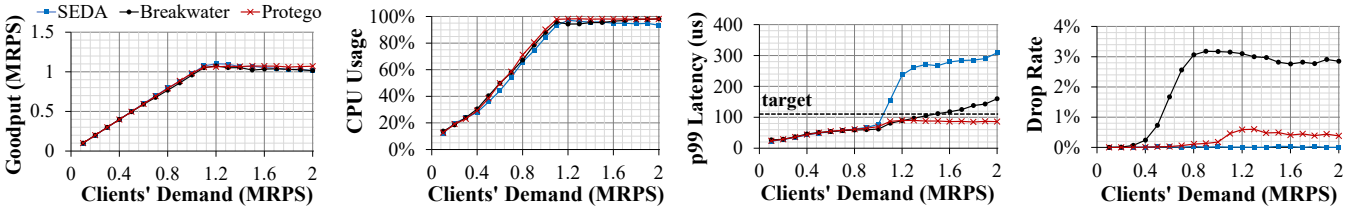
Application modification: We replaced the `item_locks` and `slabs_lock` with Protego's latency-aware mutexes. When a request is dropped, Protego delivers a failure message to the client immediately. Furthermore, it cleans up the intermediate state processed by the request, freeing up the chunk allocated to the request before the thread handling that request exits. We don't allow drop when a request tries to reacquire `slabs_lock` to free up the memory to avoid memory leaks. In total, we modified 50 LOC in Memcached [4], excluding the modifications to port it to Shenango.

Workload and configuration: For Memcached experiments, we use the VAR workload from Facebook Memcached cluster [33]. VAR is a `SET`-heavy workload for server-side browser information where 82% of the requests are `SET` requests. The key distribution of the workload is skewed with 10% of the keys used by 90% of the requests. With a `SET`-heavy workload, `slabs_lock` becomes the bottleneck as all SET requests require `slabs_lock` to allocate memory region. We approximately follow the key and value size distribution for each workload as described in [33]. We generate 100,000 key-value pairs and use the hash power of 17, providing 131,072 buckets in the hash table, which is sufficient to avoid severe hash col-



(a) Goodput (b) CPU usage (c) 99%-ile latency (d) Drop rate

Figure 9: Performance of SEDA, Breakwater, and Protego for synthetic workload with $p = 50\%$ and 10 µs average service time



(a) Goodput (b) CPU usage (c) 99%-ile latency (d) Drop rate

Figure 10: Performance of SEDA, Breakwater, and Protego for synthetic workload with $p = 50\%$ and 1 µs average service time

lisions. Since `SET` requests complete within less than 1 µs on average, we set the target delay to 110 µs. For SEDA, we set *timeout* to 1 ms, adj_i to 100, and adj_d to 1.02. For Protego, we set the initial queueing delay budget to 70 µs.

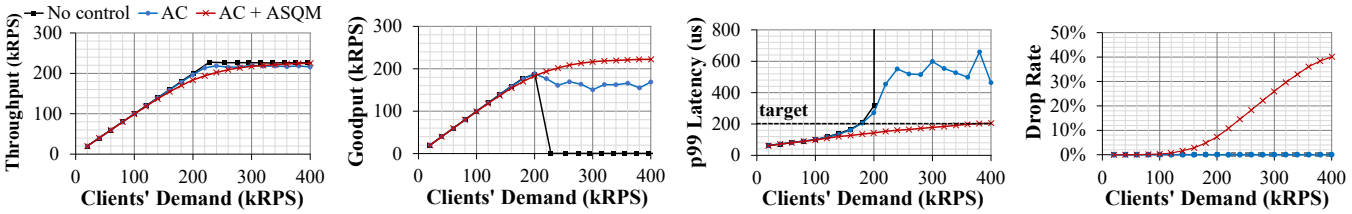
Performance with a global mutex bottleneck: Figure 7 demonstrates the performance of the three overload control schemes. When the `slabs_lock` becomes contended with clients' demand of more than 550 kRPS, both Breakwater and SEDA experience a goodput drop because of the increase in latency. As with Lucene, the admission control and AQM of Breakwater are not triggered because the CPU is not congested. On the other hand, SEDA suffers from incast. The goodput of Protego increases further by utilizing uncongested data paths with `GET` requests achieving 1.6 times higher goodput than SEDA and 7 times higher goodput than Breakwater. The increment in Protego's goodput is limited by the overhead of request drops. Most of the dropped requests are `SET` requests, and some of them require the `slabs_lock` to free the allocated memory. As more requests are dropped, the `slabs_lock` becomes more contended by new `SET` requests that need to allocate the memory as well as old and dropped requests that need to release their memory, resulting in lower throughput of `SET` requests at very high loads.

Maintaining the SLO under retransmissions: To better understand the impact of request drops on the overall SLO, we construct a simple scenario where Memcached has two replicas, but we otherwise use the same configuration as before. When a client makes a request, it sends the request to Server 1. If it is dropped, the client then retransmits it to Server 2 (after receiving a failure message from Server 1). This structure is similar to how Memcached is operated at Facebook [24] where they don't provide a strong consistency guarantee. Note that if both servers are overloaded, the problem ceases to be an overload control problem as the service operator needs to allocate more servers. Thus, our experiment captures the case where there is sufficient capacity to handle all requests, but retransmission may still be necessary. We anticipate up to one retransmission could happen, considering the capacity of the two servers and the demand the clients generate during the experiment, so we set the service-level objective (SLO) to two times the single server target delay, or 220 µs.

Figure 8 demonstrates the total goodput of both servers, the 99th percentile end-to-end latency, and failure message delay for the VAR workload. When the clients' demand exceeds 400 kRPS, Server 1 starts to drop requests. Protego drops the requests before they wait for the contended mutex if the delay at the mutex exceeds a request's budget. Thus, most of the failure messages are delivered within the target delay. Note that if a client doesn't receive a credit for a request within 10 µs from Server 1, it sends the request to Server 2 with the locally generated failure message. As clients' demand increases, the 99th percentile delay of failure messages decreases because more requests are retransmitted to Server 2 with local failure message. The overall 99th percentile end-to-end latency achieved by Protego is higher than the per-server target delay because some requests need to be retransmitted. However, it is still 1.7 × lower than the SLO.

5.4 Microbenchmark

Workload and configuration: To further analyze Protego's performance, we run the synthetic application depicted in Figure 1. We choose the configuration $p = 50\%$, making both data paths equally likely to be congested, to provide a best-case scenario for SEDA. We use a workload with exponential service time distribution of 10 µs and 1 µs average. The target delay values are 200 µs and 110 µs, respectively for the two settings. For SEDA, we set *timeout* = 1 ms, $adj_i = 10$, and $adj_d = 1.04$ for the first setting, and *timeout* = 1 ms, $adj_i = 40$, and $adj_d = 1.04$ for the second setting. For Protego, we set the initial queueing delay budget to 134 µs and



(a) Throughput (b) Goodput (c) 99%-ile latency (d) Drop rate

Figure 11: Performance of Protego by incrementally applying performance-driven admission control (AC) and ASQM with the synthetic application with 10 µs average service time

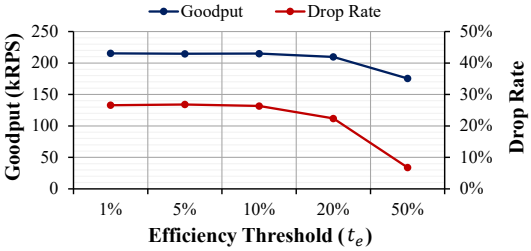
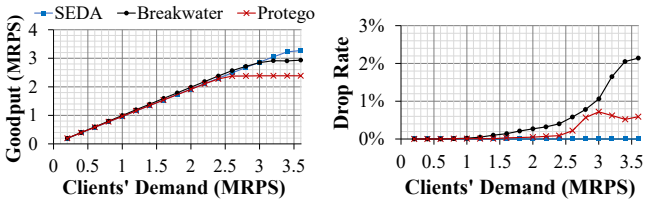


Figure 12: Protego parameter sensitivity (efficiency threshold, t_e)



(a) Goodput (b) Drop rate

Figure 13: Performance of SEDA, Breakwater, and Protego for Memcached with USR workload

85 µs, respectively, for the two settings.

Overall performance: Figure 9 shows the goodput, CPU usage, 99th percentile latency, and drop rate for a workload with 10 µs average service time. The performance is bottlenecked by the mutexes, leaving the CPU underutilized even with a high clients' demand. Thus, at high load, the admission control or AQM logic of Breakwater is not triggered, leading to congestion collapse. SEDA limits the sending rates of clients as soon as it measures high tail latency with a single temporarily congested data path. Thus, SEDA's goodput is limited to 168 kRPS leaving the other data path uncongested. With a larger clients' demand, SEDA suffers from incast because 1,000 clients are each running a control loop separately. As a result, it shows up to three times higher tail latency than the target delay. Protego improves goodput by up to 32% compared to SEDA, maintaining latency within the target delay by dropping up to 40% of incoming requests. Note that the performance benefits of Protego compared to SEDA increase as p deviates from 50%, making SEDA more conservative as it reacts to the most congested path.

Impact of average service time: We reduce the average service time to 1 µs, reducing the time requests can spend with the lock, allowing the CPU to become the bottleneck. The results are shown in Figure 10. As demand exceeds 1.1 million RPS, the CPU is saturated, triggering Breakwater mechanisms. However, it still suffers at high loads when the mutexes become contended. SEDA still suffers from high tail latency up to three times of the target delay because of the incast, but its impact on goodput is limited. Protego maintains the tail latency lower than the target delay while dropping less than 1% of the requests in a CPU-bounded scenario.

Performance breakdown: We measure the performance of Protego after incrementally activating its two components: the performance-driven admission control scheme (AC) and Active Synchronization Queue Management (ASQM). We run the experiments with the synthetic application with $p = 50\%$ and an average service time of 10 µs. Figure 11 shows the throughput, the goodput, the 99th percentile latency, and drop rate. With no overload control, goodput collapses as soon as one of the data paths becomes congested. Enabling admission control bounds the tail latency by limiting incoming load if there is no throughput improvement. However, when mutexes start to be congested, its goodput degrades with up to three times higher tail latency than the target because one of the mutexes can have a high queueing delay with the requests' probabilistic data path selection. By employing ASQM, Protego ensures the tail latency does not miss the target delay by dropping requests.

Parameter sensitivity: Protego balances goodput and drop rate using the efficiency threshold (t_e). To quantify the tradeoff between them, we repeat the experiment with the synthetic application with $p = 50\%$ and the average service time of 10 µs varying the t_e from 1% to 50%. Figure 12 shows the goodput and drop rate of Protego with different t_e values when the clients' demand is 300 kRPS, around 1.4 × of the capacity (consider Figure 11 as a reference). For all values of t_e smaller than 10%, the goodput and drop rate don't change because throughput improvements with a small t_e are always marginal. With larger t_e values, both the goodput and drop rate decrease as admission control targets to operate the server on the left side of the Phase II region in Figure 5. With $t_e = 50\%$, it achieves 23% less goodput and 4 × lower drop rate than $t_e = 1\%$, allowing server operators to navigate the tradeoff between goodput and drop rate.

5.5 Limitations of Protego

To demonstrate the limitations of Protego, we repeat the Memcached experiment in §5.3 with the USR workload, a `GET`-dominated workload for user account status information where 99.8% of the requests are `GET` requests and about 20% of the keys are used by 80% of the requests. With the USR workload, Memcached saturates the CPU when it's configured with a high enough hash power (i.e., a large number of buckets compared to the number of key-value pairs). However, some `item_locks` can still become congested intermittently because of the skewed key distribution. Figure 13 shows the goodput and drop rate, comparing Protego to Breakwater and SEDA. With clients' demand of 3.6 million RPS, Protego achieves 37% less goodput than SEDA and 23% less goodput than Breakwater.

The USR workload is CPU bottlenecked, allowing Breakwater mechanisms to be triggered. Protego achieves lower goodput than Breakwater due to the slow reaction of Protego's admission control. In particular, Protego changes its credit pool size every four end-to-end RTTs. On the other hand, Breakwater adjusts its credit pool size every network RTT. As a result, Protego reacts to both congestion and added capacity slowly, leading to a lower goodput. Breakwater and Protego achieve lower goodput than SEDA because of the overhead of credit management at the server. Specifically, SEDA doesn't add any extra logic at the server while Breakwater and Protego perform all their admission control and AQM calculations at the server. This overhead is significant when the request execution time is very small. Note that increasing the number of clients from 1,000 to 10k can lead to performance degradation in SEDA with a larger size of incast [9]. This experiment shows that Protego can lead to goodput degradation in some scenarios where the CPU is bottlenecked. However, if the setting has any significant likelihood of mutex congestion, Protego can introduce significant benefits even when the CPU is bottlenecked (Figure 10).

6 Discussion

Fairness. Protego does not provide any mechanisms to ensure fairness between clients. For example, a client issuing more requests that require contended locks will get more failure messages because it faces a higher drop rate. However, it does provide flexibility for clients in their selection of replicas. A client can choose to send requests to a replica with a lower drop rate or distribute requests to multiple replicas to lower its drop rate. In this paper, we assume that the system as a whole has enough capacity to handle requests, relying on elastic resource allocation schemes like auto-scaling.

Generalizing Protego for other in-application congestion. An evaluation of DeathStarBench [14] revealed a challenging overload scenario where the tail latency of an upstream service (NGINX) spiked more than 10× while its CPU usage remains low due to the blocking network socket call used in HTTP. The delay introduced by such calls cannot be detected with the overload signal used in DeathStarBench (i.e., CPU Usage). Thus, the auto-scaler is never triggered to launch a new instance, causing high tail latency. Protego can be used to handle such overload scenarios where blocking calls (e.g., network or storage system calls) are the bottleneck. More specifically, the performance-driven admission control can back-pressure upstream services when it observes that there is no throughput improvement as load increases due to blocking calls. If the invocation of blocking calls by requests is unpredictable, it would require editing those calls to support ASQM. Furthermore, in a multi-tier microservice architecture, upstream microservices might be able to abstract calls made to downstream microservices as blocking calls, allowing Protego to be used to perform overload control over the entire microservice chain.

7 Related Work

Overload control. To avoid congestion collapse with receive live lock, an overload control system tries to bound the incoming requests or drop the request to prevent overload. Overload can be detected using several metrics. Breakwater [9] and DAGOR [34] use thread and network queueing delay. SEDA [32] and ORCA [20] use response time as a congestion signal. The way a system controls the overload also differs across these systems. Breakwater utilizes credit-based admission control with AQM. DAGOR utilizes priority-based admission control with AQM. SEDA adjusts the request sending rate at the client side. ORCA uses TCP-like window-based approach at the client side.

Flow Control. In TCP and eRPC [18], flow control advertises the size of the available receive buffer to clients to prevent receiving more packets than the network stack can accommodate. Akka [1] Stream has a similar but more flexible flow control mechanism where a server signals the maximum number of requests it can handle to the clients based on the remaining buffer size, the amount of idle resources, etc. The clients do not send more requests than the demand signaled by the server. Flow control is useful to avoid high latency when the CPU is the bottleneck. However, when a blocking synchronization becomes the bottleneck, it achieves either low throughput by underutilizing uncongested data paths or high latency with long queueing delay.

Measurement-based network congestion control. BBR [7] and PCC [12] employ mechanisms similar to Protego's performance-driven admission control. BBR explores the maximum network bandwidth by measuring the throughput with increasing window size. It concludes that the network bandwidth has reached its maximum value if it observes less than 25% of bandwidth increase with doubled window size. Unlike Protego, BBR does not utilize a performance-based approach to detect network congestion but to determine a parameter used for congestion control. In PCC, the system operator defines a utility function (e.g., TCP friendliness, latency, or throughput). PCC conducts multiple micro-experiments

with a randomized set of parameters to find the configuration that achieves the highest utility. PCC-like algorithms require multiple rounds to find the best configuration, which slows down the reaction of the algorithm to the congestion. Unlike PCC, Protego deterministically modifies the credit pool size based on the measurement, which makes its reaction to congestion faster.

Auto-scaling. Auto-scaling [2, 5, 15, 21, 27, 28] dynamically changes the amount of resources allocated to a service based on various signals including CPU usage, estimated demand, or custom-defined signals. It ensures that a service has enough resources to serve requests by allocating more resources when the chosen signal indicates that a load has exceeded a configurable threshold. Some auto-scalers [15, 21] let service operators specify the signal (e.g., response time, SLO violation, cost, etc.). More recently, machine learning models are used for auto-scaling. Facebook [5] and Google Autopilot [28] auto-scale resources based on the estimated demand learned from historical data. FIRM [27] uses system-wide performance metrics (CPU, Memory, Disk I/O, Network usage, or arrival rate) to train and predict which microservices require how much additional resources not to violate SLO. Auto-scaling mechanisms are useful with consistent overload over a long time scale, but it does not handle transient bursts in a load that happen over small timescales. Such bursts can be handled by Protego. In addition, auto-scaling alone is not enough to achieve both high throughput and low latency in the presence of lock contention as it does not provide any way to drop requests in a congested data path.

8 Conclusion

In this paper, we presented Protego, an overload control system that handles overloaded blocking synchronizations with performance-driven admission control and Active Synchronization Queue Management (ASQM). Protego's admission control decisions are based on measured throughput, admitting more load only if it improves throughput, admitting less load otherwise. To ensure low latency even for congested data paths, Protego sheds load by dropping requests at contended blocking synchronization points using ASQM. Our extensive evaluation of Protego demonstrates that it can effectively handle overload when combined with lock contention, achieving high goodput and low latency for a wide range of conditions. In particular, Protego achieves up to 3.3× higher goodput with 12.2× lower 99th percentile latency than state-of-the-art overload control schemes when applied to Lucene, a realistic search workload.

Acknowledgments

We thank our shepherd Marios Kogias and the anonymous reviewers for their valuable feedback, and Cloudlab [13] for providing us with infrastructure for development and evaluation. This work was funded in part by NSF grants CNS-2104398, CNS-2212098, CNS-2104398, and CNS-2212099; DARPA FastNICs (HR0011-20-C-0089); VMware and Google.

References

[1] Akka. https://akka.io/.

[2] AWS Auto Scaling. https://aws.amazon.com/autoscaling/.

[3] Breakwater implementation on shenango. https://inhocho89.github.io/breakwater.

[4] Memcached. http://memcached.org/.

[5] Throughput autoscaling: Dynamic sizing for Facebook.com. https://engineering.fb.com/2020/09/14/networking-traffic/throughput-autoscaling/.

[6] J. M. Banda, R. Tekumalla, G. Wang, J. Yu, T. Liu, Y. Ding, K. Artemova, E. Tutubalina, and G. Chowell. A large-scale COVID-19 Twitter chatter dataset for open scientific research - an international collaboration, May 2020. https://doi.org/10.5281/zenodo.3723939.

[7] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 2016.

[8] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM*, 2017.

[9] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay. Overload control for μs-scale rpcs with breakwater. In *OSDI*, 2020.

[10] A. Daglis, M. Sutherland, and B. Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *ASPLOS*, 2019.

[11] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[12] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting congestion control for consistent high performance. In *NSDI*, 2015.

[13] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of cloudlab. In *ATC*, 2019.

[14] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, 2019.

[15] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *ICAC*, 2014.

[16] R. Haecki, R. N. Mysore, L. Suresh, G. Zellweger, B. Gan, T. Merrifield, S. Banerjee, and T. Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *NSDI*, 2022.

[17] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, 2017.

[18] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *NSDI*, 2019.

[19] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[20] B. C. Kuszmaul, M. Frigo, J. M. Paluska, and A. S. Sandler. Everyone loves file: File storage service (FSS) in oracle cloud infrastructure. In *ATC*, 2019.

[21] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid*, 2010.

[22] J. C. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 1997.

[23] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, 2013.

[25] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[26] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[27] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *OSDI*, 2020.

[28] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, et al. Autopilot: workload autoscaling at google. In *EuroSys*, 2020.

[29] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. Martin, M. McLaren, P. Chandra, R. Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *SIGCOMM*, 2020.

[30] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu. Distributed resource management across process boundaries. In *SoCC*, 2017.

[31] B. van Klinken. Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

[32] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *SIGOPS European Workshop*, 2002.

[33] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Characterizing facebook's memcached workload. *IEEE Internet Computing*, 2013.

[34] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang. Overload control for scaling wechat microservices. In *SoCC*, 2018.

TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs

Weiyang Wang[*] *Moein Khazraee*[*] *Zhizhen Zhong*[*] *Manya Ghobadi*[*]
Zhihao Jia[†,‡] *Dheevatsa Mudigere*[†] *Ying Zhang*[†] *Anthony Kewitsch*[§]

[*]*Massachusetts Institute of Technology* [†]*Meta* [‡]*CMU* [§]*Telescent*

Abstract

We propose TopoOpt, a novel direct-connect fabric for deep neural network (DNN) training workloads. TopoOpt co-optimizes the distributed training process across three dimensions: computation, communication, and network topology. We demonstrate the mutability of AllReduce traffic, and leverage this property to construct efficient network topologies for DNN training jobs. TopoOpt then uses an alternating optimization technique and a group theory-inspired algorithm called TotientPerms to find the best network topology and routing plan, together with a parallelization strategy. We build a fully functional 12-node direct-connect prototype with remote direct memory access (RDMA) forwarding at 100 Gbps. Large-scale simulations on real distributed training models show that, compared to similar-cost Fat-tree interconnects, TopoOpt reduces DNN training time by up to 3.4×.

1 Introduction

Our society is rapidly becoming reliant on deep neural networks (DNNs). New datasets and models are invented frequently, increasing the memory and computational requirements for training. This explosive growth has created an urgent demand for efficient distributed DNN training systems.

Today's DNN training systems are built on top of traditional datacenter clusters, with electrical packet switches arranged in a multi-tier Fat-tree topology [47]. Fat-tree topologies are traffic oblivious fabrics, allowing uniform bandwidth and latency between server pairs. They are ideal when the workload is unpredictable and consists mostly of short transfers–two inherent properties of legacy datacenter workloads [49, 50, 54, 67, 68]. But Fat-tree networks are becoming a bottleneck for distributed DNN training workloads [58, 69, 77, 85, 102, 105, 136].

Previous work has addressed this challenge by reducing the size of parameters to transmit through the network [48, 58, 59, 69, 73, 79, 82, 83, 94, 105, 123, 139] and developing techniques to discover faster parallelization strategies while considering the available network bandwidth [46, 48, 85, 105, 129]. These proposals co-optimize computation and communication as two important dimensions of distributed DNN training, but they do not consider the *physical layer topology* as an optimization dimension.

In this paper, we analyze DNN training jobs from production clusters of Meta. We demonstrate that training workloads do not satisfy common assumptions about datacenter traffic that underlie the design of Fat-tree interconnects. Specifically, we show that (*i*) the communication overhead of large DNN training jobs increases dramatically as we increase the number of workers; and (*ii*) the traffic pattern of a DNN training job depends on its parallelization strategies.

Motivated by these observations, we propose TopoOpt, a direct-connect DNN training system that co-optimizes network topology and parallelization strategy. TopoOpt creates dedicated partitions for each training job using reconfigurable optical switches and patch panels, and jointly optimizes the topology and parallelization strategy within each partition. To achieve our goal, we grapple with the *algorithmic* challenges of finding the best topology, such as how to navigate the large search space across computation, communication, and topology dimensions, and also with various *operational* challenges, such as which optical switching technologies match well with the traffic patterns of DNN models.

We cast the topology and parallelization strategy co-optimization problem as an off-line alternating optimization framework. Our optimization technique *alternates* between optimizing the parallelization strategy and optimizing the network topology. It searches over the parallelization strategy space assuming a fixed topology, and feeds the traffic demand to a TopologyFinder algorithm. The updated topology is then fed back into the parallelization strategy search algorithm. This alternating process repeats until the system converges to an optimized parallelization strategy and topology.

We demonstrate that finding an optimized network topology for DNNs is challenging because the ideal network topology needs to meet two goals simultaneously: (*i*) to complete large AllReduce transfers efficiently, and (*ii*) to ensure a small

hop-count for Model Parallel transfers. To meet these goals, we propose a novel *group theory-based technique*, called TotientPerms, that exploits the *mutability* of AllReduce transfers. Our TotientPerms approach builds a series of *AllReduce permutations* that not only carry AllReduce transfers efficiently, but are also well-positioned to carry Model Parallel transfers and, hence, improve the overall training performance.

Optical circuit-switched networks traditionally support point-to-point traffic across hosts with direct circuits between them. As a result, for a given set of circuits, only directly connected hosts can communicate leaving the rest of the hosts wait for new circuits to be established. To support arbitrary communication across all hosts participating in a job, we enable TopoOpt's hosts to act as relays and forward the traffic that does not belong to them. Host-based forwarding introduces a new challenge for RDMA flows since RDMA NICs drop packets that do not belong to them. To enable host-based RDMA forwarding, we exploit the network partition (NPAR) function of modern NICs, creating an efficient logical overlay network for RDMA packet forwarding (§6).

To evaluate TopoOpt, we build a 12-server prototype with NVIDIA A100 GPUs [37], 100 Gbps NICs and a Telescent reconfigurable optical patch panel [43]. Moreover, we integrate our TotientPerms AllReduce permutations into NCCL and enable it to load-balance parameter synchronization across multiple ring-AllReduce sub-topologies. Our evaluations with six representative DNN models (DLRM [20], CANDLE [4], BERT [134], NCF [75], ResNet50 [74], and VGG [126]) show that TopoOpt reduces the training iteration time by up to 3.4× compared to a similar-cost Fat-tree. Moreover, we demonstrate that TopoOpt is, on average, 3.2× cheaper than an ideal full bisection bandwidth Fat-tree. TopoOpt is the first system that co-optimizes topology and parallelization strategy for ML workloads and is currently being evaluated for deployment at Meta. The source code and scripts of TopoOpt are available at `https://topoopt.csail.mit.edu`.

2 Motivation

Prior research has illustrated that *demand-aware* network fabrics are flexible and cost-efficient solutions for building efficient datacenter-scale networks [64, 68, 113]. However, predicting the upcoming traffic distribution is challenging in a traditional datacenter setting. This section demonstrates that DNN training workloads present a unique opportunity for demand-aware networks, as the jobs are long-lasting, and the traffic distribution can be *pre-computed* before the jobs start to run. First, we provide the necessary background to understand distributed DNN training and introduce three types of data dependencies between accelerator nodes in training jobs (§2.1). Then, we present measurements from production clusters in Meta (§2.2), and discuss the important properties of DNN training traffic.

2.1 Background on Distributed DNN training

Training iteration. A common approach to training DNNs is stochastic gradient descent (SGD) [90]. Each SGD *iteration* involves selecting a random batch of labeled training data, computing the error of the model with respect to the labeled data, and calculating gradients for the model's weights through backpropagation. The SGD algorithm seeks to update the model weights so that the next evaluation reduces the error [55]. Training iterations are repeated with new batch of data until the model converges to the target accuracy.

Data parallelism. Data parallelism is a popular parallelization strategy, whereby a batch of training samples is distributed across training accelerators. Each accelerator holds a replica of the DNN model and executes the forward and backpropagation steps locally. In data parallelism, all accelerators synchronize their model weights during each training iteration. This step is commonly referred to as *AllReduce* and can be performed using various techniques, such as broadcasting [141], parameter servers [93], ring-AllReduce [3, 83, 130], tree-AllReduce [116], or hierarchical ring-AllReduce [131, 133].

Hybrid data and model parallelism. Large DNN models cannot fit in the memory of a single accelerator or even a single server with multiple accelerators. As a result, the model needs to be divided across multiple accelerators using *model parallelism* [84, 92]. Moreover, pure data parallelism is becoming suboptimal for large training jobs because of the increasing cost of synchronizing model parameters across accelerators [20, 78, 85, 104, 106, 125]. As a result, large DNNs are distributed using a hybrid of data and model parallelism, where different parts of a DNN and its dataset are processed on different accelerators in parallel.

Types of data dependencies in DNN training. Each training iteration includes two major types of *data dependencies*. Type (1) refers to *activations* and *gradients* computed during the Forward and Backpropagation steps. This data dependency is required for each input sample. Type (2) refers to synchronizing the *model weights* across accelerators through the AllReduce step once a batch of samples is processed. Depending on the parallelization strategy, these data dependencies may result in local memory accesses or cross-accelerator traffic. For instance, in a hybrid data and model parallelization strategy, type (1) and (2) both result in cross-accelerator traffic, depending on how the model is distributed across accelerators. Given that type (1) is related to model parallelism, we refer to the network traffic created by type (1) as *MP transfers*. Similarly, we refer to the network traffic created by type (2) as *AllReduce transfers*. Note that AllReduce transfers do not strictly mean data parallelism traffic, as model parallelism can also create AllReduce transfers across a subset of nodes.

Example: DLRM traffic pattern. Deep Learning Recommendation Models (DLRMs) are a family of personalization and recommendation models based on embedding table lookups that capitalize on categorical user data [107]. DLRMs

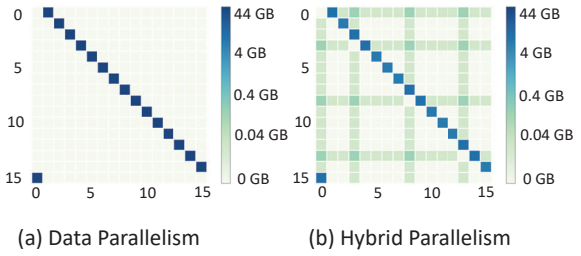


Figure 1: DLRM traffic heatmaps for different parallelization strategies.

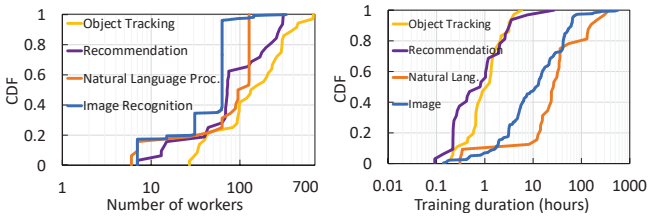


Figure 2: Profiling distributed DNN training jobs in Meta.

are large, typically with 100s of billions of parameters, primarily because of their large embedding tables. Using pure data parallelism to distribute a DLRM results in massive AllReduce transfers. For instance, consider a DLRM architecture with four embedding tables $E_0, \cdots, E_3$, each with embedding dimensions of 512 columns and 10^7 rows (total size 22 GB for the model) distributed across 16 servers $S_0, \cdots, S_{15}$ with data parallelism. We compute the resulting traffic distribution, and Figure 1a illustrates the traffic pattern for a single training iteration. The rows and columns indicate source and destination servers, while the color encodes the amount of traffic between server pairs. The heatmap shows that using ring-AllReduce for synchronization, a pure data parallelism strategy results in 44 GB of AllReduce transfers.

Hence, a common parallelization strategy for DLRMs is to use a hybrid of data and model parallelism where the embedding tables are divided across nodes, while the rest of the model is replicated on all nodes [102]. Following the parallelization strategy used at Meta, we place E_0 on S_0, E_1 on S_3, E_2 on S_8, and E_3 on S_{13}, and replicate the rest of the model on all servers. This parallelization strategy creates a mix of MP and AllReduce traffic, shown in Figure 1b. It reduces the maximum transfer size from 44 GB to 4 GB.

Note that MP transfers in DLRM form one-to-many broadcast and many-to-one incast patterns to transfer the activation and gradients to all nodes because the servers handling embedding tables must communicate with *all other servers*. In this example, the size of each AllReduce transfer is 4 GB, whereas the size of MP transfers is 32 MB, as shown by darker green elements in the heatmap.

2.2 Production Measurements

We study traffic traces from hundreds of production DNN training jobs running on multiple clusters at Meta. We instrument each job to log its training duration, number of workers, and the total amount of data transferred across its workers during training.

Number of workers and job duration. Figure 2a shows the cumulative distribution function (CDF) of the number of workers for different models in Meta's clusters. Most jobs are distributed across 32 to 700 workers, agreeing with recent announcements by other major players in the industry [45, 104], where each worker is a single GPU. Figure 2b demonstrates the CDF of total training job duration; as the figure shows, most jobs last over 10 hours. In fact, the top 10% of jobs take more than 96 hours (four days) to finish. This measurement shows production DNN jobs at Meta are long-lasting, and take up to weeks to finish.

Network overhead. Figure 3 illustrates the percentage of network overhead as the number of GPUs is increased from 8 to 128 for six DNN jobs in production. We use RDMA to transmit packets between servers and measure the percentage of time consumed by communication during training as network overhead. The figure shows that as the number of GPUs increases, the network quickly takes up a significant portion of training iteration time. In fact, the network overhead accounts for up to 60% of a DNN training iteration time in Meta's production environment. Similar observations have been made in prior work [59, 77, 89, 105, 110, 123]. Such bottleneck suggests the existing datacenter networks are insufficient for the emerging DNN training workloads.

Traffic heatmaps. Figure 4 shows the heatmap of server-to-server traffic for four training jobs running in Meta's production GPU clusters. The values on the colormap and the exact names of DNN models are not shown for confidentiality reasons. All heatmaps in the figure contain diagonal squares (in dark blue), indicating a ring communication pattern between servers. This is expected, as ring-AllReduce is the common AllReduce communication collective at Meta. But the MP transfers (light blue and green squares) are *model-dependent* because MP transfers depend on the parallelization strategy and device placement of a training job. Moreover, we find that the traffic patterns of training jobs do not change between iterations *for the entire training duration*, resulting in the same per-iteration heatmap throughout the training. Once a training job starts, the same parallelization strategy and synchronization method are used across training iterations, resulting in a periodic and predictable traffic pattern. Similar observations have been made in previous work [140]. In particular, the traffic heatmap is identical *across* training iterations. Note that the traffic pattern changes *within* a training iteration during forward, backward, and AllReduce phases.

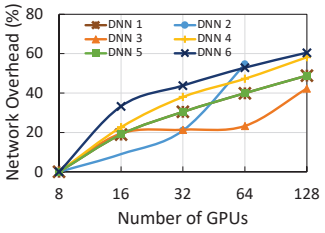


Figure 3: Network overhead measurements in Meta.

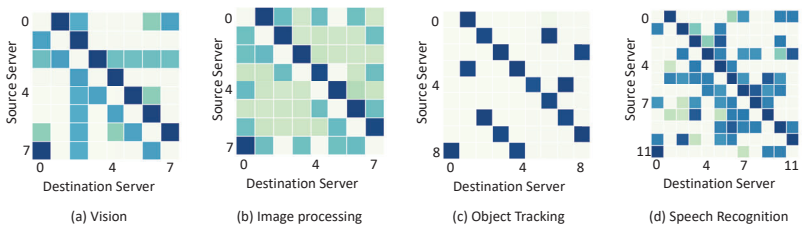


Figure 4: Traffic heatmaps of production jobs in Meta.

3 TopoOpt System Design

The observations in the previous section suggest that demand-aware fabrics are excellent candidates for a DNN training cluster. In this section, we seek to answer the following question: *"Can we build a demand-aware network to best support distributed training?"* To answer this question, we propose TopoOpt, a novel system based on optical devices that jointly optimizes DNN parallelization strategy and topology to accelerate today's training jobs.

TopoOpt interconnect. A TopoOpt cluster is a *shardable* direct-connect fabric where each server has d interfaces connected to a core layer of d optical switches, as shown in Figure 5. The optical switches enable TopoOpt to shard the cluster into dedicated partitions for each training job. The size of each shard depends on the number of servers the job requests. Given a DNN training job and a set of servers, TopoOpt first finds the best parallelization strategy and topology between the servers off-line (§4.1). Then, it reconfigures the optical switches to realize the target topology for the job. Appendix C provides details on how TopoOpt achieves sharding and dynamic job arrivals in shared clusters.

Degree of each server. We denote the number of interfaces on each server (i.e., the degree of the server) by d. Typically, d is the same as the number of NICs installed on the server. In cases where the number of NICs is limited, the degree can be increased using NICs that support break-out cables or the next generation of co-packaged optical NVLinks [11]. In our testbed, we use one 100 Gbps HPE NIC [29] with 4×25 Gbps interfaces to build a system with degree four ($d = 4$).

Direct-connect topology. In TopoOpt, optical switches connect the servers directly, forming a *direct-connect topology*. To further scale a TopoOpt cluster, we create a hierarchical interconnect by placing the servers under Top-of-Rack (ToR) switches and connecting ToR switches to the optical layer, creating a direct-connect topology at the ToR or spine layers, similar to previous work [53, 71, 72, 100, 114].

Host-based forwarding. In DNN training workloads, the degree of each server is typically smaller than the total number of neighbors with whom the server communicates during training. To ensure traffic is not blocked when there is no direct link between two servers, we use a technique called *host-based forwarding*, where hosts act as switches and forward incoming traffic toward the destination. Previous work used similar technique at the ToR switch level [53, 99, 100].

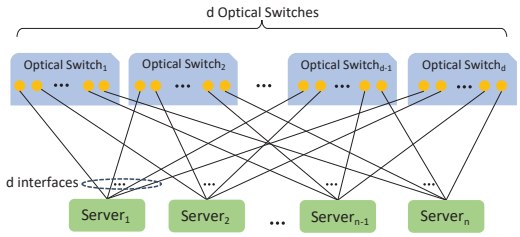


Figure 5: Illustration of TopoOpt's interconnect.

Optical switching technologies. A wide range of optical switches is suitable for TopoOpt, including commodity available optical patch panels [43] and 3D-MEMS [6, 41], as well as futuristic designs such as Mordia [113], MegaSwitch [57], and Sirius [53, 60]. Table 1 lists the characteristics of these devices. TopoOpt is compatible with any of these technologies. Appendix B provides details about these devices.

One-shot reconfiguration. Patch panel and OCS are both applicable for an immediate deployment of TopoOpt, as shown in Table 1. The choice of which technology to use depends on several factors, including scale of the cluster, iteration time of jobs, and frequency of job arrivals. For instance, OCSs can potentially be used to reconfigure the topology of a job *within* training iterations, whereas patch panels are only suitable when the topology remains intact throughout the entire training of a particular job. Our evaluations demonstrate that the reconfiguration latency of today's OCSs is too high for today's DNNs, leading to sub-optimal performance when the topology is reconfigured within iterations (§5). As a result, given that faster technologies are not yet available, TopoOpt uses a one-shot reconfiguration technique based on an offline co-optimization framework (§4) that jointly optimizes the parallelization strategy and topology. TopoOpt then reconfigures the interconnection between training servers of each job before the job starts and keeps the topology intact until the training is complete (or to recover from failures).

Technology	Port-count	Reconfig. latency	Insertion Loss (dB)	Cost /port
Optical Patch Panels [43]	1008	minutes	0.5	$100
3D MEMS [6, 41]	384	10 ms	1.5–2.7	$520
2D MEMS [57, 113]	300	11.5 μs	10–20	Not commercial
Silicon Photonics [89, 122]	256	900 ns	3.7	Not commercial
Tunable Lasers [53, 60]	128	3.8 ns	7-13	Not commercial
RotorNet [99, 100]	64	10 μs	2	Not commercial

Table 1: Comparison of optical switching technologies.

4 Co-optimizing Parallelization Strategy and Network Topology

This section describes TOPOOPT's co-optimization framework for finding a network topology and parallelization strategy for a given DNN training job.

4.1 Alternating Optimization

The search space is too large. Finding the optimal parallelization strategy alone is an NP-complete problem [85], and adding network topology and routing makes the problem even harder. An extreme solution is to jointly optimize compute, communication, and topology dimensions using a *cross-layer optimization formulation*. Theoretically, this approach finds the optimal solution, but the search space quickly explodes, even at modest scales (e.g., six nodes [129]).

Naive approach. The other extreme is to optimize the network topology *sequentially after* the parallelization strategy has been found. While this approach is able to reconfigure the network to better match its traffic demand, the eventual combination of topology and parallelization strategy is likely to be sub-optimal in the global configuration space.

TOPOOPT's approach: alternating optimization. In TOPOOPT, we seek to combine the best of both worlds. To make the problem tractable, we divide the search space into two planes: *Comp.* $\times$ *Comm.* and *Comm.* $\times$ *Topo.* We use an alternating optimization technique to iteratively search in one plane while keeping the result of the other plane constant. Figure 6 illustrates our alternating optimization framework. We use FlexFlow's MCMC (Markov Chain Monte Carlo) search algorithm [85] to find the best parallelization strategy for a given network topology while considering the communication cost. If the parallelization strategy improves the training iteration time, we feed it to the *Comm.* $\times$ *Topo.* plane to find the efficient network topology and routing using our TOPOLOGYFINDER algorithm. The discovered topology is then fed back into the *Comp.* $\times$ *Comm.* plane, which further optimizes the parallelization strategy and device placement based on the new topology. This optimization loop repeats until convergence or after k iterations, where k is a configurable hyper-parameter.

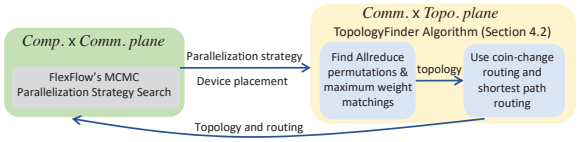


Figure 6: TOPOOPT searches for the best parallelization strategy, jointly with routing, and topology.

4.2 TOPOLOGYFINDER Algorithm

TOPOLOGYFINDER steps. Algorithm 1 presents the pseudocode of our TOPOLOGYFINDER procedure. The algorithm takes the following inputs: n dedicated servers for the training job, each with degree d, as well a list of AllReduce and MP transfers ($T_{AllReduce}$ and T_{MP}) based on the parallelization strategy and device placement obtained from the *Comp.* $\times$ *Comm.* plane. The algorithm then finds the best topology (G) and routing rules (R) and returns them to the *Comp.* $\times$ *Comm.* plane for the next round of alternating optimization. Our algorithm consists of the following four steps.

Step 1: Distribute the degree. This step distributes the degree d between AllReduce and MP sub-topologies proportionally, based on their share of total traffic. We specifically start with AllReduce transfers and allocate at least one degree to the AllReduce sub-topology to ensure the network remains connected (line 2). The remaining degrees, if any, are allocated to the MP sub-topology (line 3).

Step 2: Construct the AllReduce sub-topology. To find the AllReduce sub-topology, the algorithm iterates over every AllReduce group k and allocates degree d_k to each group proportionally based on the amount of traffic (line 6). Note that in hybrid data and model parallelism strategies, the AllReduce step can be performed across a subset of servers when a DNN layer is replicated across a few servers instead of all servers. To efficiency serve both AllReduce and MP transfers, TOPOOPT constructs the AllReduce sub-topology such that the diameter of the cluster is minimized. Section 4.3 explains two algorithms, called `TotientPerms` (line 8) and `SelectPermutations` (line 9) to construct the AllReduce sub-topology.

Step 3: Construct the MP sub-topology. We use the Blossom maximum weight matching algorithm [63] to find the best connectivity between servers with MP transfers (line 14). We repeat the matching algorithm until we run out of degrees. To increase the likelihood of more diverse connectivity across server pairs, we divide the magnitude of T_{MP} for pairs that already have an edge between them by two (line 17). In general, division by two can be replaced by a more sophisticated function with a diminishing return.

Step 4: Final topology and routing. Finally, we combine the MP and AllReduce sub-topologies to obtain the final topology (line 18). We then use a modified version of the coin-change algorithm [52] (details in Appendix E.1) to route

Algorithm 1 TOPOLOGYFINDER pseudocode

1: **procedure** TOPOLOGYFINDER(n, d, $T_{AllReduce}$, T_{MP})
▷ **Input** n: Number of dedicated training servers for the job.
▷ **Input** d: Degree of each server.
▷ **Input** $T_{AllReduce}$: AllReduce transfers.
▷ **Input** T_{MP}: MP transfers.
▷ **Output** G: Topology to give back to the $Comp. \times Comm.$ plane.
▷ **Output** R: Routing rules to give back to the $Comp. \times Comm.$ plane.
▷ *Distribute degree d between AllReduce and MP sub-topologies*
2: $d_A = \max(1, \lceil d \times \frac{sum(T_{reduce})}{sum(T_{reduce})+sum(T_{MP})} \rceil)$
3: $d_{MP} = d - d_{AllReduce}$
▷ *Construct the AllReduce sub-topology $G_{AllReduce}$*
4: $G_{AllReduce} = \{\}$
5: **for** each AllReduce group k with set of transfers T_k **do**
▷ *Assign degree d_k to group k according to its total traffic*
6: $d_k = \lceil d_A \times \frac{sum(T_k)}{sum(T_{reduce})} \rceil$
7: $d_A = d_A - d_k$
▷ *Find all the permutations between servers in group k*
8: P_k = `TotientPerms(`n, k`)` ▷ *(Details in §4.3)*
▷ *Select d_k permutations from P_k*
9: $G_{AllReduce} = G_{AllReduce} \cup$ `SelectPermutations(`n, d_k, P_k`)` ▷ *(§4.3)*
10: **if** $d_{AllReduce} == 0$ **then**
11: break
▷ *Construct the MP sub-topology G_{MP}*
12: $G_{MP} = \{\}$
13: **for** $i : i < d_{MP}$ **do**
▷ *Find a maximum weight matching according to T_{MP}*
14: $g =$ `BlossomMaximumWeightMatching(`T_{MP}`)`
15: $G_{MP} = G_{MP} \cup g$
▷ *Reduce the amount of demand for each link l in graph g*
16: **for** $l \in g$ **do**
17: $T_{MP}[l] = T_{MP}[l] / 2$
▷ *Combine the AllReduce and MP topologies*
18: $G = G_{AllReduce} \cup G_{MP}$
▷ *Compute routes on $G_{AllReduce}$ using the coin change algorithm [52]*
19: $R =$ `CoinChangeMod(`n, $G_{AllReduce}$`)` ▷ *(Appendix §E.1)*
▷ *Compute routes on G_{MP} with shortest path*
20: R += `ShortestPath(`G, T_{MP}`)`
21: **return** G, R

AllReduce on the AllReduce sub-topology (line 19). Further, we use k-shortest path routing for the MP transfers to take advantage of the final combined topology (line 20).

4.3 Traffic Mutability and AllReduce Topology

Finding an efficient AllReduce sub-topology. At first blush, finding an AllReduce sub-topology for a given DNN seems straightforward: we just need to translate the parallelization strategy and device placement from the $Comp. \times Comm.$ plane into a traffic matrix and map the traffic matrix into circuit schedules. Several papers have used this technique for datacenter networks [57, 64, 68, 72, 89, 95–97, 113, 137]. However, the conventional wisdom in prior work is to allocate as many direct parallel links as possible to *elephant flows* and leave *mice flows* to take multiple hops across the network. In principle, this approach works well for datacenters but it leads to sub-optimal topologies for distributed DNN training. While the size of AllReduce transfers is larger than MP transfers, MP transfers have a higher communication degree than AllReduce (Appendix D). Hence, the conventional approach creates parallel direct links for carrying AllReduce traffic and forces MP flows to have a large hop-count, thereby degrading the training performance.

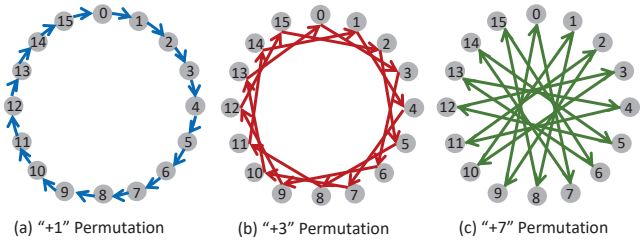


Figure 7: Ring-AllReduce permutations.

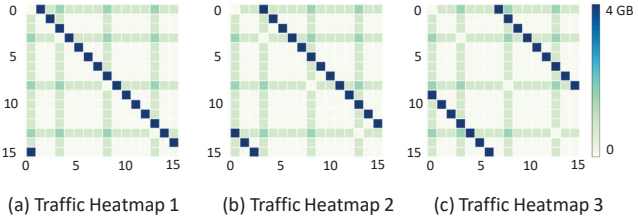


Figure 8: DLRM traffic heatmaps.

TOPOOPT's novel technique. In TOPOOPT, we seek to meet two goals simultaneously: (*i*) allocate ample bandwidth for AllReduce transfers, as the bulk of the traffic belongs to them, but (*ii*) ensure a small hop-count for MP transfers. We meet both goals by demonstrating a unique property of DNN training traffic – the AllReduce traffic is *mutable*.

Mutability of AllReduce transfers. We define traffic mutability as the ability to change the traffic pattern without altering parallelization strategy or device placement while maintaining correctness, and demonstrate that AllReduce transfers are *mutable* whereas MP transfers are not. Intuitively, this is because MP traffic is composed of network flows among nodes that contain *different* parts of a DNN model thus creating immutable data dependencies, while AllReduce transfers contain network flows among nodes that handle the *same* part of the model, providing flexibility in the order of nodes participating in AllReduce. For instance, consider a DLRM distributed across 16 servers each with three NICs. The common AllReduce pattern is shown as a ring with consecutive node IDs, as shown in Figure 7a. However, *this is not the only possible permutation*. Each heatmap in 8a, 8b, and 8c corresponds to a different ring-AllReduce permutation, shown in Figures 7a, 7b, and 7c. We denote each of these permutations as $+p$, where server S_i connects to server $S_{(i+p)\%n}$, and n is the number of servers, as shown in Figure 7. Although all three heatmaps correspond to the *exact same parallelization strategy and device placement*, the blue diagonal lines appear at different parts of the heatmaps, depending on the order of servers in the ring-AllReduce permutation. But MP transfers (green vertical and horizontal lines in each heatmap) are dictated by the parallelization strategy and device placement; thus, they remain at the same spot in all three heatmaps.

Algorithm 2 `TotientPerms` pseudocode

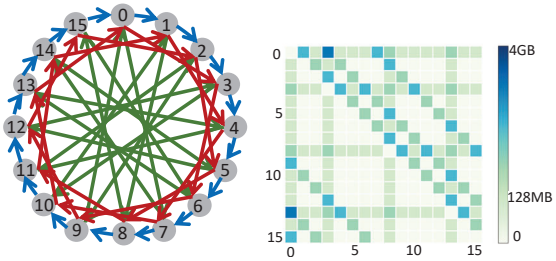
```
1:  procedure TotientPerms(n, k)
      ▷ Input n: Total number of nodes
      ▷ Input k: AllReduce group size
      ▷ Output P_k: Set of permutations for AllReduce group of size k
        ▷ Initially, P_k is empty
2:    P_k = {}
        ▷ This loop runs ϕ(p) times, where
        ▷ ϕ is the Euler Totient function, ϕ(p) = |{k < p : gcd(k, p) = 1}|
        ▷ one can also restrict p to be prime only
3:    for p ≤ k, gcd(p,k) == 1 do
4:      one_perm = []
5:      for i in 0 to N/k do
6:        one_perm += [i + j × p for j in 0 to k]
7:      P_k += one_perm
8:    return P_k
```

Algorithm 3 `SelectPermutations` pseudocode

```
1:  procedure SelectPermutations(n, d_k, P_k)
      ▷ Input n: Total number of nodes
      ▷ Input d_k: Degree allocated for group this AllReduce group of size k
      ▷ Input P_k: Candidate permutations for this AllReduce group of size k
      ▷ Output G_k: Parameter synchronization topology, given as a set of
      permutations
        ▷ Initially, G_k is empty
2:    G_k = {}
        ▷ q now is the minimum candidate in P_k
3:    q = P_k[0]
        ▷ GetConn(q) gives the connection described
        ▷ by the permutation corresponding to q
4:    G_k = G_k ∪ GetConn(q)
        ▷ Ratio of the geometric sequence to fit
5:    x = ∜N
6:    for i ∈ {1, ··· , d_k − 1} do
          ▷ Select the next candidate based on the ratio
7:      q' = x × q
          ▷ Project q' onto P_k \ G_k with minimal distance (L1-norm)
8:      q' = argmin_{r∈P_k\G_k} |r − q'|
          ▷ Add this candidate to final topology
9:      G_k = G_k ∪ GetConn(q')
10:     q = q'
11:   return G_k
```

Leveraging AllReduce traffic mutability. Traffic mutability implies that if a group of servers is connected in a certain order, simply permuting the label of the servers gives another ordering that will finish the AllReduce operation with the same latency while potentially providing a smaller hop-count for MP transfers. Instead of selecting just one AllReduce order, TopoOpt finds multiple permutations for each AllReduce group and overlaps their corresponding sub-topologies. In doing so, TopoOpt efficiently serves the AllReduce traffic while decreasing the hop-count for MP transfers.

TotientPerms algorithm. While overlapping multiple permutations sounds straightforward, navigating through the set of all possible AllReduce orderings is non-trivial, since the number of possible permutations is $O(n!)$. To reduce the search space of all possible permutations, we design the TotientPerms algorithm to find the ring generation rule for all *regular rings*, based on group theory. Regular rings are those where the distance between indices of consecutive servers is equal; i.e., server S_i is connected to server $S_{(i+p)\%n}$ for some p. Algorithm 2 presents the pseudocode of `TotientPerms`. Inspired by Euler's totient function [25], we find all integer numbers $p < n$, where p is co-prime with n (i.e. $gcd(p,n) = 1$, line 3, Algorithm 2), represent a valid ring-AllReduce permutation (§E.1). For instance, for $n = 12$ servers, the ring generation rule for $p = 1, 5, 7, 11$ will lead into four distinct ring-AllReduce permutations between the servers. Note that each p describes a unique regular permutation. To handle large scale clusters, we restrict p to be a prime number, thereby reducing the search space size to only $O(\frac{n}{\ln n})$, as per the Prime Number Theorem [66].



(a) TopoOpt topology (b) TopoOpt traffic pattern

Figure 9: TopoOpt's topology and traffic matrix.

SelectPermutations algorithm. For a group of n servers participating in AllReduce, `TotientPerms` finds a set of regular permutations $P_k = \cup_{p:gcd(p,n)=1}\{p\}$ across them. TopologyFinder then selects d_k permutations using a module called `SelectPermutations`, where d_k is the number of degree allocated to the group of nodes running AllReduce (line 6, Algorithm 1). Algorithm 3 presents the pseudocode of `SelectPermutations`. Several metrics can be used in the `SelectPermutations` module. In our implementation, `SelectPermutations` aims to reduce the cluster diameter to benefit the MP transfers. To this end, `SelectPermutations` chooses $\{p_1, \cdots, p_{d_k}\} \subset P_k$, such that $\{p_1, \cdots, p_{d_k}\}$ is close (in L1-norm) to a geometric sequence (line 7, Algorithm 3).

Theorem 1. *TopoOpt's `SelectPermutations` algorithm bounds the diameter of the AllReduce sub-topology to $O(d_A \cdot n^{1/d_A})$, under certain assumptions.*

We list the assumptions and proof of Theorem 1 in Appendix E.2. Intuitively, each server in the topology is able to reach a set of servers with a geometrically distributed hop-count distance (line 5, Algorithm 3), creating a topology similar to Chord [128].

Example. Consider the DLRM model in Figure 8. Instead of choosing one of the AllReduce permutations in Figure 7, TopoOpt combines the three ring-AllReduce permutations to load-balance the AllReduce transfers while providing a short hop-count for MP transfers. Figure 9 illustrates TopoOpt's topology and traffic matrix and shows a more balanced traffic matrix than Figure 8.

5 Large Scale Simulations

This section evaluates the performance of a large-scale TopoOpt interconnect. First, we explain our simulation software and methodology (§5.1). Then, we provide a cost analysis of TopoOpt to inform our simulations when comparing different interconnects (§5.2). Next, we demonstrate the performance of TopoOpt when a cluster is dedicated to a single distributed DNN training job (§5.3). We perform a sensitivity analysis to quantify the impact of all-to-all traffic (§5.4) and host-based forwarding (§5.5). We extend this setting to a case where a training cluster is shared among multiple jobs (§5.6). Finally, we evaluate the impact of reconfiguration latency (§5.7) on TopoOpt's performance.

5.1 Methodology & Setup

We implement two simulators to evaluate TopoOpt.

***FlexNet* simulator.** We augment FlexFlow's simulator [27] to be network-aware and call it *FlexNet*. Given a DNN model and a batch size, FlexFlow's simulator explores different parallelization strategies and device placements to minimize iteration training time. The output of this simulator is a *task graph* describing the set of computation and communication tasks on each GPU and their dependencies. The current implementation of FlexFlow ignores the network topology by assuming servers are connected in a *full-mesh* interconnect. Our *FlexNet* simulator extends the FlexFlow simulator and enables it to consider multiple networks, including Fat-trees, TopoOpt, and expander networks. Moreover, *FlexNet* implements our alternating optimization framework (§4) to find an optimized network topology and routing rules for TopoOpt.

***FlexNetPacket* simulator.** FlexFlow's simulator only provides course-grind estimation of training iteration time, because it does not simulate individual packets traversing through a network. Extending *FlexNet* to become a packet-level simulator is computationally infeasible, because FlexFlow generally requires thousands of MCMC iterations to converge. To faithfully simulate per-packet behavior of network switches, buffers, and multiple jobs sharing the same fabric, we build a second event-based packet simulator, called *FlexNetPacket*, on top of htsim [7]. *FlexNetPacket* takes the output of *FlexNet* (i.e., the optimized parallelization strategy, device placement of each operator, network topology, and routing rules) and simulates several training iterations. The link propagation delay is set to 1 μs throughout this section.

Simulated network architectures. We simulate distributed training clusters with n servers equipped with four NVIDIA A100 GPUs [37]. We vary n in different experiments and simulate the following network architectures:

- **TopoOpt.** A TopoOpt interconnect where each server is equipped with d NICs, each with bandwidth B connected via a flat layer of optical devices. At the beginning of each job, a shard of the network is selected, and the topology of the shard is reconfigured based on the output of our alternating optimization framework (§4) and remains unchanged throughout the entire training job. Both OCS and patch panels are suitable for this architecture.
- **OCS-reconfig.** To study the impact of changing the network topology within training iterations, we simulate a reconfigurable TopoOpt interconnect. We rely on commercially available Optical Circuit Switches (OCSs) for this design and assume the reconfiguration latency is 10 ms. Given that FlexFlow's parallelization strategy search is not aware of dynamically reconfigurable networks, following prior work [89], we measure the traffic demand every 50 ms and adjust the circuits based on a heuristic algorithm to satisfy the current traffic demand as much as possible. We also enable host-based forwarding such that the communication is not blocked even when a direct link is not available (Appendix E.4).
- **Ideal Switch.** An ideal electrical switch that scales to any number of servers, where each server is connected to the switch via a link with $d \times B$ bandwidth. For any pair of d and B, no network can communicate faster than this ideal case. In practice, the Ideal Switch can be approximated with a full-bisection bandwidth Fat-tree where the bandwidth of each link is $d \times B$.
- **Fat-tree.** To compare the performance of TopoOpt to that of a similar-cost Fat-tree architecture, we simulate a full bisection bandwidth Fat-tree where each server has one NIC and the bandwidth of each link is $d \times B'$, where B' is lower than B and is selected such that Fat-tree's cost is similar to TopoOpt (§5.2).
- **Oversub. Fat-tree.** This is a 2:1 oversubscribed Fat-tree interconnect, where the bandwidth of each link is $d \times B$ but half of the links in the ToR uplink layer are omitted.
- **SiP-ML [89].** SiP-ML is a futuristic DNN training cluster with Tbps of bandwidth per GPU. While having a Tbps network is beneficial, our goal is to compare the algorithmic contributions of TopoOpt and SiP-ML. Hence, to make a fair comparison, we allocate d wavelengths, each with bandwidth B, to each SiP-ML GPU and follow its SiP-Ring algorithm to find a topology with a reconfiguration latency of 25 μs. Appendix F elaborates on our modifications to SiP-ML.
- **Expander [127, 135].** Finally, we simulate a fabric where each server has d NICs with bandwidth B interconnected via an Expander topology.

DNN Workloads. We simulate six real-world DNN models: DLRM [20], CANDLE [4], BERT [62], NCF [75], ResNet50 [74] , and VGG [126]. List 1 (Appendix D) provides details about model configurations and batch sizes used in this paper.

Parallelization strategy. We use *FlexNet*'s topology-aware parallelization strategy search for Ideal Switch, Fat-tree, Oversub. Fat-tree, SiP-ML, and Expander networks. For TopoOpt, we use *FlexNet*'s alternating optimization framework to find the best parallelization strategy jointly with topology, where the final parallelization strategy is either hybrid or pure data-

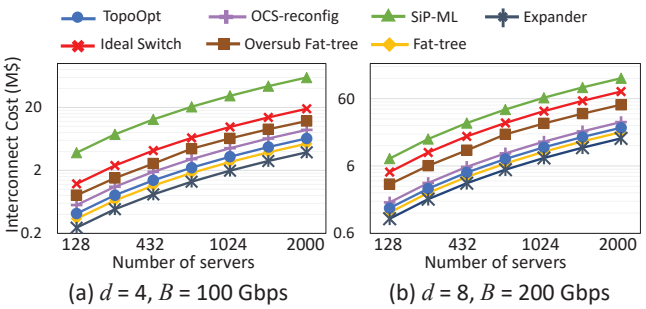


Figure 10: Interconnect cost comparison.

parallel. We use ring-AllReduce and distributed parameter server [93] as default AllReduce communication collectives between servers and within servers, respectively. Each data point averages 5–10 simulation runs.

5.2 Cost Analysis

We begin our evaluations by comparing the cost of various network architectures. Details about the cost of each component used in each architecture are given in Appendix G.

Figure 10 compares the interconnect cost across various network architectures as the number of servers is increased. We estimate the cost of Ideal Switch with a full-bisection Fat-tree of the same bandwidth. We make the following observations. First, using OCSs for TopoOpt is more expensive (1.33×, on average) than patch panels. Note that OCSs can be used in both TopoOpt and OCS-reconfig interconnects. Second, the cost of TopoOpt overlaps with that of the Fat-tree. This is intentional, because having a cost-equivalent architecture enables us to compare the performance of TopoOpt to a cluster at the same price point. Third, the ratio of Ideal Switch's cost to TopoOpt's cost is 3.2× on average. Finally, the most and least expensive fabrics are SiP-ML and Expander, respectively, and as this section shows, they both perform worse than TopoOpt for certain workloads.

We acknowledge that estimating the cost of networking hardware is challenging because prices are subject to significant discounts with bulk orders. Assuming all components in this analysis are subject to similar bulk order discounts, the relative comparison across architectures remains valid. As a point of comparison, we compute the cost of a cluster with 4,394 servers ($k = 26$ Fat-tree) by following the discounted cost trends in Sirius [53] and with 50% discounts for patch panels. For a cluster at this scale, the cost of full-bisection bandwidth Fat-tree (which approximates our Ideal Switch baseline) relative to the cost of TopoOpt changes from 3.0× to 3.6×, indicating our estimates are reasonable. Moreover, a TopoOpt cluster incurs lower energy cost than Fat-trees, as optical switches are passive.

5.3 Performance Comparison on Dedicated Clusters

This section compares the training iteration time of TopoOpt with that of other network architectures when the cluster is dedicated to serving one DNN training job.

Figure 11a compares the training iteration times of various architectures for CANDLE distributed on a dedicated cluster of 128 servers with a server degree of four ($d = 4$). We vary the link bandwidth (B) on the x-axis. The figure shows that Ideal Switch, TopoOpt, and SiP-ML architectures achieve similar performance because the best parallelization strategy for CANDLE at this scale is mostly data parallel, with few MP transfers. The OCS-reconfig architecture performs poorly because it uses the instantaneous demand as the baseline to estimate the future traffic to schedule circuits. This estimation becomes inaccurate during training, in particular when the current AllReduce traffic is about to finish but the next round of AllReduce has not started. The Expander architecture has the worst performance, as its topology is not optimized for DNN workloads. Averaging across all link bandwidths, compared to Fat-tree interconnect, TopoOpt improves the training iteration time of CANDLE by 2.8×; i.e., the ratio of CANDLE's iteration time on Fat-tree to TopoOpt is 2.8. TopoOpt's servers have more raw bandwidth, resulting in faster completion time.[1]

Figures 11b and 11c show the training iteration times for VGG and BERT. The trends are similar to CANDLE, as these models have similar degree requirements. Compared to Fat-tree, on average, TopoOpt improves the iteration time of VGG and BERT by 2.8× and 3×, respectively.

The cases of DLRM and NCF are more interesting, as they have more MP transfers than the other DNNs. As shown in Figures 11d and 11e, TopoOpt's performance starts to deviate from Ideal Switch, especially for NCF, because it uses host-based forwarding for the many-to-many MP transfers (§5.4 and §5.5). For DLRM (and NCF), TopoOpt is 2.8× (and 2.1×) faster than Fat-tree, while Ideal Switch further improves the training iteration time by 1.3× (and 1.7×) compared to TopoOpt. SiP-ML performs poorly, and even when we increase the link bandwidth, its training iteration time stays flat. This happens because MP transfers in DLRM and NCF require several circuit reconfigurations to meet the traffic demand.

Finally, Figure 11f shows most architectures achieve similar training iteration times for ResNet50 since it is not a communication-heavy model. The Expander architecture performs poorly when the link bandwidth is lower than 100 Gbps, as the topology does not match the AllReduce traffic pattern.

We repeat this simulation with $d = 8$ and observe a similar performance trend (Appendix H).

[1] It is possible to improve the performance of the Expander fabric by augmenting Blink's approach [136] to a cluster-level solution.

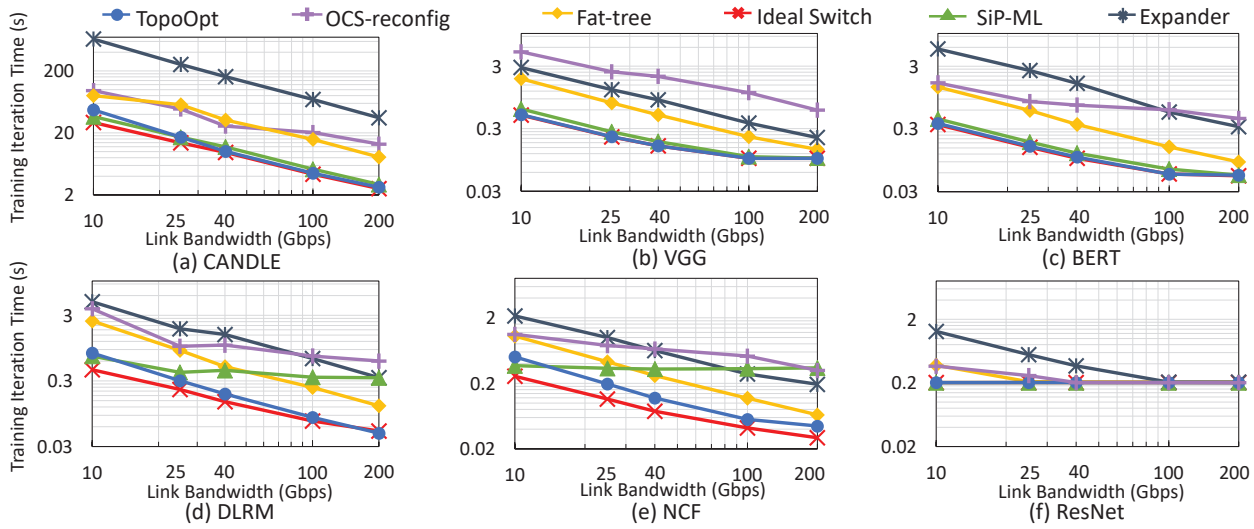


Figure 11: Dedicated cluster of 128 servers ($d = 4$).

5.4 Impact of All-to-all Traffic

This section evaluates the impact of all-to-all traffic patterns on TopoOpt's performance. In particular, TopoOpt's host-based forwarding approach incurs bandwidth tax [99] exacerbated by *all-to-all* and *many-to-many* communication patterns. This tax is defined as the ratio of the traffic volume in the network (including forwarded traffic) to the volume of logical communication demand. Hence, the bandwidth tax for a full bisection bandwidth Fat-tree topology is always one, because hosts do not act as relays for each other.

Consider a DNN model with R bytes of AllReduce traffic and A bytes of all-to-all traffic, distributed on a full bisection bandwidth topology with total network bandwidth $N \cdot B_F$ (i.e., number of servers multiplied by the bisection bandwidth). The training iteration time of this DNN is: $T_F = \frac{R}{N \cdot B_F} + \frac{A}{N \cdot B_F} + C_{bs}$, where C_{bs} is the computation time of the model with batch size bs.[2]

Now suppose the same DNN is distributed on a TopoOpt topology with total network bandwidth NB_T. In this case, assuming the entire AllReduce traffic is carried on Totient-Perms with direct links, the training iteration time becomes $T_T = \frac{R}{N \cdot B_T} + \frac{\alpha \cdot A}{N \cdot B_T} + C_{bs}$ (Eq. 1), where α represents the slowdown factor that all-to-all transfers create in the network, due to host-based forwarding. The value of α depends on the amount of bandwidth tax and routing strategy (§5.5).

Increasing the amount of all-to-all traffic (A) increases the iteration time for both T_F and T_T. But when $N \cdot B_F$ and $N \cdot B_T$ are equal, TopoOpt's performance degrades faster because of the α factor in the numerator. To quantify this behavior concretely, we distribute a DLRM training task with 128 embedding tables on a cluster with 128 servers. We choose large embedding tables and distribute each table on each server, creating worst-case all-to-all traffic.

Figure 12 compares the training iteration times of TopoOpt, Ideal Switch, and Fat-tree as the batch size is increased. The top x-axis lists the ratio of all-to-all to AllReduce traffic for each batch size value given on the bottom x-axis. As shown in Figure 12a, when the batch size is 128 and $d = 4$, TopoOpt's performance matches that of Ideal Switch, while Fat-tree is a factor of 2.7 slower. This result agrees with the performance gains in Figure 11d, as the batch sizes are the same.

Increasing the batch size increases A, and this, in turn, increases the training iteration times in all three architectures. As predicted by Eq. (1), TopoOpt's iteration time increases faster. Specifically, when the batch size is 2048 and all-to-all traffic is 80% of AllReduce traffic, TopoOpt performs poorly, and the iteration time is a factor of 1.1 higher than that of the Fat-tree architecture. Increasing the server degree d mitigates the problem, as shown in Figure 12b. Note that increasing the batch size does not always result in faster training time [89, 109, 124]. Moreover, publicly available data suggest 2048 is the largest batch size for training DLRM [102]. The number of columns in the embedding tables and the number of servers are smaller in their workload: (92, 16) vs. (128, 128), respectively. Hence, the DLRM workload we evaluate contains more all-to-all traffic than the state-of-the-art model used in industry.

5.5 Impact of Host-based Forwarding

Two factors impact the performance of host-based forwarding in TopoOpt: bandwidth tax and routing strategy.

[2] For clarify of presentation, this formulation assumes no overlap between communication and computation stages and no competing traffic.

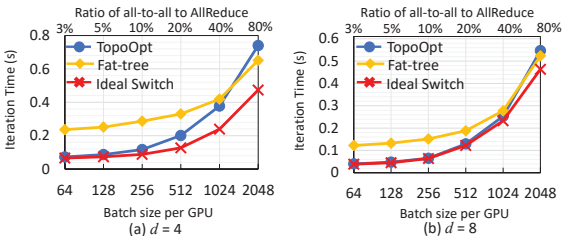


Figure 12: Impact of all-to-all traffic on a dedicated cluster of 128 servers ($B = 100$ Gbps).

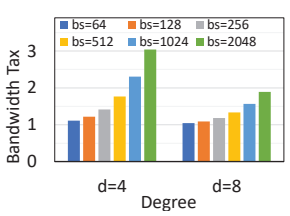


Figure 13: Bandwidth tax.

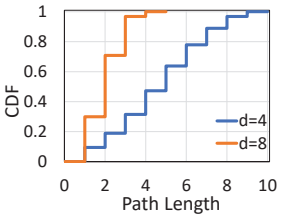


Figure 14: Path length CDF.

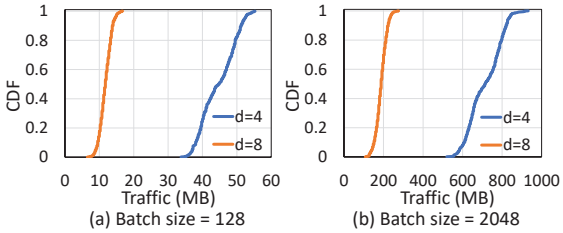


Figure 15: Traffic distribution.

Bandwidth tax. Figure 13 shows the amount of bandwidth tax experienced by the DLRM job in the previous section. Each bar represents a different batch size. At batch size 64 with $d = 4$, TopoOpt experiences a bandwidth tax of 1.11, indicating that host-based forwarding creates 11% extra traffic in the network. Increasing the degree to $d = 8$ further improves this number to 1.05. In the worst-case scenario with batch size 2048, TopoOpt pays a bandwidth tax of 3.03 when $d = 4$, causing it to perform worse than Fat-tree, as shown in Figure 12a. Determining the value of tolerable bandwidth tax is challenging for a TopoOpt cluster, as it depends on the compute time and the amount of compute-communication overlap, and this varies for different DNN models.

Impact of path length. Intuitively, the amount of bandwidth tax grows with the path length [99]. Figure 14 shows the CDF of path length across all server pairs. When $d = 4$, the average path length is 5.7, resulting in at least 5.7× overhead of host-based forwarding relative to Ideal Switch for all-to-all traffic. Based on Eq. (1), and since the total network bandwidth in TopoOpt is higher than Fat-tree ($NB_T > NB_F$), the overhead of host-based forwarding becomes at least 1.4× for the Fat-tree architecture. Increasing the server degree to 8 reduces the average path length to 3, thereby reducing the overhead bound. Appendix H evaluates the impact of increasing node degree on performance for other models.

Routing strategy. Building a topology with a small path length is necessary but not sufficient to reduce the impact of host-based forwarding. To handle forwarded traffic with minimum performance impact, the routing strategy also needs to be efficient. The best routing strategy *minimizes the maximum link utilization* for a given network topology, similar to WAN traffic engineering solutions [91]. However, finding the optimal routing strategy requires solving a set of linear equations with a centralized controller [76, 81]. To quantify the load imbalance in TopoOpt, Figure 15 illustrates the CDF of the amount of traffic carried by each physical link for an all-to-all traffic matrix. When the batch size is 128 (Figure 15a), the link with the least traffic carries 39% and 59% less traffic than the link with the most traffic, for $d = 4$ and $d = 8$, respectively. This imbalance in load suggests further opportunities to improve the performance of TopoOpt. Achieving optimal routing makes α (Eq. (1)) equal to the average path length. Without a centralized controller, however, the link utilization becomes non-uniform, and the average path length only serves as a lower bound. We leave optimizing the routing strategy in TopoOpt to future work.

5.6 Performance on Shared Clusters

We now compare the performance of different network architectures when the cluster is shared across multiple DNN jobs. Following prior work [98, 115], we run a series of simulations where 40% of the jobs are DLRM, 30% are BERT, 20% are CANDLE, and 10% are VGG16. We change the number of active jobs to represent the load on the cluster. Assuming each job requests 16 servers (64 GPUs), we execute 5, 10, 15, 20, and 27 jobs on the cluster to represent 20%, 40%, 60%, 80% and 100% load, respectively.

Figure 16 compares the average and 99%-tile iteration time at different loads for a cluster with 432 servers where $d = 8$ and $B = 100$ Gbps. SiP-ML does not support multiple jobs; hence, we omit it in this experiment. We omit OCS-reconfig and Expander networks, as they both show poor performance in this setting. Instead, we add the Oversub. Fat-tree interconnect to demonstrate the impact of congestion on Fat-tree topologies. Figure 16a shows that TopoOpt improves the average iteration time by 1.7× and 1.15×, compared to the Fat-tree and Oversub. Fat-tree architectures, respectively. We observe a similar trend for the tail iteration completion times, depicted in Figure 16b. At the extreme case when all servers are loaded, TopoOpt's tail training iteration time is 3.4× faster compared to Fat-tree architecture. Averaging across all load values on the x-axis, TopoOpt improves the tail training iteration time by 3× and 1.4× compared to Fat-tree and

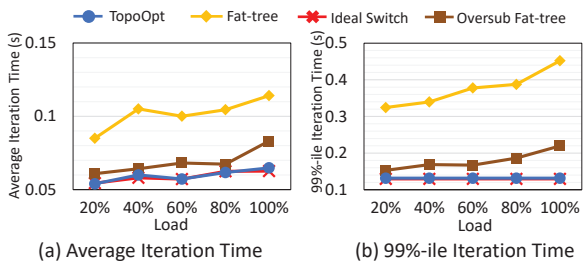


Figure 16: Shared cluster of 432 servers ($d = 8$, $B = 100$ Gbps).

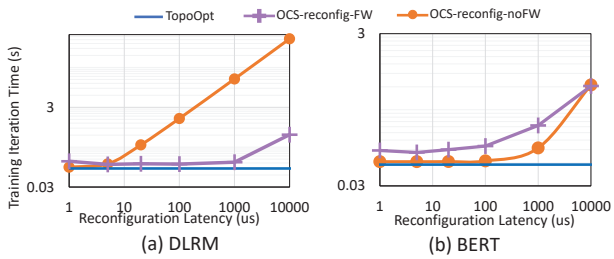


Figure 17: Impact of reconfiguration latency (d=8, B=100 Gbps).

Oversub. Fat-tree architectures.

5.7 Impact of Reconfiguration Latency

Figure 17 shows the training iteration time of DLRM and BERT in the same setting as Figure 11, while sweeping the reconfiguration latency of OCSs in OCS-reconfig from 1 μs to 10 ms. The horizontal blue line corresponds to TopoOpt's iteration time; it remains constant as it does not reconfigure the network topology. We find host-based forwarding is challenging when the network is reconfigurable, as the circuit schedules need to account for forwarding the traffic while the topology reconfigures. Therefore, we evaluate the performance of OCS-reconfig with and without host-based forwarding. The purple line corresponds to OCS-reconfig with host-based forwarding (same as OCS-reconfig evaluated in Figure 11), denoted by OCS-reconfig-FW. For the orange line, we disable host-based forwarding (similar to SiP-ML) and call it OCS-reconfig-noFW.

We find enabling host-based forwarding when the topologies reconfigures within a training iteration is not always beneficial. For DLRM (Figure 17a), OCS-reconfig-FW achieves better performance than OCS-reconfig-noFW, as DLRM has all-to-all MP transfers which benefit from host-based forwarding. However, for BERT (Figure 17b), enabling forwarding increases the chance of inaccurate demand estimation and imposes extra bandwidth tax, therefore increasing the iteration time of OCS-reconfig-FW by a factor of 1.4 compared to OCS-reconfig-noFW.

Reducing the reconfiguration latency all the way to 1 μs enables OCS-reconfig-noFW to match the performance of TopoOpt. However, OCS-reconfig-FW still suffers from inaccurate demand estimations. Although fast reconfigurable switches are not yet commercially available, they are going to be essential in elastic scenarios where the cluster is shared across multiple jobs and servers join and leave different jobs unexpectedly, or when large, high-degree communication dominates the workload. We believe futuristic fast reconfigurable switches, such as Sirius [53], are well-suited for this setting. Finding a parallelization algorithm that is aware of reconfigurability within training iterations is a challenging and exciting future research problem.

6 Prototype

Testbed setup. We build a prototype to demonstrate the feasibility of TopoOpt. Our prototype includes 12 ASUS ESC4000A-E10 servers and a G4 NMT patch panel [43]. Each server is equipped with one A100 Nvidia GPU [37] (40 GB of HBM2 memory), one 100 Gbps HP NIC [29], and one 100 Gbps Mellanox ConnectX5 NIC. Our HP NICs are capable of supporting 4×25 Gbps interfaces using a PSM4 transceiver with four breakout fibers [8], enabling us to build a TopoOpt system with degree $d = 4$ and $B = 25$ Gbps. We use RoCEv2 for communication, and enable DCB [19] and PFC on these interfaces to support a lossless fabric for RDMA. We build a completely functional TopoOpt prototype with our patch panel (Figure 18). We compare TopoOpt's performance with two baselines: (*i*) Switch 100Gbps, where the servers are connected via 100 Gbps links to a switch, and (*ii*) Switch 25Gbps, where the servers are connected via 25 Gbps links to a switch. The Switch 100Gbps baseline corresponds to the Ideal Switch case in our simulations.

Distributed training framework. We use FlexFlow's training engine [26], based on Legion's parallel programming system [30], to train four DNN models: ResNet50 [74], BERT [62], VGG16 [126], and CANDLE [4]. For DLRM, we use Facebook's implementation from [20]. Since our prototype is an order of magnitude smaller in scale than our simulation setup, we use smaller model and batch sizes.

Modifications to NCCL. By default, the NCCL communication library [36] assumes all network interfaces are routable from other interfaces. This assumption is not ideal for TopoOpt because we have a specific routing strategy to optimize training time. We modify NCCL to understand TopoOpt's topology and respect its routing preferences. Moreover, we integrate our TotientPerms AllReduce permutations into NCCL and enable it to load-balance parameter synchronization across multiple ring-AllReduce permutations.

RDMA forwarding. Implementing TopoOpt with today's RDMA NICs requires solving an engineering challenge, because the RDMA protocol assumes a switch-based network. Packet processing and memory access in RDMA protocol are offloaded to the NIC, and a RoCEv2 packet whose destination

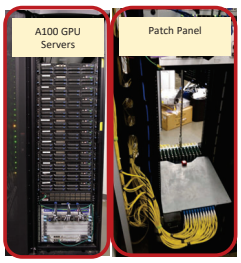


Figure 18: Testbed photo.

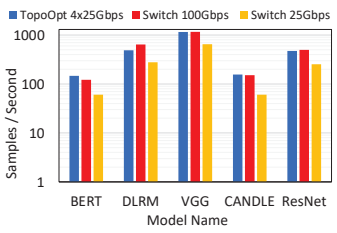


Figure 19: Training throughput (samples/second).

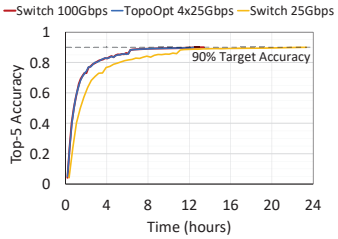


Figure 20: Time-to-accuracy of VGG19 with ImageNet.

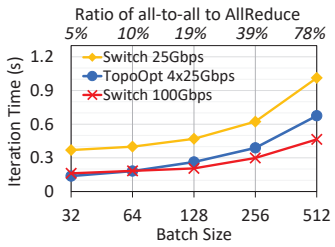


Figure 21: Impact of all-to-all traffic in our testbed.

IP address is different from that of the host is assumed to be corrupted. Therefore, the NIC silently drops forwarded packets. To address this issue, we collaborated with engineers at Marvell who developed the firmware and driver of our HP NICs. Our solution uses a feature called network partitioning (NPAR) which enables the NIC to separate host-based forwarding traffic from direct traffic, and uses the Linux kernel to route them (details in Appendix I). Our conversations with Marvell indicate that updating the firmware and the driver enables the NIC to route forwarded RoCEv2 packets, thereby bypassing the kernel entirely.

Training performance. Figure 19 demonstrates that TopoOpt's training throughput (samples/second) is similar to our Switch 100 Gbps baseline for all models. The performance of Switch 25Gbps baseline is lower because its available bandwidth is lower than TopoOpt. Figure 20 shows the time-to-accuracy plot of training VGG19 on the ImageNet [61] dataset. As the figure indicates, TopoOpt reaches the target accuracy of 90% 2.0× faster than the Switch 25Gbps baseline. TopoOpt achieves similar performance to the Switch 100Gbps baseline, as the blue and red lines overlap in Figure 20.

Impact of all-to-all traffic. Similar to Section 5.4, we evaluate the impact of all-to-all MP traffic on our RDMA-forwarding enabled testbed by measuring the average iteration time across 320 iterations of a DLRM job distributed in our testbed. We vary the amount of all-to-all traffic by changing the batch size. To create worst-case traffic, we increase the embedding dimensions by 128× relative to the state-of-the-art [20] (model details are in List 1, Appendix D). Figure 21 shows the training iteration time for various batch sizes. The results are consistent with Figure 12, but since the bandwidth tax in our 12-server testbed is much smaller than a 128-server cluster in simulations, TopoOpt performs better relative to the switch-based architectures for a given all-to-all to AllReduce traffic ratio. For instance, for batch size 512, the ratio of all-to-all traffic to AllReduce is 78%, and the training iteration time with TopoOpt is 1.6× better than the Switch 25Gbps baseline.

7 Discussion

Target workload. The most suitable workload for a TopoOpt cluster is a set of large DNN training jobs with hybrid data and model parallelism (or simply data parallelism). We assume the set of servers assigned to each job remains the same throughout the lifetime of the job, and the GPUs are not shared across multiple jobs.

Storage and control plane traffic. Meta's training clusters consist of custom-designed servers, each with eight GPUs, eight dedicated NICs for training traffic (GPU NICs), and four additional NICs for storage and other traffic (CPU NICs) [102]. Other companies, such as NVIDIA, have similar architectures [10]. TopoOpt only considers GPU NICs as server degree and partitions the network dedicated for training traffic. The CPU NICs are connected through a separate fabric to carry storage and other control plane traffic.

Supporting dynamic scheduling and elasticity. Others have demonstrated the benefits of dynamically choosing the training servers for elastic training jobs [98, 115]. Our target use case in Meta is to leverage TopoOpt for the vast number of long-lasting training jobs that do not change dynamically. In cases where elasticity is required, instead of using patch panels, we use OCSs (or other fast reconfigurable optical switches) to change the servers participating in a job quickly. Note that dynamically changing the set of servers participating in a job while keeping both the topology and the parallelization strategy optimal requires augmenting the optimization space with an additional dimension, making the problem even more challenging. We leave this to future work.

Handling failures. Unlike SiP-ML's single ring topology [89], a single link failure does not disconnect the graph in TopoOpt. When a fiber fails, TopoOpt can temporarily use a link dedicated to MP traffic to recover an AllReduce ring. In case of permanent failures, TopoOpt reconfigures to swap ports and recover the failed connection.

Supporting multi-tenancy. To support multi-tenancy [142, 143], TopoOpt can leverage NVIDIA's MIG [39] to treat one physical server as multiple logical servers in its topology.

TotientPerms in Fat-trees. Although our TotientPerms technique is well-suited for reconfigurable optical interconnects, it may be of independent interest for Fat-tree intercon-

nects as well since load-balancing the AllReduce traffic across multiple permutations can help with network congestion.

TopoOpt's limitations. TopoOpt's approach assumes the traffic pattern does not change between iterations. However, this assumption may not hold for Graphic Neural Network (GNN) models [121] or Mixture-of-Expert (MoE) models [80]. In addition, we plan to extend TopoOpt by bringing its demand-awareness design within training iterations. This is an open research question, and as shown in Section 5.7, we will need fast-reconfigurable optical switches, as well as a more sophisticated scheduling algorithm. Another limitation of TopoOpt is that a single link failure within a AllReduce ring causes the full ring to become inefficient for AllReduce traffic. A fast optical switch addresses this problem by quickly reconfiguring the topology.

8 Related Work

Optimizing DNN training. To address the increasing computation and network bandwidth requirements of large training jobs, a plethora of frameworks have been proposed [5, 46, 58, 69, 77, 79, 85, 86, 105, 108, 111, 117, 118, 123, 129, 136, 146]. These frameworks distribute the dataset and/or DNN model across accelerators while considering the available network bandwidth, but unlike TopoOpt, they do not consider optimizing the *physical layer topology*. Specifically, Blink [136] builds collectives for distributed ML, but it needs a physical topology to generate spanning trees. Zhao et al. [147] study the optimal topology for collective communication operations, but this does not apply for general MP traffic. In addition, several methods have been proposed to quantize and compress the gradients to reduce the amount of communication data across servers [48, 56, 144]. While these approaches are effective, they are designed for data parallel strategies and do not consider the large amount of data transfers caused by model parallel training. Wang et al. [138] compare the performance of Fat-trees and BCube topologies for distributed training workloads and highlight several inefficiencies in Fat-trees. SiP-ML [89] demonstrates the benefits of 8 Tbps silicon photonics-based networks for distributed training. However, unlike TopoOpt, these proposed approaches do not *co-optimize* topology and parallelization strategy.

DNN parallelization strategies. Data and model parallelism are widely used by today's DNN frameworks (e.g., TensorFlow [44], PyTorch [42], MXNet [17]) to parallelize training across multiple devices. Recent work has also proposed *automated frameworks* (e.g., FlexFlow [85], ColocRL [101], MERLIN [38]) that find efficient parallelization strategies by searching over a comprehensive space of potential strategies. These frameworks rely on and are optimized for the conventional Fat-tree interconnects. TopoOpt proposes a new approach to building DNN training systems by jointly optimizing network topology and parallelization strategy.

DNN training infrastructures and schedulers. Several training infrastructures have been proposed recently, including NVIDIA DGX SuperPOD [10], TPU cluster [9], and supercomputers [1]. All these systems assume non-reconfigurable network topologies, such as Fat-tree, Torus, and other traffic-oblivious interconnects. TopoOpt is the first DNN system to use commodity reconfigurable interconnects to accelerate DNN jobs.Gandiva [140], Themis [98], Tiresias [70], BytePS [86, 111], and Pollux [115] seek to improve the utilization of GPU clusters through scheduling algorithms. These approaches are complementary to ours, and many of their techniques can be applied to a TopoOpt cluster.

Optical Interconnects. Several papers have demonstrated the benefits of optically reconfigurable interconnects for datacenters [51, 53, 57, 60, 64, 68, 95–97, 99, 100, 113]. These designs lead to sub-optimal topologies for distributed DNN traffic. Similarly, *traffic oblivious* interconnects, such as RotorNet [99, 100], are a great fit for datacenter workloads, but they are not suitable for DNN training jobs characterized by repetitive traffic demands. Hybrid electrical/optical datacenter proposals [64, 137] can be used to route AllReduce traffic through the optical fabric and MP flows through a standard electrical Fat-tree network. But hybrid clusters are not cost effective and suffer from many problems, including TCP ramp-up inefficiencies [103], segregated routing issues [65], and uncertainty in terms of how to divide the cluster between electrical and optical fabrics [68, 72].

9 Conclusion

We present TopoOpt, a novel system based on optical devices that jointly optimizes DNN parallelization strategy and topology to accelerate training jobs. We design an alternating optimization algorithm to explore the large space of *Computation* $\times$ *Communication* $\times$ *Topology* strategies for a DNN workload, and demonstrate TopoOpt obtains up to 3.4$\times$ faster training iteration time than Fat-tree.

10 Acknowledgments

We thank our shepherd Sangeetha Abdu Jyothi and anonymous reviewers for their valuable feedback. We also acknowledge Meta for supporting this research. In particular, we thank Omar Baldonado, Gaya Pradeep Sindhu, and Jahangir Hasan. In addition, we thank Alan Gibbemeyer, Bob Shine, Karl Kuhn and Ramiro Voicu from Telescent for their support on the Telescent NTM-G4. We also thank Arial Elior, Karl Erickson, and Nishant Lodha from Marvell for their help on RDMA forwarding. The MIT-affiliated authors are supported by ARPA-E ENLITENED PINE DE-AR0000843, DARPA FastNICs 4202290027, NSF grants CNS-2008624, SHF-2107244, ASCENT-2023468, CAREER-2144766, PPoSS-2217099, CNS-2211382, Meta faculty award, Google faculty award, and Sloan fellowship FG-2022-18504.

References

[1] Summit Supercomputer, 2014. https://www.olcf.ornl.gov/summit/.

[2] Datasheet for Single Mode Network Optical Switch up to 384x384 ports, 2016. https://www.hubersuhner.com/en/documents-repository/technologies/pdf/data-sheets-optical-switches/polatis-series-7000n.

[3] Baidu, 2017. https://github.com/baidu-research/baidu-allreduce.

[4] CANDLE Uno: Predicting Tumor Dose Response across Multiple Data Sources, 2017. https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno.

[5] Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow, 2017. https://eng.uber.com/horovod.

[6] CALIENT Edge 640™ Optical Circuit Switch, 2018. https://www.calient.net/2018/03/calient-edge640-optical-circuit-switch-offers-industrys-highest-density-fiber-optic-cross-connect/.

[7] htsim packet simulator, 2018. https://github.com/nets-cs-pub-ro/NDP/wiki/NDP-Simulator.

[8] AOI 100G PSM4 Transceiver, 2020. https://www.ebay.com/itm/234092018446?hash=item3680f8bb0e:g:WoMAAOSwLFJg8dKF.

[9] Google TPU, 2020. https://cloud.google.com/tpu.

[10] Nvidia DGX SuperPOD, 2020. https://www.nvidia.com/en-us/data-center/dgx-superpod/.

[11] NVIDIA is Preparing Co-Packaged Photonics for NVLink, Dec. 2020. https://www.techpowerup.com/276139/nvidia-is-preparing-co-packaged-photonics-for-nvlink.

[12] 100GBASE-SR4 QSFP28 850nm 100m DOM MTP/MPO MMF Optical Transceiver Module, 2022. https://www.fs.com/products/48354.html.

[13] 10GBASE-SR SFP+ 850nm 300m DOM LC MMF Transceiver Module, 2022. https://www.fs.com/products/11552.html.

[14] 1x2 PLC Fiber Splitter, Splice/Pigtailed ABS Module, 2.0mm, SC/APC, Singlemode, 2022. https://www.fs.com/products/11615.html.

[15] 25GBASE-SR SFP28 850nm 100m DOM LC MMF Optical Transceiver Module, 2022. https://www.fs.com/products/67991.html.

[16] 40GBASE-SR4 QSFP+ 850nm 150m DOM MTP/MPO MMF Optical Transceiver Module, 2022. https://www.fs.com/products/36143.html.

[17] Apache MXNet, 2022. https://mxnet.apache.org/.

[18] Colfax Direct, HPC and Date Center Gear, 2022. https://www.colfaxdirect.com/.

[19] Data Center Bridging eXchange (DCBX), 2022. https://man7.org/linux/man-pages/man8/dcb-dcbx.8.html.

[20] Deep Learning Recommendation Model for Personalization and Recommendation Systems, 2022. https://github.com/facebookresearch/dlrm.

[21] Edgecore AS5812-54X 48-Port 10GbE Bare Metal Switch with ONIE - Part ID: 5812-54X-O-12V-F, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3614.

[22] Edgecore AS6812-32X 32-Port 40GbE Bare Metal Switch with ONIE - Part ID: 6812-32X-O-AC-F-US, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3078.

[23] Edgecore AS7312-54XS 48-Port 25GbE + 6-Port 100GbE Bare Metal Switch with ONIE - Part ID: 7312-54XS-O-AC-F-US, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3598.

[24] Edgecore AS7816-64X 64-Port 100GbE Bare Metal Switch with ONIE - Part ID: 7816-64X-O-AC-B-US, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3483.

[25] Euler's totient function, 2022. https://en.wikipedia.org/wiki/Euler%27s_totient_function.

[26] Flex Flow's Training Engine, 2022. https://flexflow.ai/.

[27] FlexFlow source code, 2022. https://github.com/flexflow/FlexFlow.

[28] FS.COM, 2022. https://www.fs.com/.

[29] HPE Ethernet 4x25Gb 1-port 620QSFP28 Adapter, 2022. https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c05220334.

[30] Legion Programming System, 2022. https://legion.stanford.edu/overview/.

[31] Managing edge data centers through automation and remote diagnostics, 2022. https://www.telescent.com/blog/2021/11/11/managing-edge-data-centers-through-automation-and-remote-diagnostics.

[32] Mellanox ConnectX-4 Single Port 25 Gigabit Ethernet Adapter Card, PCIe 3.0 x8 - Part ID: MCX4111A-ACAT, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2814.

[33] Mellanox ConnectX-4 Single Port 40 Gigabit Ethernet Adapter Card, PCIe 3.0 x8 - Part ID: MCX4131A-BCAT, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2817.

[34] Mellanox ConnectX-5 EN Single Port 100 Gigabit Ethernet Adapter Card, PCIe 3.0 x16 - Part ID: MCX515A-CCAT, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3150.

[35] Mellanox ConnectX-5 VPI Adapter Card with Multi-Host Socket Direct, Dual PCIe 3.0 x8 - Part ID: MCX556M-ECAT-S25, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3209.

[36] NCCL, 2022. https://github.com/NVIDIA/nccl-tests.

[37] NVIDIA A100 Tensor Core GPU, 2022. https://www.nvidia.com/en-us/data-center/a100/.

[38] NVIDIA MERLIN, 2022. https://developer.nvidia.com/nvidia-merlin.

[39] NVIDIA MULTI-INSTANCE GPU, 2022. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[40] Patch Panel Wiki, 2022. https://en.wikipedia.org/wiki/Patch_panel.

[41] Polatis Optical Circuit Switch, 2022. https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabled.asp.

[42] PyTorch, 2022. https://pytorch.org.

[43] Telescent G4 Network Topology Manager, 2022. https://www.telescent.com/products.

[44] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[45] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale, 2020.

[46] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Learning generalizable device placement algorithms for distributed machine learning. In *Advances in Neural Information Processing Systems*, volume 32, pages 3981–3991. Curran Associates, Inc., 2019.

[47] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[48] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 1709–1720. Curran Associates, Inc., 2017.

[49] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[50] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

[51] Daniel Amir, Tegan Wilson, Vishal Shrivastav, Hakim Weatherspoon, Robert Kleinberg, and Rachit Agarwal. Optimal oblivious reconfigurable networks, 2021.

[52] Javed A. Aslam. Dynamic Programming Solution to the Coin Changing Problem, 2004. https://www.ccs.neu.edu/home/jaa/CSG713.04F/Information/Handouts/dyn_prog.pdf.

[53] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[54] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[55] J. Brownlee. *Better Deep Learning: Train Faster, Reduce Overfitting, and Make Better Predictions*. Machine Learning Mastery, 2018.

[56] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp : Adaptive residual gradient compression for data-parallel distributed training. *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[57] Li Chen, Kai Chen, Zhonghua Zhu, Minlan Yu, George Porter, Chunming Qiao, and Shan Zhong. Enabling wide-spread communications on optical fabric with megaswitch. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 577–593, Boston, MA, 2017. USENIX Association.

[58] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. SysML Conference, 2019.

[59] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, and Michael Haselman. Accelerating persistent neural networks at datacenter scale. In *Hot Chips*, volume 29, 2017.

[60] K. Clark, H. Ballani, P. Bayvel, D. Cletheroe, T. Gerard, I. Haller, K. Jozwik, K. Shi, B. Thomsen, P. Watts, H. Williams, G. Zervas, P. Costa, and Z. Liu. Sub-nanosecond clock and data recovery in an optically-switched data centre network. In *2018 European Conference on Optical Communication (ECOC)*, pages 1–3, 2018.

[61] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[62] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[63] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[64] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. *SIGCOMM'10*, pages 339–350.

[65] K. Foerster, M. Ghobadi, and S. Schmid. Characterizing the algorithmic complexity of reconfigurable data center architectures. In *Proc. ANCS '18*, pages 89–96, 2018.

[66] Everest G. and Ward Thomas. An introduction to number theory, 2005.

[67] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, Berkeley, CA, USA, 2016. USENIX Association.

[68] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 216–229, New York, NY, USA, 2016. Association for Computing Machinery.

[69] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[70] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[71] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 63–74, New York, NY, USA, 2009. Association for Computing Machinery.

[72] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. *SIGCOMM'14*, pages 319–330.

[73] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 418–430, 2019.

[74] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[75] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering, 2017.

[76] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[77] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS*, 2019.

[78] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[79] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR*, abs/1511.00175, 2015.

[80] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[81] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[82] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed DNN training. *CoRR*, abs/1905.03960, 2019.

[83] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *CoRR*, abs/1807.11205, 2018.

[84] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2274–2283, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[85] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *SysML*, 2019.

[86] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[87] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. Optimizing bulk transfers with software-defined optical wan. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 87–100, New York, NY, USA, 2016. Association for Computing Machinery.

[88] A. S. Kewitsch. Large scale, all-fiber optical cross-connect switches for automated patch-panels. *Journal of Lightwave Technology*, 27(15):3107–3115, 2009.

[89] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[90] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[91] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering with smore. In *Proceedings of the Applied Networking Research Workshop*, ANRW '18, page 21, New York, NY, USA, 2018. Association for Computing Machinery.

[92] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On model parallelization and scheduling strategies for distributed machine learning. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 2834–2842. Curran Associates, Inc., 2014.

[93] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. OSDI'14, pages 583–598. USENIX Association, 2014.

[94] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[95] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with REACToR. *NSDI'14*, pages 1–15.

[96] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[97] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: A new design element for low-latency dcns. *SIGCOMM'14*, pages 283–294.

[98] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, February 2020. USENIX Association.

[99] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. *NSDI'20*, 2020.

[100] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. *SIGCOMM '17*, pages 267–280, 2017.

[101] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2430–2439, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[102] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021.

[103] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting TCP for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, Santa Clara, CA, February 2020. USENIX Association.

[104] Shar Narasimhan. NVIDIA Clocks World's Fastest BERT Training Time and Largest Transformer Based Model, Paving Path For Advanced Conversational AI, Aug. 2019. https://devblogs.nvidia.com/training-bert-with-gpus/.

[105] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[106] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. Deep learning training in facebook data centers: Design of scale-up and scale-out systems, 2020.

[107] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[108] T. T. Nguyen, M. Wahib, and R. Takano. Topology-aware sparse allreduce for large-scale deep learning. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2019.

[109] Yosuke Oyama, Naoya Maruyama, Nikoli Dryden, Erin McCarthy, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Peter Nugent, and Brian Van Essen. The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[110] Heng Pan, Zhenyu Li, JianBo Dong, Zheng Cao, Tao Lan, Di Zhang, Gareth Tyson, and Gaogang Xie. Dissecting the communication latency in distributed deep sparse learning. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 528–534, New York, NY, USA, 2020. Association for Computing Machinery.

[111] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[112] Genzhi Photonics. 1x2 Mechanical Optical Switch, 2022. https://www.gezhiphotonics.com/1x2-optical-switch.html.

[113] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. *SIGCOMM'13*, pages 447–458.

[114] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of ACM SIGCOMM 2022*, 2022.

[115] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[116] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[117] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022.

[118] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021.

[119] Leslie Reid. MOX Announces New Telescent Automation Technology on Its Latest Hillsboro to Portland Fiber Route, Sept. 2020. https://www.businesswire.com/news/home/20200915005391/en/MOX-Announces-New-Telescent-Automation-Technology-on-Its-Latest-Hillsboro-to-Portland-Fiber-Route.

[120] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[121] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[122] Tae Joon Seok, Niels Quack, Sangyoon Han, Richard S. Muller, and Ming C. Wu. Large-scale broadband digital silicon photonic switches with vertical adiabatic couplers. *Optica*, 3(1):64–70, Jan 2016.

[123] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[124] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

[125] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[126] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[127] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 17–17, Berkeley, CA, USA, 2012. USENIX Association.

[128] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[129] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of DNN graph operators. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[130] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[131] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[132] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[133] Yuichiro Ueno and Rio Yokota. Exhaustive study of hierarchical allreduce patterns for large messages between gpus. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 430–439, 2019.

[134] Jakob Uszkorei. Transformer: A Novel Neural Network Architecture for Language Understanding, Aug. 2017. https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html.

[135] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 205–219, New York, NY, USA, 2016. ACM.

[136] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *Conference on Machine Learning and Systems (MLSys 2020)*, March 2020.

[137] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time optics in data centers. *SIGCOMM'10*, pages 327–338.

[138] S. Wang, D. Li, J. Geng, Y. Gu, and Y. Cheng. Impact of Network Topology on the Performance of DML: Theoretical Analysis and Practical Factors. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1729–1737, 2019.

[139] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. *SoCC '16*, 2016.

[140] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[141] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. Lighter-communication distributed machine learning via sufficient factor broadcasting. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*, pages 795–804, Arlington, Virginia, USA, 2016. AUAI Press.

[142] Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[143] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. Fluid: Resource-aware hyperparameter tuning engine. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 502–516, 2021.

[144] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, volume 32, pages 4438–4449. Curran Associates, Inc., 2019.

[145] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 235–254, Boston, MA, February 2019. USENIX Association.

[146] H. Zhao and J. Canny. Kylix: A sparse allreduce for commodity clusters. In *2014 43rd International Conference on Parallel Processing*, pages 273–282, 2014.

[147] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Optimal direct-connect topologies for collective communications, 2022.

[148] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A case for redundant arrays of inexpensive links in data center networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 561–576, Boston, MA, March 2017. USENIX Association.

A Tree-AllReduce and Other AllReduce Permutations

Section 2 established that we can manipulate the traffic of a ring-AllReduce collective by permuting the labeling of servers in the AllReduce group. Here, we illustrate how to use the same technique on another AllReduce algorithm, called tree-AllReduce.

In the tree-AllReduce algorithm, the servers are connected logically to form a tree topology. The AllReduce operation starts by running a reduce operation to the root node with recursive halving, followed by a broadcast to the rest of the cluster with recursive doubling [132].

A common instantiation of tree-AllReduce is the double binary tree (DBT) algorithm [120]. In this algorithm, the first step is to create a balanced binary tree for the nodes. The properties of balanced binary trees guarantee that one half of the nodes will be leaf-nodes, and the other half will be in-tree; thus, a second binary tree is constructed by flipping the labeling of the leaf and in-tree nodes. This way, each node (except the root in both trees) has the same communication requirements for the AllReduce operation, as described in the last paragraph, and bandwidth-optimally is achieved. Figure 23a shows an example where in the first binary tree, the in-tree nodes are even, and the leaf nodes are odd, while the second tree flips the labeling.

The DBT itself is essentially an example of permuting the node labeling to achieve an AllReduce operation with balanced communication load. We also note that we can permute the labeling *for the entire set of nodes* for a pair of DBTs to create a new pair of trees that can perform the AllReduce operation at the same speed. Figures 23b and 23c illustrate two other possible double binary trees, and their corresponding traffic demand matrix for the DLRM and CANDLE example shown in Figures 22 and 24 (§2). Arbitrary permutations can be used, and to limit the cases, we could simply consider the cyclic permutations in the modular space as described in TotientPerms.

In general, all AllReduce operations can be described as a directed graph $G = (V, E)$ where V is the set of nodes in the cluster, and E denotes data dependencies. The *permutable* property says every graph $G' = (V, E')$ that is isomorphic to G can perform the AllReduce operation equally well, where the homomorphism between G and G' is described by the symmetric group on V (generally denoted by $Sym(V)$ in group theory).

B Commercially Available Patch Panels and Optical Circuit Switches

Optical patch panels. A patch panel is a device to facilitate connecting different parts of a system. For instance, electrical patch panels are used in recording studios and concert sound

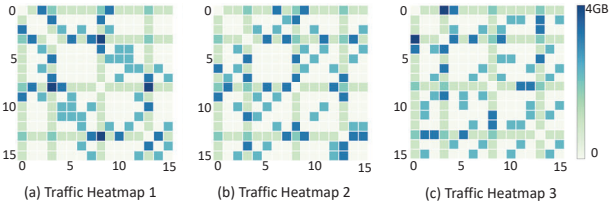


Figure 22: DLRM traffic heatmaps with double binary tree AllReduce.

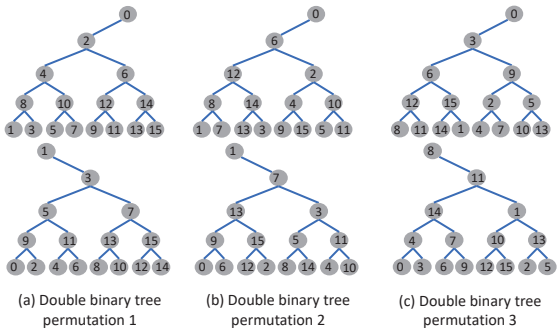


Figure 23: Double binary tree (DBT) permutations.

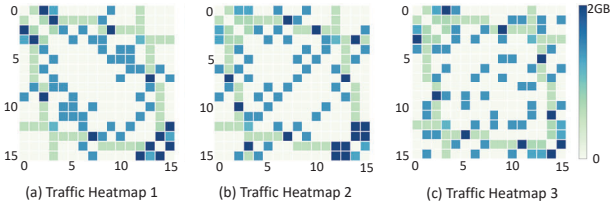


Figure 24: CANDLE traffic heatmaps with double binary tree AllReduce.

systems to connect microphones and electronic instruments on demand [40]. Fiber optic patch panels are commonly used for cable management, and have been proposed in recent datacenter topology designs [145]. Reconfigurable optical patch panels are a new class of software-controlled patch panels and are already commercialized at scale [119]. For instance, Telescent offers 1008 duplex ports with insertion loss less than 0.5 dB and cost ≈$100K ($100/port) [88, 119]. Reconfiguration is performed using a robotic arm that grabs a fiber on the transmit side and connects it to a fiber on the receive side [88]. However, the reconfiguration latency of optical patch panels is several minutes [43]. Note that reliability is of utmost concern for operation in unmanned locations; for example, Telescent NTM patch panels have been certified to NEBS Level 3 and have over 1 billion port hours in operation [31].

3D MEMS-based Optical Circuit Switches (OCSs). An OCS uses tiny mirrors to change the direction of light, thereby

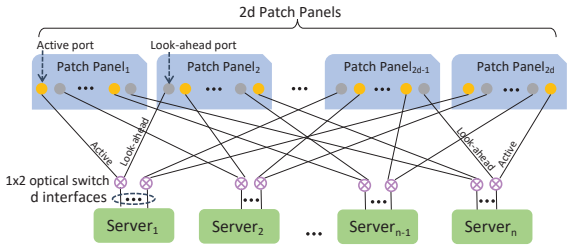


Figure 25: Active & Look-ahead ports for high reconfiguration latency.

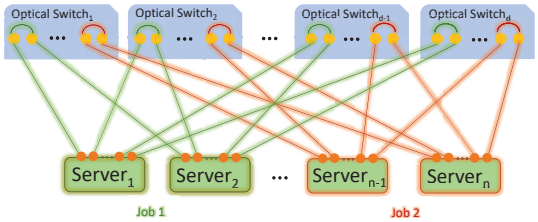


Figure 26: Sharding TopoOpt cluster for two jobs.

reconfiguring optical links. The largest optical circuit switch on the market has 384 duplex ports with ≈10 ms reconfiguration latency and is available for $200K ($520/port) [41]. However, the optical loss of these switches is 1.5–2.7 dB [2]. Compared to patch panels, OCSs have the following disadvantages: (*i*) each port is five times more expensive; (*ii*) their insertion loss is higher; and (*iii*) their port-count is three times lower. The main advantage of OCSs is that their reconfiguration latency is *four orders of magnitude* faster than patch panels.

C Handling Sharding and Dynamic Job Arrivals in Shared Clusters

Section 3 explained how TopoOpt can support multiple job sharing the cluster through sharding; here we provide a detailed explanation of how sharding works. Figure 26 shows how a TopoOpt cluster is sharded to train two jobs together. In this scenario, the optical switches are configured in a way such that the green part (Server 1, 2 and their corresponding links) is completely disjointed from the red part (Server $n-1$, server n). The complete isolation ensures each job gets its dedicated resources, and benefits the performance (especially the tail latency) as shown in Section 5.6.

To start a job with k servers, we need to reconfigure the interconnection between these k servers before the job starts. This can be done quickly when OCSs are used, but when patch panels are used, there could be several minutes of delay before the job can start. To address this challenge, we use a look-ahead approach to pre-provision the next topology while current jobs are running. More specifically, we use a simple 1×2 mechanical optical switch [112] at each server's interface to choose between *Active* and *Look-ahead* ports. These 1×2 switches are inexpensive ($25) and have 0.73 dB optical loss measured in our prototype. Unlike optical splitters [14], that incur 3 dB loss, these switches choose where to send light between their two output ports. We then connect the two ends of each 1×2 switch to different patch panels, as shown in Figure 25. As a result, a TopoOpt cluster with n servers, each with d interfaces, has $2d$ patch panels where each interface is split into two parts: Active and Look-ahead. At any point in time, only one end of each 1×2 switch is participating in the active topology; the other end is pre-provisioning the topology for the next job. Since the topology and parallelization strategy are calculated off-line, we already know the sequence of job arrivals and the number of servers required by each job. This design allows each server to participate in two independent topologies. Hence, when a set of servers uses one topology for a training job, TopoOpt pre-provisions the next topology, optimized for the next task by reconfiguring Look-ahead ports. Once all the servers for the new job are ready, TopoOpt immediately *flips* to the new topology by reconfiguring the corresponding 1×2 switches.

D Model Configurations and Transfer Sizes

List 1 summarizes the parameters we used in our simulation and testbed. Model parameters and batch sizes are selected based on common values used in Meta for simulations. For the prototype, we reduce parameter values and batch sizes to fit the models in our 12-node cluster.

In most workloads observed in Meta, the size of AllReduce transfers is larger than the size of MP transfers for each iteration, because in most cases, it would not be worthwhile if MP transfers were as large as AllReduce transfers. Consider the DLRM example in Section 4.3 with 20 GB embedding tables with double-precision floating parameters. If we were to distribute this embedding table using data parallelism, each server would need to send and receive 37.5 GB of data for the AllReduce operation. On a 100 Gbps fabric, this would take 3 seconds by itself, but if we put it on one server, it would only need to transfer 32 MB/server (assume we have a per-server batch size of 8192; then, MP traffic is calculated as 16 servers $\times$ 8192 samples/server $\times$ 512 activation per sample $\times$ 8 bytes per activation / 16 servers = 32 MB). We note that adding *pipeline parallelism* can increase the amount of MP traffic as it overlaps forward and backward passes. Efficient ways to pipeline batches remains an active research area [77, 105] especially when hybrid parallelism is employed. Pure model parallelism creates another type of sparse traffic pattern where only accelerators with inter-layer dependencies need to communicate. Our TopologyFinder algorithm can support such communication patterns.

Conceptually, however, when the network bandwidth goes

VGG:
Batch/GPU: 64 (§5.3, §5.6), 32 (§6)
ResNet50:
Batch/GPU: 128(§5.3), 20 (§6)
BERT:
Batch/GPU: 16 (§5.3, §5.6), 2 (§6)
#Trans. blks: 12 (§5.3), 6 (§5.6, §6)
Hidden layer: 1024 (§5.3), 768 (§5.6), 1024(§6)
Seq. length: 64 (§5.3), 256 (§5.6), 1024(§6)
#Attn. heads: 16 (§5.3), 6 (§5.6), 16(§6)
Embed. size: 512 (§5.3, §5.6, §6)
DLRM:
Batch/GPU: 128 (§5.3),[32,$\cdots$,2048] (§5.4), 256 (§5.6), [64,$\cdots$,512] (§6)
#Dense layer: 8 (§5.3, §5.6), 4 (§6)
Dense layer size: 2048 (§5.3), 1024 (§5.6, §6)
#Dense feat. layer: 16 (§5.3, §5.6), 8 (§6)
Feat. layer size: 4096 (§5.3), 2048 (§5.6, §6)
Embed.: 128×10^7 (§5.3), 256×10^7 (§5.6), 32768×10^5 (§6)
#Embed. tables: 64 (§5.3), 16 (§5.6), 128 (§5.4) , 12 (§6)
CANDLE:
Batch/GPU: 256 (§5.3, §5.6), 10 (§6)
#Dense layer: 8 (§5.3, §5.6), 4 (§6)
Dense layer size: 16384 (§5.3), 4096 (§5.6, §6)
#Dense feat. layer: 16 (§5.3, §5.6), 8 (§6)
Feat. layer size: 16384 (§5.3), 4096 (§5.6, §6)
NCF:
Batch/GPU: 128 (§5.3)
#Dense layer: 8 (§5.3)
Dense layer size: 4096 (§5.3)
#User embedding table (MF, MLP): 32, 32 (§5.3)
#User per table: 10^6 (§5.3)
#Item embedding table (MF, MLP): 32, 32 (§5.3)
#Item per table: 10^6 (§5.3)
MF embedding dimension: 64 (§5.3)
MLP embedding dimension: 128 (§5.3)

List 1: DNN models used in our simulations and testbed.

to infinity, other overheads in the system (e.g. CUDA kernel launch) will dominate the latency. In such cases, it might be beneficial to choose model parallelism instead of data parallelism, to reduce the amount of system overheads. In particular, prior work shows 10 Tbps Silicon Photonics links enable more aggressive model parallelism where the size of MP traffic is significant [89]. TopoOpt's approach to distribute the degree between the MP and AllReduce sub-topologies enables us to accommodate this case as well.

E Algorithm Details

E.1 TopologyFinder

Using group theory to find AllReduce permutations. For a ring-AllReduce group with n servers labeled $S_0,...,S_{n-1}$, a straightforward permutation is $(S_0 \to S_1 \to S_2 \cdots \to S_{n-1} \to S_0)$. We denote this permutation by a ring generation rule as: $S_i \to S_{(i+1) \ mod \ n}$. Since the servers form a ring, the index of the starting server does not matter. For instance, these two rings are equivalent: $(S_0 \to S_1 \to S_2 \to S_3 \to S_0)$ and $(S_1 \to S_2 \to S_3 \to S_0 \to S_1)$.[3]

We first provide the mathematical foundation of the ring permutation rule.

Theorem 2 (Ring Generation)**.** *For a cluster of n nodes $V = \{S_0, S_1, \cdots, S_{N-1}\}$, all integer numbers $p < n$, where p is co-prime with n (i.e. $gcd(p,n) = 1$) represent a unique ring-AllReduce permutation rule.*

Proof. Consider the integer modulo n group with addition $\mathbb{Z}_n^+ = \{0, 1, \cdots, (n-1)\}$. $\mathbb{Z}_n^+$ is a cyclic group. By the fundamental theorem of cyclic groups, p is a generator of $\mathbb{Z}_n^+$ if and only if $gcd(p,n) = 1$. Hence we can cover the entire $\mathbb{Z}_n^+$ by repeatedly adding p to itself.

Now consider the graph $G_{\mathbb{Z}_n^+,p} = (V_{\mathbb{Z}_n^+}, E_p)$ where the set of vertices $V_{\mathbb{Z}_n^+} = \mathbb{Z}_n^+$ and $E_p = \{(a \times p, (a+1) \times p) \in V_{\mathbb{Z}_n^+}^2, \ a \in \mathbb{Z}_n^+\}$. The set E_p forms a cycle on $G_{\mathbb{Z}_n^+,p}$. Now denote our cluster as $G = (V,E)$ where V is defined as above and E represents a set of directed links. Then $G_{\mathbb{Z}_n^+,p}$ is isomorphic to G, hence following the rule in E_p we can define a valid ring in G. Furthermore, since $\forall p_i \neq p_j$ we can guarantee that $(0, p_i) \in E_{p_i}$ and $(0, p_j) \notin E_{p_i}$, and each p_i is guaranteed to describe a unique ring. □

One way to extend our approach to other AllReduce algorithms is to generalize TotientPerms (Algorithm 2) so that the E_p described in theorem 2 simply represents a *permutation* which we apply to the original node labeling, while keeping the edge relation, to create an isomorphic graph that describes the new AllReduce topology.

E.2 Bounding maximum hop count with TotientPerms

In this section, we argue that fitting a geometric sequence for choosing permutation provides an approximately $O(d\sqrt[d]{n})$ bound for the maximum diameter of a cluster with n nodes and degree d. Denote $x \equiv \sqrt[d]{n}$. We simplify the question to the following: given a contiguous set of numbers $\mathcal{N} = \{1,\ldots,n\}$ and a set of numbers from the geometric sequence $S = \{x^0, x^1, \ldots, x^{d-1}\}$, choose h numbers (allow repetition) $s_1, \cdot, s_k$ from S so that $m = \sum_{i=1}^{h} s_i$ for some $m \in \mathcal{N}$. Let $h = \kappa(m)$, find $\min_{m \in \mathcal{N}} \kappa(m)$.

Again for simplicity, assume $x \in \mathbb{Z}$. Then for a given $m \in \mathcal{N}$, we get the recursive relation $\kappa(m) = 1 + \kappa(m - x^i)$ where $i = \text{argmax}_{i \leq d, x^i \leq m}$. $m = N-1$ gives the maximum $\kappa(N-1) = dx$.

The problem above is simpler than the one in TopoOpt. In TopoOpt, x is rarely an integer, and S is a projection of the geometric sequence $S = \{x^0, x^1, \ldots, x^{d-1}\}$ onto the

[3] Given that ring-AllReduce is the dominant AllReduce collective, we describe our algorithms based on ring-AllReduce. Appendix E.1 explains how to extend our algorithm to other AllReduce communication collectives.

Algorithm 4 `CoinChangeMod` pseudocode

```
1:  procedure CoinChangeMod(n, G)
      ▷ Input n: Total number of nodes
      ▷ Input G: Network Topology
      ▷ Output R: Routings
        ▷ R is the routing result
2:    R = {}
      ▷ Acquire the set of "coins" from the topology,
      ▷ which are the choices of Algorithm 3
3:    C = GetCoins(G)
4:    for i ∈ [1, N−1] do
        ▷ curr_dist denotes the "distance" of a value
        ▷ (node distance) counted by number of "coins"
5:      curr_dist[i] = ∞
        ▷ curr_bt record a back-trace of "coins" to
        ▷ get to a value (node distance)
6:      curr_bt[i] = ∞
7:    for c ∈ C do
8:      curr_dist[c] = 0
9:      curr_bt[c] = c
10:   while curr_dist has at least one ∞ in it do
11:     for i ∈ [1, N−1] do
12:       new_dist[i] = curr_dist[i]
13:       new_bt[i] = curr_bt[i]
14:       for c ∈ C do
15:         if curr_dist[(i−c) mod N] < new_dist[i] then
16:           new_dist[i] = cur_dist[(i−c) mod N] + 1
17:           new_bt[i] = c
18:     curr_dist = new_dist
19:     curr_bt = new_bt
        ▷ Construct the routing for each node distance from the back-trace
20:   R = GetRouteSeq(curr_bt)
21:   return R
```

candidates (co-prime numbers with the size of a subset of node participating in AllReduce). The intuition still holds.

Note that when $\sqrt[d]{n} < 2$, it is advantageous to choose $x = 2$ and spend less degree to create a geometric sequence with a ratio of at least 2. In this case, the d factor becomes the actually used degree $d = \log_2 n$, and the bound holds at $O(\log_2 n)$.

E.3 Coin Change Routing

Consider servers S_i and S_j that need to exchange AllReduce transfers but do not have a direct edge between them. We use a modified version of the classical coin change problem [52] to find an efficient routing path (line 19). In classical coin change, the goal is to find the minimum number of *coins* that would sum to a certain *total value*. Our ring generation rules enable us to treat the routing problem similarly. In particular, the p values of AllReduce permutations that have been selected in the AllReduce sub-topology are the coin values, and the difference between server i and j indices $((j-i) \ mod \ n)$ is the target total value that we want to achieve. The only difference is that our problem runs with *modulo n* arithmetic, as the server IDs wrap around in the ring structure. Algorithm 4 lists the pseudocode of `CoinChangeMod`.

E.4 OCS-reconfig Heuristic

Algorithm 5 describes the heuristic we use for OCS-reconfig. As mentioned in Section 4, our goals are (*i*) to have enough bandwidth for large transfer demands, (*ii*) while also minimizing the latency of indirect routing for nodes that do not have a direct link between them.

To achieve this goal in a reconfigurable interconnect, we propose a utility function that finds a balance between the two goals by maximizing the number of parallel links between high demand nodes but with a *diminishing return*. More formally, assume a network topology is represented by graph $G = (V, E)$ and each node has degree d. We define $L(i, j)$ to be the number of parallel links between node-pair (i, j). Let $T(i, j)$ be the amount of unsatisfied traffic demand. We define a topology G's utility function as follows:

$$Utility(G) = \sum_{\{i,j\} \in E} T(i,j) \times Discount(L(i,j)) \quad (1)$$

The *Discount* function can be defined in different ways; in Algorithm 5 and Algorithm 1's MP construction, we use

$$Discount(l) = \sum_{x=1}^{l} 2^{-x} \quad (2)$$

to reduce the utility of additional links exponentially. We can also explore other discount scaling, such as linear or factorial functions.

When the fabric is reconfigurable (as in OCS-reconfig), we collect the unsatisfied traffic demand every 50 ms and run Algorithm 5 to decide the new network topology. After the new topology is computed, we pause all the flows for 10 ms representing the reconfiguration delay of the OCS, apply the new topology, and then resume the flows with one or more corresponding physical links across the flow source and destination. The two-edge replacement algorithm from OWAN [87] in line 21 ensures the topology is connected, when we enable host-based forwarding.

F Modifications to SiP-ML

Since SiP-ML's SiP-Ring proposal is based on a physical ring topology, its reconfiguration algorithm has several constraints on wavelength allocation for adjacent nodes. Given that TopoOpt's physical topology is not a ring, directly applying SiP-Ring's optimization using the original C++ code causes SiP-ML to perform extremely poorly in our setup. To give SiP-ML a leg up, we observe that its formulation tries to optimize a utility function very similar to Equation 1 without the *Discount* part (i.e. $Discount = 1$), but with an integer liner program (ILP). While an ILP gives the optimal solution, its runtime makes it prohibitive for the amount of simulation parameters we explore. Therefore, we substitute the ILP

Algorithm 5 OCS-reconfig pseudocode

```
1:  procedure OCS-RECONFIG(V, T, d, L)
      ▷ Input V: Nodes in the network
      ▷ Input T: Unsatisfied traffic demand matrix
      ▷ Input d: Node degree limit
      ▷ Input L: Number of links between ordered node-pair, initially zero
      ▷ Output E: Allocated links, initially empty
        ▷ Initially, E is empty
2:      E = {}
        ▷ Initially, each node has d available tx and rx interfaces
3:      for v ∈ V do
4:          available_tx[v] = d
5:          available_rx[v] = d
        ▷ Create new links according to the demand list
6:      while ∃i, j < |V| : i ≠ j, available_tx[v_i] > 0, available_rx[v_j] > 0 do
          ▷ allocate a direct connection for the highest demand pair
7:        (v1, v2) = node-pair with highest demand in T
8:        e = NewLink(v1, v2)
9:        E = E ∪ {e}
          ▷ Increment the number of parallel links from v1 to v2
10:       L(v1, v2) += 1
          ▷ Scale the demand down by the number of links
11:       T(v1, v2) ×= 1/2
          ▷ Update available interfaces
12:       for v ∈ (v1, v2) do
13:           available_tx[v1] −= 1
14:           available_rx[v2] −= 1
                ▷ Stop considering nodes with zero available interfaces
15:           if available_tx[v1] == 0 then
16:               for u ∈ V do
17:                   Remove (v1, u)'s entry from T
18:           if available_rx[v2] == 0 then
19:               for u ∈ V do
20:                   Remove (u, v2)'s entry from T
        ▷ Ensure the network graph is connected
21:     2-EdgeReplacement(E, T)
        ▷ Updte route for host-based forwarding
22:     UpdateRoute(E)
23:     return E
```

with Algorithm 5 with $Discount = 1$, a heuristic that tries to achieve a similar goal.

Note that the SiP-ML paper has another design called SiP-OCS, which is similar architecturally to TopoOpt. In the paper, SiP-OCS is proposed as a one-shot reconfiguration approach due to the long reconfiguration latency of 3D-MEMS based OCSs.

G Cost of Network Components

Table 2 lists the cost of network components we use in Section 5.2, namely NICs, transceivers, fibers, electrical switches, patch panels, and optical switches. The cost of transceivers, NICs, and electrical switch ports is based on the lowest available prices in official retailer websites [18, 28]. Note that for 200 Gbps, we use more 100 Gbps ports and fibers, because they were less expensive than high-end 200 Gbps and 400 Gbps components, or their price was not available. To estimate the cost of electrical switch ports, we consider Edgecore

[4] 200 G transceivers and switch ports are estimated as 2× 100G cost.

Link bandwidth	Transceiver ($)	NIC ($)	Electrical switch port ($)	Patch panel port ($)	OCS port ($)	1×2 switch ($)
10 Gbps	20 [13]	185 [32]	94 [21]	100 [43]	520 [41]	25 [112]
25 Gbps	39 [15]	185 [32]	144 [23]	100 [43]	520 [41]	25 [112]
40 Gbps	39 [16]	354 [33]	144 [22]	100 [43]	520 [41]	25 [112]
100 Gbps	99 [12]	678 [34]	187 [24]	100 [43]	520 [41]	25 [112]
200 Gbps[4]	198 [12]	815 [35]	374 [24]	100 [43]	520 [41]	25 [112]

Table 2: Cost of network components.

bare metal switches with L3 switching and maximum number of ports to amortize the per port cost. The cost of NICs is taken from the Mellanox ConnectX series, and we consider two 2-port NICs as one 4-port NIC. We obtain the cost of the patch panel, OCS, and 1×2 optical switch directly from their suppliers, Telescent [43] and Polatis [41] (with 40% discount). The cost of transceivers matches that reported in Sirius [53].

To compute the network cost of Fat-tree and Ideal Switch, we consider number of nodes in a full bisection bandwidth Fat-tree. For example, a standard $k = 8$ Fat-tree has 80 switches with 64 ports, or 640 switch ports in total, in addition to 1 NIC per host and one transceiver per NIC and switch port. A TopoOpt system of 128 nodes with degree d uses $128 \times d$ NICs and transceivers, but $128 \times 2 \times d$ patch panel ports because of the look-ahead design. Note that the cost of optical components stays constant as link bandwidth increases, an inherent advantage of optics. Following prior work, we estimate the cost of fiber optics cables as 30 cents per meter [68] and select each fiber's length from a uniform distribution between 0 and 1000 meters [148]. We calculate the cost of TopoOpt based on $2d$ patch panels and 1×2 switches at each link to support its look-ahead design (§C). OCS-reconfig's cost is based on d OCSs connected to all servers in a flat topology.

H Impact of Server Degree on TopoOpt's Performance

Figure 27 shows the same setting as Figure 11 except that each server has a degree of eight ($d = 8$). The results show a similar trend: even though per server bandwidth has increased, the behavior of different network architectures remains consistent.

Next we do a sensitivity analysis of impact of server degree d on TopoOpt's performance. Specifically, we vary the degree of each server in TopoOpt for two link bandwidths: 40 Gbps and 100 Gbps. Figure 28 shows the trend for different DNN models. Both DLRM and CANDLE are network-heavy; therefore, they benefit more from the additional bandwidth obtained by increasing d. CANDLE's improvement is almost linear as degree goes up, as the strategy is closer to data parallel and the amount of bandwidth available to AllReduce operation increases linearly as well. In the case of DLRM, we observe a super-liner scaling when B = 100 Gbp because DLRM has one-to-many and many-to-one

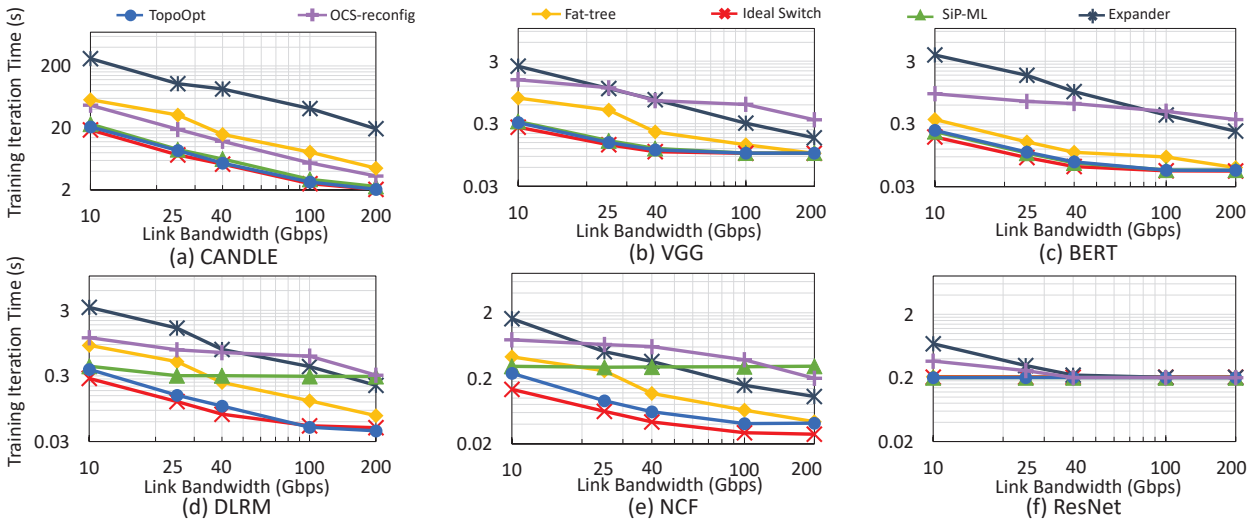


Figure 27: Dedicated cluster of 128 servers ($d = 8$).

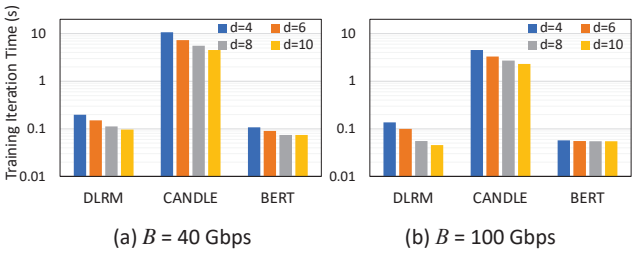


Figure 28: Impact of server degree (d) on performance.

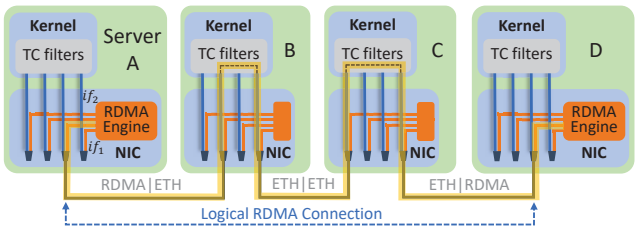


Figure 29: Host-based RDMA forwarding to create a logical RDMA connection between end hosts.

MP transfers which require a low hop count in the topology. As we increase d, TopologyFinder is able to find network topologies with much lower diameter, consequently benefiting the performance by both increasing bandwidth and reducing hop-count for MP transfers. Finally, BERT is mostly compute bound at higher bandwidth; hence, increasing the server degree and bandwidth per node has marginal impact on its iteration time.

I Enabling Host-based Forwarding in RDMA

To support a multihop TopoOpt interconnect using host-based forwarding, we enable RDMA RoCEv2 forwarding on all our HP NICs. RoCEv2 is an implementation of RDMA on top of UDP/IP protocol, by utilizing a particular UDP port (4791) and encapsulating an InfiniBand (IB) data packet. Hence, each RoCEv2 packet can be routed with its source and destination IP addresses. However, host-based forwarding is challenging in RDMA protocol, as the packet processing and memory access are offloaded to the NIC, and the host does not have access to individual packets. More precisely, if a packet's IP destination IP address does not match the NIC's IP address, the RDMA engine silently drops the packet.

To address this issue, we collaborated with engineers from Marvell, the provider of the firmware and driver for our HP NICs. The solution that came out of our collaboration does not require proprietary software or firmware, and is applicable to commodity NICs with the same ASIC. We will release our scripts publicly. At a high-level, we use a feature called *NPAR*, or *network partitioning*. It allows us to split each 25 Gbps physical interface into two *logical interfaces* in the hardware level: if_1 and if_2, as shown in the right-most port of server A in Figure 29. if_1 is a normal RDMA interface, where the RDMA engine of the NIC bypasses the kernel, and it has an IP address. This enables the upper layer software to consider if_1 as a normal RDMA interface. However, if_2 does not have an IP address and RDMA is disabled. if_2 has a different MAC address from if_1, and we use this address to split the traffic across if_1 and if_2. The traffic that needs to be forwarded uses the MAC address of if_2 and hence is delivered to the host networking stack instead of NIC's RDMA engine.

Furthermore, we establish a set of `iproute`, `arp`, and `tc flower` rules in Linux to enable the proper forwarding of packets. If two servers are directly connected, such as the third port of server A and the second port of Server B in Figure 29, we only need to indicate the outgoing interface

on each of these servers. RDMA engines will handle the communication. However, for the connection between server A and D, we set the `iproute` and `arp` tables on server A and server D to dictate which port the traffic should go out, as well as the proper MAC address of the next server in the forwarding chain. In this case, the packets are delivered to the kernel. Then, on servers B and C, we set the `tc flower` rules to forward the packets to the next server with the proper MAC address. In these `tc flower` rules, we look-up the final destination IP and assert the routing that was computed by our algorithm.

Walk-through of an example of a packet going from server A to server D. In Figure 29, the RDMA engine of server A assumes server D is connected on the third port. It uses the kernel's routing tables for the destination MAC address, which is set to the MAC address of if_2 of the second port on server B. Therefore, a packet which starts as an RDMA packet of server A is treated as an Ethernet packet when it arrives at server B, and goes to server B's kernel. In the kernel, based on the packet's final destination IP of server D, server B redirects the packet to the fourth port, with destination MAC address set to if_2 of server C. In this connection, the packet is treated as a normal Ethernet packet. Finally, on server C, the kernel rewrites the destination MAC address to that of if_1 on the third port of server D, and redirects it to that port. In this connection, the outgoing Ethernet packet is considered an RDMA packet because of the destination MAC address. For the reverse connection from server D to A, the same process happens in reverse, to support a bidirectional connection.

With these forwarding rules, we construct logical RDMA connections between all pairs of servers. Upper layer communication libraries such as NCCL requires all-to-all connectivity, and they will utilize these connections. We also modify NCCL to be topology-aware, as certain pairs of servers are only connected through specific ports.

Compared to native point-to-point RDMA, this approach takes a performance penalty. Our experiments indicate the overhead is negligible when the amount of forwarded traffic is small. Our NICs currently support TCP forwarding offload. With firmware and driver modifications or future versions of the NICs, they will also support RDMA forwarding offload. This will further reduce the overhead of our approach.

ModelKeeper: Accelerating DNN Training via Automated Training Warmup

Fan Lai, Yinwei Dai, Harsha V. Madhyastha, Mosharaf Chowdhury
University of Michigan

Abstract

With growing deployment of machine learning (ML) models, ML developers are training or re-training increasingly more deep neural networks (DNNs). They do so to find the most suitable model that meets their accuracy requirement while satisfying the resource and timeliness constraints of the target environment. In large shared clusters, the growing number of neural architecture search (NAS) and training jobs often result in models sharing architectural similarities with others from the same or a different ML developer. However, existing solutions do not provide a systematic mechanism to identify and leverage such similarities.

We present ModelKeeper, the first automated training warmup system that accelerates DNN training by repurposing previously-trained models in a shared cluster. Our key insight is that initializing a training job's model by transforming an already-trained model's weights can jump-start it and reduce the total amount of training needed. However, models submitted over time can differ in their architectures and accuracy. Given a new model to train, ModelKeeper scalably identifies its architectural similarity with previously trained models, selects a parent model with high similarity and good model accuracy, and performs structure-aware transformation of weights to preserve maximal information from the parent model during the warmup of new model weights. Our evaluations across thousands of CV and NLP models show that ModelKeeper achieves 1.3×–4.3× faster training completion with little overhead and no reduction in model accuracy.

1 Introduction

Modern machine learning (ML) clusters train thousands of deep neural networks (DNNs) every day [37, 67]. For a specific ML task, ML developers often start with exploring various model architectures using Neural Architecture Search (NAS) to find the one with desired accuracy [77]. In preparation for model serving, developers may train tens of models to customize the latency-accuracy trade-off across hardware [21, 35], to organize weak and powerful DNNs into different inference stages for fast feature extraction [20], and/or to dynamically select tens of models and combine their predictions to maximize ensemble accuracy [26, 30, 65]. Overall, from inception to deployment, ML development often requires training hundreds of models across developers [62, 75].

Naturally, many recent advances in ML training optimizations have focused on faster DNN execution, e.g., by increasing parallelism [50, 70], improving communication [40, 55], or increasing GPU utilization [28, 68, 69, 73]. However, little has been done to exploit the natural similarity between models that are trained as part of the same NAS process, models targeting the same ML task in different hardware, or models embedded in different applications. Indeed, our analysis of three large CV and NLP model zoos shows that more than 60% of widely-used models can find an architecturally similar counterpart within the same zoo (§2.2).

In this paper, our key insight is that *one can reduce the amount of training needed for model convergence by leveraging a well-trained model's weights to warm up the training of a new model.* This is because any DNN model is fundamentally a computation graph of tensor weights and operators; transforming weights of trained models with similar architectures to a new model can accelerate model convergence (similar to transfer learning [60, 71] but across architectures).

Despite the potential for large benefits, there exists little systematic support for automated repurposing of weights. Today's frameworks may provide pre-trained models, but are limited to a few models and specific datasets, and/or require domain knowledge to manually search, transfer and contribute a trained model's weights [6]. As such, ML developers have to train models from scratch more often [56]. A few recent AutoML frameworks (e.g., Retiarii [77]) repurpose trained models. However, they are limited to individual jobs within a NAS task because they rely on the lineage of model mutation to enable the transfer. When models are submitted by various developers and/or frameworks with distinct architectures and performance requirements, these solutions do not apply.

We introduce ModelKeeper, a cluster-wide training warmup system, to reduce the training execution needed for model convergence via automated model weight transformation (§3). ModelKeeper adaptively manages a collection of trained models (i.e., *model zoo*) from prior training jobs corresponding to different ML tasks. For a new training job, ModelKeeper selects and transforms a trained model's weights (i.e., *parent model*) to the training model (i.e., *query model*) before training takes place. It can benefit various ML applications, including exploratory training (e.g., improving Retiarii [77] further) and general training (e.g., using PyTorch [9]) of CV/NLP models, with few-lines-of-code change.

ModelKeeper addresses two primary challenges toward selecting a suitable parent model and repurposing its weights. First, ModelKeeper must determine similarity between two models (§4.1). Intuitively, we can treat each DNN model as a directed graph, where nodes represent tensors (layers) and edges represent data flows, and use heuristics for the classic NP-hard graph edit distance problem [31] to find the matching similarity. However, maximizing matching by skipping nodes can be harmful because the computation of each tensor affects that of the subsequent ones in a trained parent model. To this end, we present a structure-aware dynamic programming approach to capture the similarity (transformable tensor weights) between two models. To scale to real-world zoos with thousands of models, we then introduce a two-stage hierarchical search algorithm to identify similar models efficiently.

Second, perfect matching is unlikely as two models are seldom identical. Therefore, given many candidate parent models with different similarity scores and each with different accuracy, which one to pick and then how to transform its weights to the query model (§4.2)? A more similar parent model enables transforming more weights, while a more accurate one implies a better training jump start after the transformation. When the two are at odds, we adopt a bucketing heuristic: potential parent models are put into different buckets in terms of their similarity to the query model, grouping comparable parent models together. We then pick the most accurate parent from the bucket containing the most similar parent models. Nevertheless, tensor mappings from the parent to the query model can be incomplete (e.g., due to non-identical architectures). To preserve maximal parent model information, we introduce width and depth operators to transform parent model weights into the query model with negligible overhead.

We have integrated ModelKeeper with four popular ML frameworks (§5): Ray [49], AutoKeras [41], MLFlow [75], and Microsoft NNI with Retiarii backend [77].[1] Our evaluations across thousands of DNN training jobs in CV and NLP applications (§6) show that ModelKeeper can save 23%–77% total amount of training needed (i.e., 1.3×–4.3× faster training) than the state-of-the-art without model accuracy drop, while efficiently serving cluster-scale warmup requests.

Overall, we make the following contributions in this paper:

1. We present ModelKeeper, a system to enable automated training warmup for faster DNN training in clusters;
2. In order to maximize training speedup, we demonstrate how to scalably compute similarities between models and how to transform an already-trained model's weights to a yet-to-be trained model with little overhead;
3. We integrate ModelKeeper with multiple advanced ML frameworks, and evaluate it across thousands of CV and NLP models to show large improvements.

[1] ModelKeeper is available at https://github.com/SymbioticLab/ModelKeeper.

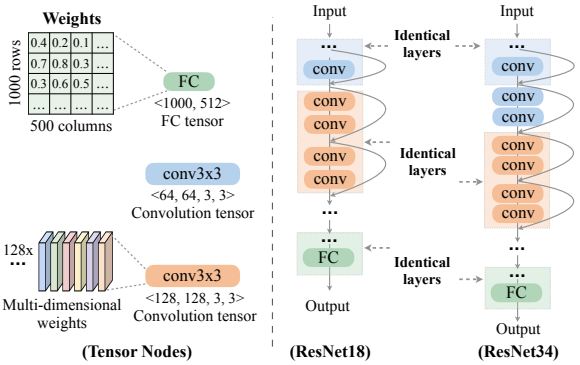


Figure 1: ***A DNN model is essentially a graph of tensors. Model outputs are determined by tensor weights and their control flow.***

2 Background and Motivation

2.1 DNN Model Training

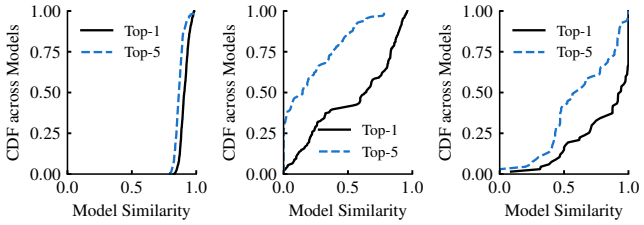
Modern DNN frameworks represent DNN computations as a directed computation graph with tens to thousands of nodes across branches (Figure 1) [9, 38]. Each node implies a mathematical tensor operation (e.g., matrix multiplication or convolution) along with its tensor weights and input, where weights are n-dimensional arrays typically consisting of floats. DNN training often covers thousands of iterations across mini-batches of data to minimize the training loss. In each iteration, the computation graph takes a data mini-batch as the input, and performs a (1) *forward pass*, where each node conducts the tensor operation on the output of parent nodes to get the training loss regarding the model output and ground truth; and a (2) *backward pass*, which updates the weight values, from the last to front tensors, using the gradients derived by the training loss with respect to the current weight. Therefore, the DNN model is essentially a graph of weights orchestrated by tensor operators, and training searches the best weight values.

2.2 Opportunities for Repurposing Models

In this paper, we focus on *reducing the amount of training needed* to train a new model by automatically repurposing the weights of previously trained models. Our approach of warming up the weights of a new model before its training starts is based on the following observations.

Pervasive model similarity. With the rapid increase in the number of ML training jobs in datacenters [28, 37], similarities between training jobs are increasing too [67]:

- First, for a specific ML task, ML developers often explore various model architectures using Neural Architecture Search (NAS) to find the preferred model architecture (e.g., better capacity-accuracy frontier [77]), or to investigate the performance consistency of new optimizations across models (e.g., ML ablation study) [48]. For example, Microsoft tuning clusters perform as many as 75 exploratory training jobs in median for user apps [46].
- Second, in preparation for ML deployment, developers can train dozens of models to either customize the latency-



(a) NASBench (b) Imgclsmob (c) HuggingFace

Figure 2: ***Pervasive model similarity in today's model zoos. We measure the top-1 and top-5 architectural similarities of each model to other models, and report the distribution across models. 1 indicates identical model architectures.***

accuracy tradeoff across hardware (e.g., in video analysis systems [35, 39]), or to dynamically select tens of models and combine their predictions in order to maximize accuracy under changing loads in today's ensemble-serving systems [30, 65] (e.g., AWS Autogluon [26]).

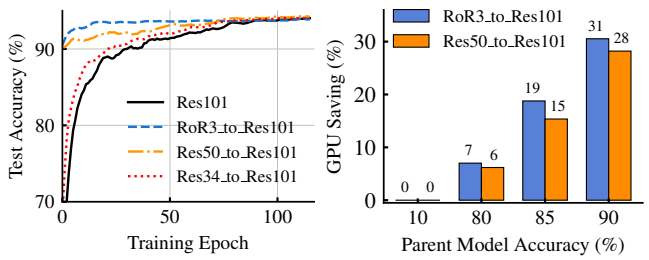
- Third, the potential for similar models increases with increasing users. For example, over 100K ML models are submitted to Kaggle competitions each month [3]. Each competition can have thousands of participants developing their models independently, and participants are reported to have trained many similar models [22].

Indeed, our analysis of three large public model zoos – Imgclsmob [10] for ImageNet classification (435 models), HuggingFace [2] for English text generation (2.5K models), and NASBench [24] for NAS task (16K models) – reinforces these observations. Figure 2 reports the pairwise architectural similarity across models in each model zoo. We measure the similarity of each model to other zoo models[2] in terms of the normalized graph edit distance of two directed model computation graphs [27] ($\in$ [0, 1]), where 1 indicates identical graphs. We observe that more than 60% of the models have at least one other model (top-1) in the zoo with a similarity over 0.6. Pervasive similarity is prominent in all these model zoos because modern models often rely on similar architecture designs but with wider/deeper layers or branches. For example, convolution layers are widely-used in CV models [33, 36], while NLP models are often stacked by attention layers [63].

Similar models can warm-start training. Recent theoretical [60] and empirical [71] efforts from the transfer learning community show that inheriting well-trained parent model weights can speed up the training of a new model, because this warm start enables an informed weight initialization (e.g., training from the basin of loss curvature). Yet, different from their focus that manually transfers the same model across datasets for better model generalization accuracy [53, 78], we notice that transforming a trained model's weights to a new model (i.e., across architectures) can accelerate its training.

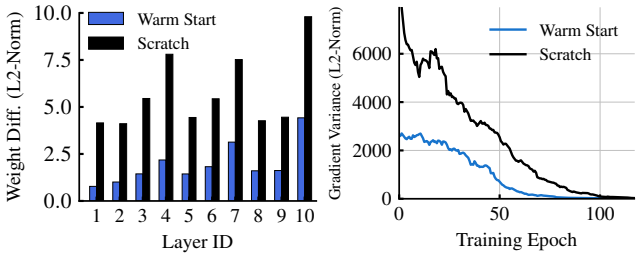
Consider the training of ResNet101 on CIFAR-10 dataset

[2]To avoid over-optimistic identification of model similarity, we removed identical models in each zoo and focus on different model architectures.



(a) Warm start accelerates training. (b) Parent model accuracy matters.

Figure 3: ***Transferring model weights from well-trained models with similar architectures can accelerate new model training.***



(a) Smaller divergence to the optimal. (b) Smaller gradient variance.

Figure 4: ***Warm start provides better initial weights search space. We use RoR3 to warm start ResNet101.***

as an example. We copy the tensor weights of a well-trained parent model (e.g., ResNet50 or RoR3 [76]) to the ResNet101 tensor if two tensors have identical properties (e.g., same operator and weight dimensions), while the rest of the training proceeds as normal. We notice that (1) warmup training can reduce the amount of training needed, while obtaining the same final accuracy to that of training from scratch with random weight initialization (Figure 3(a)); and (2) the savings are more encouraging when inheriting from more similar models – similarity of ResNet34, ResNet50, and RoR3 to ResNet101 is 0.19, 0.48, and 0.85, respectively – and better performing models (Figure 3(b)), which respectively determine whether it is possible and beneficial to transform the weights.

These improvements are because they speed up the search in the space of weight values. If we consider ResNet101 as an example, (1) warming it up using RoR3's weights before training starts results in a smaller distance to the final weights achieved when the model converges (Figure 4(a)), and (2) during the training, this informed weight initialization enables smaller gradient variance (i.e., more consistent gradient directions) towards the basin of loss curvature (Figure 4(b))), thus requiring fewer iterations to convergence in theory [12, 53].

3 ModelKeeper Overview

ModelKeeper is an automated training warmup system for various ML tasks that accelerates DNN training by warm-starting models with weights from already-trained models.

Design Space Large training clusters are shared between users with varying expertise, and they can train a large num-

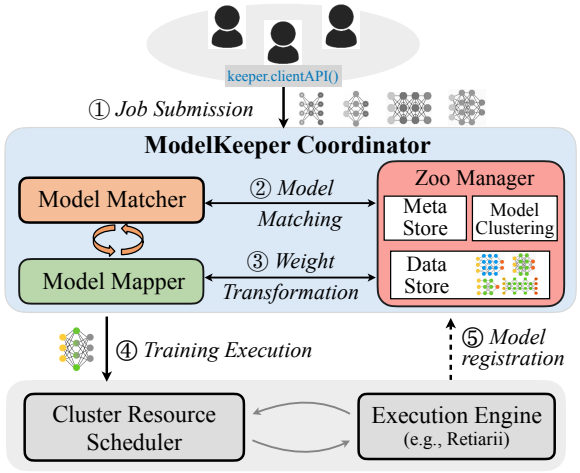


Figure 5: ***ModelKeeper architecture. It can run as a cluster-wide service to serve different users and/or frameworks.***

ber of jobs with different model architectures. Consequently, ModelKeeper must minimize the information needed and overhead incurred for each training model (i.e., *query model*), while offering users the flexibility in their request (e.g., using ImageNet model zoo to warm start models on other image datasets). In fact, determining which dataset (model zoo) as the source to transfer is as yet an open problem in the transfer learning community [45, 71, 78]. ModelKeeper is complementary to and benefits existing ML efforts as it automates training warmup (e.g., searching, transforming, and contributing a trained parent model's weights) for a given model zoo, instead of making the developer keep tracking all models and handcraft which model to repurpose [6]. We empirically show that ModelKeeper can benefit the model training across datasets too (§6.4).

For a given model zoo, the effectiveness of transforming parent model weights relies on two key aspects: (i) *Model similarity*: it dictates the similarity of two model architectures, including the weights shape and operation type of a tensor; and (ii) *Parent model accuracy*: it determines the value of transformation. Having architectural similarity is the prerequisite to transforming more weights information of a parent model, while better parent model accuracy implies potentially better warm start after transformation.

As such, ModelKeeper should repurpose a parent model with large similarity and better accuracy. We provide the theoretical analysis to support why ModelKeeper can benefit model convergence following this principle in Appendix A.

System Components ModelKeeper is a complementary system to existing ML training (Figure 5), and has integrations with various frameworks (e.g., Microsoft NNI [4] and Ray [49]). It consists of the remote coordinator, which serves user query models before their training executes, and the client agent that allows users to submit model warmup requests. ModelKeeper coordinator employs three key components to warm up models by transforming a trained model's weights:

- *Model Matcher*: to identify architecturally similar models in the zoo of trained models;
- *Model Mapper*: to select a zoo model with good architectural similarity and accuracy as the parent model, and transforms the parent model weights to the query model;
- *Zoo Manager*: to adaptively manage zoo models that can be submitted from users to support transformation at scale.

Figure 6 reports the example interface on the client agent, where the user benefits from ModelKeeper with a few lines of code in training (Coordinator interfaces are in Section 5).

```
from modelkeeper import ModelKeeperClient

def training_with_keeper(model, dataset):
  # Create client session to keeper coordinator
  keeper_client = ModelKeeperClient(coordinator_ip)
  warmed_model, meta = keeper_client.query_for_model(
              model, meta={'data': 'Flowers102',
              'task':'classification', 'tags': None})

  acc = train(warmed_model, dataset) # Training starts

  # Register model to ModelKeeper when training ends
  keeper_client.register_model(warmed_model,
          meta={'data': 'Flowers102', 'accuracy': acc,
          'task': 'classification', 'tags': None})
  keeper_client.stop()
```

Figure 6: ***Code snippet of ModelKeeper client service APIs.***

Training Lifecycle When the developer creates a new training job, ① she first initiates a client connection to the remote ModelKeeper coordinator, and then issues a query with the specified job meta information. ModelKeeper client agent will automatically extract the model information needed (e.g., model computation graph) and issue a request (mostly size < 1 MB) to the coordinator. ② Upon receiving the request, Matcher consults its metadata store, identifies zoo models that the user can access and meet the specified tag (e.g., name of the preferred parent models), and measures their architectural similarity to the query model. ③ Mapper selects a parent model with large architectural similarity and good accuracy out of these zoo models. Thereafter, it loads model weights of this parent model from the data store, and transforms parent model weights, based on pairwise tensor mapping from the Matcher, to the query model. Note that this transformation only updates tensor weight values, while others (e.g., model architecture) remain the same. ④ The coordinator responds to the developer with warmup model weights, and the rest of the training remains as usual. ⑤ When the training completes, ModelKeeper can automatically register the trained model to the Zoo Manager to benefit future jobs.

4 ModelKeeper Design

In large shared clusters, models are often submitted by various developers and/or frameworks with diverse architectures at different points in time. The large variety in model architectures and accuracy characteristics lead to novel system challenges in automating weight transformation from a parent model with high similarity and better accuracy:

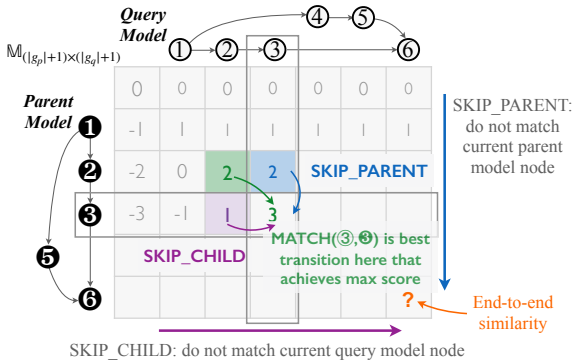


Figure 7: ***ModelKeeper relies on dynamic programming-like heuristics to measure graph-level model architectural similarity.***

- Having similar model architectures is the prerequisite to transforming weights across architectures. How to identify more architecturally similar zoo models (§4.1)?
- As the similarity and accuracy of many potential parent models can come at odds, which one to pick and then how to transform its weights to the query model even in the presence of non-identical architectures (§4.2)?
- New training jobs and trained models can join the cluster on the fly. How to serve user warmup requests at scale for high throughput clusters in the wild (§4.3)?

4.1 Matcher: Identify Similar Models

The similarity between two model architectures determines the number of tensor weights that we can transform. Hence, we need to identify the graph-level architectural similarity of each parent and query model pair (Figure 7) and their pairwise tensor mappings. It is tempting to model it as a classic NP-hard graph edit distance (GED) problem [15] by treating tensors as nodes and data flows as edges with the goal to morph one graph to the other with minimum edits (e.g., add/delete a node). However, model matching encounters new challenges: (i) *Prefix Preference*: we prefer to match the prefix over the suffix of model graphs. Because prefix tensors are more transferable since they capture general input features (e.g., image color blobs) [53, 71]. Moreover, model weights are trained systematically over tensors, so any edit on prefixes can result in information loss to subsequent tensors [25]; (ii) *Partial Matching*: we can partially transform the weights of a smaller dimensional tensor to a wider one to match more tensors, or postpone its matching to preserve its exact weights information; and (iii) *Scalability*: as each model can consist of thousands of nodes across branches, capturing the similarity to thousands of zoo models is challenging.

ModelKeeper Matcher measures the graph-level similarity of models, in terms of the total number of transformable weights after mapping tensor pairs from the parent to the query model. It uses the widely-used ONNX tool [7] to extract the computation graph. ONNX supports various model formats (e.g., Tensorflow and PyTorch), which allows us to perform the cross-framework transformation.

Structure-Aware Pairwise Model Matching We introduce a dynamic programming-based heuristic to measure the end-to-end similarity (i.e., number of weights to transform) of two models. It relies on a similarity table $\mathbb{M}_{(|g_p|+1)\times(|g_q|+1)}(i,j)$ to record the best similarity after matching the prefixes of the parent and the query model. Here, $|g_p|$ and $|g_q|$ respectively denote the number of tensors of the parent model and the query model. Then, it enumerates plausible matching operations from previous states (e.g., $\mathbb{M}(i-1,j-1)$), and takes the operation that can acquire the maximum similarity to enter the next state (i.e., $\mathbb{M}(i,j)$).

Figure 7 shows the execution of our structure-aware matching algorithm. It traverses the similarity table in the topological order of graph tensors. This allows us to embed graph-level information while prioritizing the match of prefixes. To advance to the current tensor pair (i, j), it enumerates three plausible operations:

1. *MATCH*: transform weights of i's parent to j's parents.
2. *SKIP_PARENT*: give up transforming tensor i's parent;
3. *SKIP_CHILD*: give up transforming to tensor j's parent;

Then, it updates the table to obtain the maximum similarity after each step based on previous states as follows:

$$\mathbb{M}(i,j) = \max_{k \in parent(i)} \begin{cases} \mathbb{M}(k, j_{parent}) + MATCH(k, j_{parent}) & (1) \\ \mathbb{M}(k,j) + SKIP_PARENT & (2) \\ \mathbb{M}(i, j_{parent}) + SKIP_CHILD & (3) \end{cases}$$

To get the overall transformable weights, we can reward each operation based on the number of tensor weights transformed. When tensor i and j belong to the same operator (e.g., convolution), the fraction of transformed weights along each weights dimension in *MATCH* operation (1) is:

$$MATCH(i,j) = \frac{\prod_{dim=1} \min(dim(i), dim(j))}{\prod_{dim=1} \max(dim(i), dim(j))} \quad (\in [0,1]) \quad (4)$$

Otherwise, we assign *MATCH(i, j)* to -1, as this transformation is useless and even loses the weights of that parent model tensor. Similarly, SKIP_PARENT is set to -1 as it loses the parent model tensor, and SKIP_CHILD is 0, since it does not transform the parent model tensor.

Capturing the graph-level similarity is more challenging when tensor j of the query model is the intersection of multiple upstream branches. Because different upstream branches to j may follow the same branch of the parent model during their matching, leading to repetitive (conflicting) matching. As shown in Figure 7, when we reach (❻, ⑥), branch (②→③) and (④→⑤) may both be matched to (❷→❸) that maximizes their own similarity. To avoid conflicting matching, the similarity to j is the sum of upstream branches ($\mathbb{M}(i,j) = \sum_{k \in parent(j)} \mathbb{M}(i,k)$), and we greedily adopt the matching of a branch to tensor j, whose trajectory achieves the largest similarity (i.e., match ②→③ to ❷→❸), to maximize their sum. Meanwhile, we discard the trajectory of other

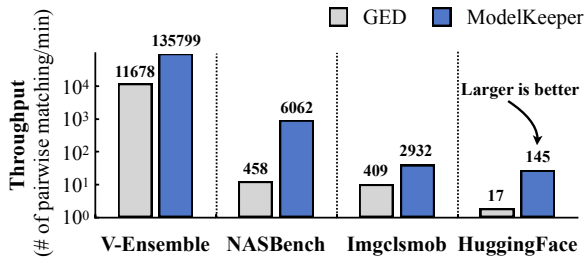


Figure 8: ***Keeper is order-of-magnitude more scalable than existing GED. V-Ensemble is a model zoo for ensemble training (§6.1).***

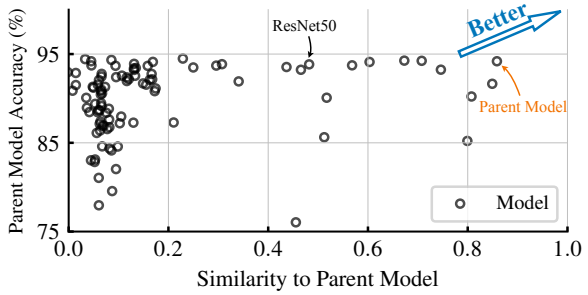


Figure 9: ***Models vary in accuracy and architecture (Imgclsmob zoo). We measure their similarity w.r.t. ResNet101, and prefer to transform a parent model with better similarity and accuracy.***

branches where conflict exists. As such, branch ④→⑤ takes the inferior match (❺), where ④ is skipped.

The last entry of the table, i.e., $\mathbb{M}(|g_p|,|g_q|)$, gives the end-to-end similarity. Note that we can learn the pairwise tensor mappings by backtracking operations taken to reach $\mathbb{M}(|g_p|,|g_q|)$ over the table in linear time. For a specific model, our heuristic will naturally treat the model itself as the most similar model, because matching will always take operation $MATCH(i,i)$ in each step to maximize the similarity.

Figure 8 reports that our pairwise matching can match thousands of model pairs in a second, and achieves higher throughput than the state-of-the-art GED solution [57] in this model matching scenario. More importantly, our empirical results report that our structure-aware matching achieves better training warmup than the GED solution (§6.3).

4.2 Mapper: Transform Maximal Parent Information

The effectiveness of weight transformation is determined by the similarity (transfer more weights) and accuracy (transfer better weights) of parent models. Unfortunately, it is impractical to pick the optimal parent model, since the performance of transformation can only be known after training each derived warmup model to converge. Worse, the variety of model similarity and performance leads to the tussle in selecting the parent model. As shown in Figure 9, while some models (e.g., ResNet34) possess high accuracy, their low similarity to ResNet101 can cap the number of weights that can transform. Next, we introduce Mapper to exploit the sweet spot of both aspects, and then to transform maximal parent model weights in the presence of partial matching.

As shown in Algorithm 1, Mapper relies on Matcher to

Input: Query model q, Model Zoo M
Output: Warmup Model Weights

```
1  NumOfBuckets B = 10                          ▷ Model similarity ∈ [0, 1]
2  Function GetModelSim (Query q, Models M)
       /* Structure-aware matching for model similarity. */
3      topo_query_tensors = SortByTopo(q)
4      model_similarity = { }
5      for model m ∈ M do
6          similarity_table = zeros(|g_m|+1, |g_q|+1)
7          for tensor i ∈ CachedModelTopo(m) do
8              for tensor j ∈ topo_query_tensors do
                   /* Enumerate and merge intersection. */
9                  similarity_table[i][j] = Equation (1-3)
10         model_similarity[m] = similarity_table[|g_m|][|g_q|]
11     return model_similarity
12 Function QueryForModel (Query q, Model Zoo M)
       /* Bucket models in terms of similarities. Pick the model in
          the top-similar bucket with the best performance. */
13     model_similarity = GetModelSim(q, M)
14     top_similar_bucket =
         BucketBySimilarity(model_similarity, B).first
15     for model ∈ top_similar_bucket do
16         if model.perf > best_parent.perf then
17             best_parent = model
       /* Perform width and depth weight transformation */
18     warmup_weights = TransWeight(best_parent, q)
19     return warmup_weights
```

Algorithm 1: Select the parent model to transform.

identify similar models (Line 2). As having a good similarity is the prerequisite for transformation, we need to first ensure picking similar models. To this end, Mapper takes the popular bucketing strategy to allocate models into B buckets in terms of their similarity (Line 14). Taking Figure 9 as an example, with $B = 10$ by default, *bucket* 10 will accommodate models with similarities between 0.9 and 1.0, so models in the same bucket have comparable similarities. Then, Mapper traverses from the last bucket (*bucket 10*) to the first until reaching the first one with nonempty models (*bucket 9*), from which it selects the model with the best performance as the parent model (Line 15). As such, the parent model approaches the boundary of better model similarity and accuracy. Later, Mapper performs structure-aware weight transformation to initialize the query model weights (Line 18).

Information-Preserving Weight Transformation To maximize the end-to-end number of weights to transform, Matcher allows partial matching: it may map a small tensor of the parent to a wider one of the query model, or skip the

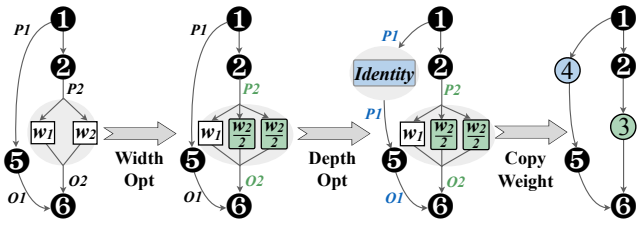


Figure 10: ***Width and depth operator to transform parent model.***

mapping of some tensors in the parent or the query model. Here, the straw-man solution (e.g., in Retiarii [77]), which transfers the weights of parent model tensors if and only if two tensors are identical, can be suboptimal (§6.3), since losing the parent model tensor can make the transfer of its subsequent tensors useless.

To preserve maximal parent model information under partial mappings, Mapper employs a width operator and a depth operator, which extend the well-known ML technique for function-preserving model transformation (e.g., Net2Net [17] and Network Morphism [66]). But unlike existing model transformation techniques [41], which are limited to *expand* the depth and width of a pre-determined model, or complicated transfer learning (e.g., knowledge distillation [34]) that requires additional computation (e.g., pre-training) and/or intrusive implementation, our operators transform the parent model weights into the query model with little overhead.

Our graph-level transformation proceeds in the topological order of tensors. Mapper handles the expanding case similar to today's function-preserving transformation (Figure 10): (i) to transform a parent model tensor to a wider query model tensor, the width operator copies the parent model weights to its mapping tensor of the query model, and pads the rest of the columns via weighted replication from other columns; and (ii) when the mapping requires inserting a new tensor into the parent model (i.e., SKIP_CHILD), the depth operator will initialize the weights of this mapping tensor to be an identity tensor. i.e., this tensor will directly pass the output of its parent tensors to the child tensors, in order to keep the same parent model's output. Readers can refer to Net2Net [17] for more details. We note that both expanding operations, in theory, can preserve the parent model information (i.e., with the same tensor output) for many tensor operators (e.g., the wide-used full connection and convolution layers).

The pruning case, however, cannot preserve full parent model information, because we lose some tensor weights of the parent model in transformation. Our solution is inspired by today's ML model pruning criteria [32]. Specifically, when we need to fit wider tensor weights (i.e., with larger array dimensions) to a smaller dimensional tensor of the query model, the width operator will progressively pick and copy the largest weight values of the parent model tensor to the mapping tensor of the query model. This is because, intuitively and empirically, larger magnitude values often have more impact on the model output [14]. From the depth perspective, when we skip transforming (i.e., SKIP_PARENT) a parent model tensor, the depth operator will add noise to the weight values of that tensor's neighbors. It disturbs the affinity of trained parent model weights so that neighboring tensors can still keep most information while being able to learn new weights [51].

Our transformation can be applied to various models for informed weight initialization. Thereafter, training proceeds as normal, and the warmup model will gradually converge the weight values that fit its architecture the best. As a generic system, ModelKeeper can accommodate other transformation techniques too as they become available. We provide a theoretical analysis of our transformation in Appendix A.2, and empirically show performance improvements using our transformation over its counterparts (§6.3).

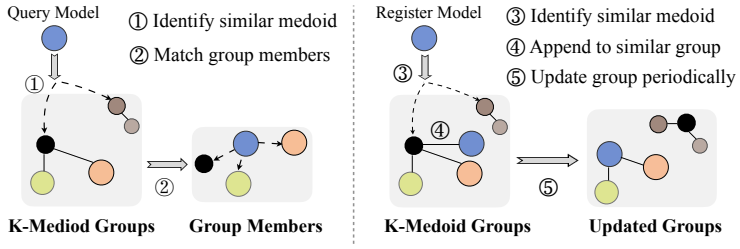
4.3 Zoo Manager: Transform Effectively At Scale

In reality, cluster users register their trained models to the model zoo on the fly, leading to scalability and performance challenges. First, while gathering more models increases the opportunity to transform better parents, the ever-growing number of models (e.g., > 70K models in the HuggingFace model hub of all tasks [2]) and model size (e.g., NLP models can be tens of GBs [16]) can lead to large matching overhead and storage cost. Moreover, models registering to the zoo may have low accuracy (e.g., due to insufficient training), which can harm the effectiveness of weight transformation. As such, ModelKeeper employs a Zoo Manager to support effective transformation at scale under dynamics.

Two-Stage Hierarchical Model Matching Despite being able to match thousands of lightweight CV models every minute (Figure 8), our pairwise matching heuristic can still be insufficient for model zoos with tens of thousands of models or complicated model architectures (e.g., NLP models). For example, to serve a query model using the HuggingFace model zoo for English text generation (2.5K models), performing pairwise matching on these zoo models can take ∼17 minutes, namely, 2.5K models over the throughput (145 matching/minute). This long search time is further exacerbated in today's large cluster with sub-minute job arrivals [37], eventually hurting the user experience.

To ensure an *interactive* service, Zoo Manager adaptively clusters zoo models into a well-defined number of groups, whereby Matcher can perform *two-stage* matching to reduce the number of matching pairs needed to identify more similar models. Intuitively, models with similar architectures would have comparable model similarity to the same query model, so we may be able to cluster zoo models into multiple groups, and then perform pairwise matching on the group members of top similar model groups. However, it is non-trivial to decide what features to use for clustering models, and how many groups are needed. Clustering too few groups does not scale down the problem enough, while too many can lead to a large overhead in identifying which group to prioritize.

We deploy K-medoids clustering [52] to combine pair-



(a) When query model arrives (b) When new zoo model joins

Figure 11: ***Matcher clusters models into groups to reduce the search space, and then performs model matching within groups.***

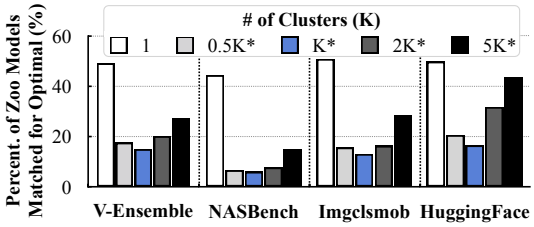
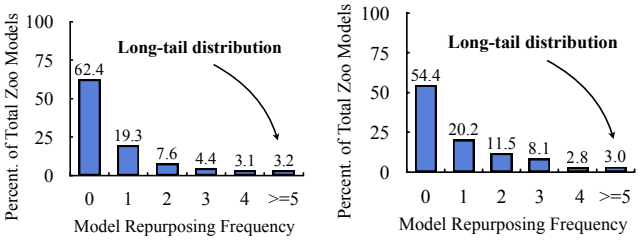


Figure 12: ***ModelKeeper can find the optimal number of clusters K^* in hierarchical matching, and can identify the most similar models with fewer zoo models (e.g., 5% in NASBench) needed to explore.***



(a) Imgclsmob model zoo. (b) NASBench model zoo.

Figure 13: ***A few zoo models are more frequently repurposed as the parent by Keeper. Numbers are from our evaluations (§6.2).***

wise model matching and clustering to find a sweet spot. K-medoids can directly take the distance of two points as input to minimize the distance between data points and their cluster center. Here, models can be taken as different points, and the distance is the reciprocal of their similarity. Compared to other clustering methods (e.g., K-means), K-medoids circumvents the need for embedding complicated model graphs, and it is more compatible with pairwise model matching.

As shown in Figure 11, when a query model arrives, Matcher identifies its similarity to each group medoid, and then conducts pairwise matching on the members of top similar groups. Similarly, when a new model registers, Matcher measures its similarity to group medoids, and assigns this new model to the group whose medoid is the most similar. This enables interactive queries to the latest models. Later, Matcher periodically triggers K-medoids to update the clustering.

To select the most similar models for each query model, Matcher identifies the best group medoid i by performing K pairwise matching, followed by K_i runs to match the members of group i. Assuming a zoo of M models ($M = \sum_i^k K_i$), to minimize the average matching runs on each group (i.e., $\min\big((K+K_i)/K\big)$), we can get the optimal number of groups $K^* = \sqrt{M}$. Figure 12 reports that, compared to the non-clustering design (i.e., $K = 1$), this two-stage design requires matching only 5%-16% of all zoo models to identify the most similar models, thus reducing the query hang time (§6.3).

Capping Zoo Size Hosting all zoo models can consume noticeable storage space. For example, the HuggingFace model zoo takes tens of TBs of storage [2]. In fact and understandably, we notice that a small portion of zoo models are more frequently repurposed than others (Figure 13). This is because certain models contain more similar blocks to other models (e.g., ResNet50 is more likely to be used to warm up other large ResNet models than ResNet18).

To harvest more warmup opportunities subject to the zoo capacity limit, we can formulate it as a knapsack packing problem, where each item (model) is associated with a weight (model size) and a value (repurposing frequency as the parent model), and our goal is to maximize the total value achieved. Namely, warm up as many jobs as possible (aka maximum total repurposing frequency). As such, solving this packing problem enables us to identify which item (model) to keep in the knapsack (model zoo). But on the other hand, models that are popular to train can change over time. For example, users incline to train more recent and/or advanced models. To account for the temporal variation in the repurposing frequency of each zoo model, we take the moving average of model values (e.g., decaying their repurposing frequency by 0.9 every day), and trigger the packing solver upon reaching the storage limit. We show that ModelKeeper can perform well even under severe storage limit (§6.4).

Avoiding Low-Accuracy Models Low accuracy models registering to the zoo (e.g., due to user error) not only wastes storage but can harm the transformation, so we need to ensure the zoo uses models with decent accuracy. To this end, other than selecting the model with better accuracy as the parent using the bucketing design at the query time, Zoo Manager evicts zoo models with outlier accuracy at runtime. By default, we take the popular Z-score criteria (i.e., model accuracy below the mean by more than two standard deviations) to identify outliers [58]. Moreover, for the same model architecture, it only keeps the model with the best accuracy. We show that ModelKeeper can accelerate training even in the presence of low accuracy models in unfavorable environments (§6.4).

Complexity Analysis The complexity of pairwise model matching is $O(|g_p| \times |g_q|)$,[3] and that of model clustering is $O(M^{2.5})$ for the zoo with M models. Mapper takes linear time to select and transform the parent model. The magnitude of these factors is mostly within O(1K) (§6.1). Our evaluations

[3]We omit the complexity in enumerating tensor parents (i.e., $k \in parent(i)$), since the node degree is orders of magnitude smaller than $|g_p|$.

show that ModelKeeper incurs negligible overhead (§6.3).

5 Implementation

We have implemented a system prototype of ModelKeeper, with around 2K lines of Python code as the frontend library and 1K lines of C++ code as the backend. Our implementation provides user-friendly APIs and supports many popular ML frameworks, such as Microsoft NNI [4], AutoKeras [41], Ray [49], and MLflow [5], with few-lines-of-code plugins.

ModelKeeper Components ModelKeeper coordinator supports distributed deployment across machines. Each coordinator controller processes a single scheduling thread to poll client requests from its queue, and reserves a thread pool for Matcher. Matcher performs pairwise model matching in parallel for each query model, and then Mapper creates a worker thread to transform parent model weights using *numpy* format. Zoo Manager updates the model clustering every 5 minutes, and uses *ortools* library to solve the knapsack problem. The client agent communicates with the coordinator via TCP connections.

Fault Tolerance ModelKeeper uses *Redis* in the coordinator to store the metadata and model weights in a fault-tolerant manner, and this metadata is cached in the memory with small footprints. Changes to the model zoo (e.g., registering new models) follow the write-ahead transaction to the storage. At runtime, the coordinator runs a daemon process to monitor the liveness of the service, which will create a new service process if the existing one crashes. The new process then fetches the latest checkpoint from *Redis* to catch up.

Interfaces We pack interfaces into a Python library. The cluster manager can initiate the coordinator in three lines:

```
from modelkeeper import ModelKeeperCoordinator
keeper_service = ModelKeeperCoordinator(config)
keeper_service.start()
```

Users can initiate the client agent in a few lines (Figure 6).

6 Evaluation

We evaluate the effectiveness of ModelKeeper on three mainstream frameworks for exploratory and general DNN training, using five large-scale CV and NLP model zoos across thousands of models. We summarize the results as follows:

- ModelKeeper saves 23%-77% total amount of training execution needed (i.e., 1.3×-4.3× faster training) than the state-of-the-art without accuracy drop of models (§6.2).
- ModelKeeper outperforms its counterparts by exploiting the parent model with high similarity and better accuracy using different design components (§6.3).
- ModelKeeper improves performance over a wide range of parameters and practical cluster setups in the wild (§6.4).

6.1 Methodology

Cluster setup. We evaluate ModelKeeper on an 80-node cluster (40 GPU nodes and 40 CPU nodes). Each GPU node has a Tesla P100 GPU with 16 GB GPU memory and 16-core CPUs. Since most HuggingFace NLP models exceed our GPU memory capacity, we resort to CPU nodes. Each node has 32-core CPUs and 384 GB of memory. ModelKeeper coordinator runs on a 32-CPU server with 10 Gbps bandwidth.

Workloads. We evaluate ModelKeeper using five widely-used CV/NLP model zoos and realistic workloads (Table 1):

- *NASBench* [24]: an image classification model zoo with thousands of lightweight models for NAS task.
- *AutoKeras Zoo* [41]: a CNN model zoo generated by AutoKeras during the bayesian NAS searching.
- *Imgclsmob* [10]: a popular zoo of state-of-the-art CV models (e.g., DenseNet [36]). Most models are heavyweight.
- *V-Ensemble* [65]: a benchmarking workload for ensemble training, which has hundreds of variants of VGG models.
- *HuggingFace* [2]: a collection of advanced HuggingFace NLP models (e.g., Bert [23]) for next word prediction.

We train Imgclsmob-Small models on CIFAR dataset and ImageNet32 dataset for 32×32 small image inputs, Imgclsmob models on Flowers102 dataset for 224×224 large images, and HuggingFace models on the large WikiText dataset. ImageNet32 is a downsampled 120-category ImageNet dataset (e.g., smaller input size) for efficient computation.

To emulate practical cluster setups, NAS models are generated by the searching algorithm on the fly, and training jobs are submitted following the arrival of Microsoft Trace [37]. The same workload does not contain identical model architectures. ModelKeeper model zoo starts empty for each workload, and jobs contribute (upload) their trained models to the zoo as they complete over time.

Parameters. We follow the default setting specified in each model zoo: (1) *CV models*: the SGD optimizer with minibatch size 64 and initial learning rate 0.01; and (2) *NLP models*: the AdamW optimizer with minibatch size 32 and initial learning rate 8e-5. We use the ReduceLROnPlateau scheduler to decay the learning rate by 0.5 once the training loss stagnates.

Baselines. We compare ModelKeeper to the following:

- *Retiarii [77]*: Microsoft's training framework that relies on the lineage of graph mutation to warm up NAS models.
- *AutoKeras [41]*: An advanced AutoML system based on Keras that applies lineage-based warmup for NAS models.
- *MotherNet [65]*: An ad-hoc ensemble training algorithm that trains a model subnet, which introduces intrusive overhead and implementation, to warm start models.

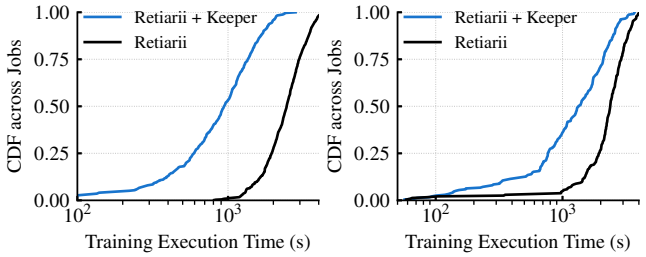
Existing efforts limit to individual NAS/ensemble jobs, while ModelKeeper can support various tasks across jobs and users.

Metrics. We care about the *training execution time* needed to train to converge and the *model convergence accuracy*.

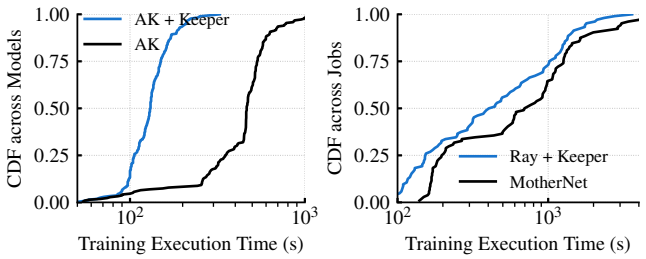
We run with five realistic Microsoft Traces [56], and report the average over 5 runs.

Category	Task	Workload	# of Models	Dataset	Avg. Time Improvement	Avg. Acc. Difference
Exploratory Training	Grid Search NAS	NASBench [24]	1,000	CIFAR-100	2.9×	0.39%
	Evolution NAS				2.4×	0.38%
	AK-Bayesian NAS [41]	AutoKeras Zoo [41]	500		4.3×	0.31%
General Training	Image Classification	Imgclsmob [10]	389	Flowers102 [54]	2.8×	0.23%
		Imgclsmob-Small	179	CIFAR-10	2.1×	0.02%
				CIFAR-100	1.6×	0.18%
				ImageNet32 [19]	1.3×	0.03%
	Ensemble Training	V-Ensemble [65]	104	CIFAR-100	1.7×	0.08%
	Language Modeling	HuggingFace [2]	69	WikiText-103 [47]	1.8×	-0.13 perplexity

Table 1: ***Summary of improvements. ModelKeeper improves training execution time without accuracy drop, by reducing the amount of training needed (i.e., GPU Saving). Accuracy difference is defined by Acc.(Keeper) - Acc.(Baseline), and smaller perplexity is better.***

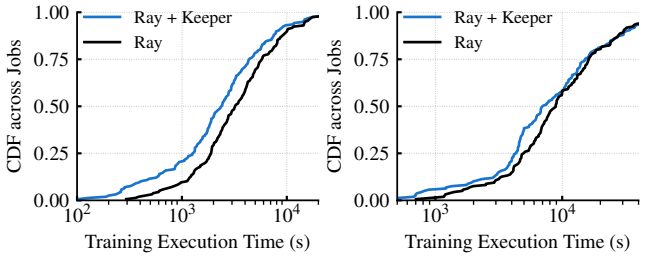


(a) Grid Search NASBench (b) Evolution NASBench

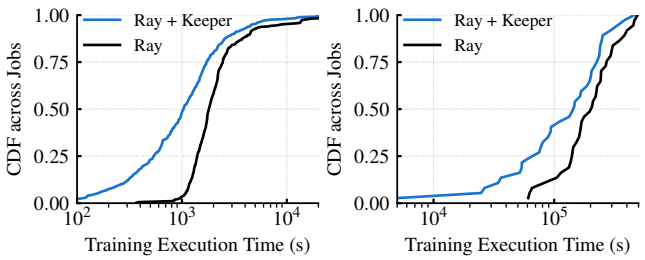


(c) Bayesian Optim. (AutoKeras) (d) Ensemble training

Figure 14: ***ModelKeeper outperforms existing warmup training.***



(a) CIFAR-100 (Imgclsmob-Small) (b) ImageNet (Imgclsmob-Small)



(c) Flowers (Imgclsmob) (d) NLP (HuggingFace)

Figure 15: ***ModelKeeper improves general training tasks.***

6.2 End-to-End Performance

In this section, we evaluate how ModelKeeper (Keeper) is complementary to and benefits today's ML frameworks. Here, we run the NAS task using Microsoft NNI (with Retiarii backend [77]) and AutoKeras, and other training tasks on Ray [49]. Table 1 summarizes the average improvement on each training workload after applying ModelKeeper.

ModelKeeper outperforms existing warmup solutions. ModelKeeper outperforms existing training warmup solutions in Retiarii, AutoKeras, and MotherNet by 1.7×-4.3× (Figure 14), by saving 43.1%-76.7% total amount of training needed. Their inefficiency is due to two primary reasons:

(i) Suboptimal parent model selection: Retiarii and AutoKeras track the lineage of graph mutation and treat the base model in evolution as the parent model. However, as multiple layers can be modified on the base model in searching new models, such rigid parent selection can miss better parent models out of other explored NAS models. Similarly, MotherNet not only requires additional training of the model subnet, but can not repurpose better-trained models on the fly.

(ii) Insufficient weight transformation: Their design, which simply copies the weights from the parent model when two tensors are identical, is lossier than ModelKeeper. For example, inserting randomly initialized prefix tensors can make the copy of subsequent tensors useless.

Meanwhile, they are limited to specific NAS or ensemble training tasks and cannot serve various DNN training jobs on the fly in the cluster wide.

ModelKeeper accelerates ML training for various tasks. Figure 15 and Table 2 report the performance of individual jobs. Compared to training from scratch, we observe that: (i) ModelKeeper achieves 1.3×-4.3× faster training, saving 23%-77% training execution, across a wide range of work-

Workload	Time Improvement			Acc.(Keeper) - Acc.(Baseline)		
	25th	50th	75th	25th	50th	75th
NAS-Grid	1.5×	2.0×	3.1×	0.01%	0.25%	0.42%
NAS-Evol	1.2×	1.6×	3.0×	0.03%	0.19%	0.48%
Flowers102	1.2×	2.1×	3.3×	0.0%	0.16%	0.37%
CIFAR-100	1.1×	1.5×	2.0×	-0.04%	0.08%	0.32%
ImageNet32	1.0×	1.2×	1.6×	-0.07%	0.0%	0.11%
V-Ensemble	1.1×	1.5×	1.9×	0.02%	0.07%	0.65%
HuggingFace	1.2×	1.4×	2.1×	0.2 ppl	-1.3 ppl	-3.87 ppl

Table 2: ***Keeper saves training execution time of individual jobs without accuracy drop. Smaller perplexity (ppl) is better.***

loads. This improvement is more pronounced in a larger zoo because of having more trained models to repurpose. (ii) Improvements on different workloads report a positive correlation with the prevalence of model similarity in that model zoo. Here, ModelKeeper achieves larger improvement on NAS-Bench, which is consistent with the fact that this model zoo owns higher inter-model similarities (Figure 2). (iii) Although ModelKeeper starts from an empty zoo and jobs arrive on the fly, we can still save the training execution for 70%-95% individual jobs (Table 2). We note that the 25th percentile improvement of small-scale model zoos (e.g., Imgclsmob) is inferior to others. This is because not all training models are warmed up due to the cold start of this online setting, and the fact that ModelKeeper will not warm start the model that does not have a similar parent (similarity > 0).

ModelKeeper speeds up training without accuracy drop. Table 1 and Table 2 report that, on average, ModelKeeper can achieve similar (or even slightly better) final model accuracy. Intuitively, ModelKeeper should perform no worse than baseline accuracy, since the rest of the training (e.g., data) remains the same. However, we note that this slightly better model performance is consistent with the observations in ML network morphism [66], which interprets it as the internal regularization ability. Specifically, by transferring from well-trained models, model weights have been placed in a good position in the space, resulting in a more regularized network to reach a better basin of the loss curvature [25, 66]. In contrast, training from scratch can get stuck in local minima.

6.3 Performance Breakdown

In the rest of the evaluations, we refer to the improvement on V-Ensemble as that over training from scratch for brevity.

Breakdown of Components We break down ModelKeeper by disabling Matcher and Mapper respectively: (1) *Keeper w/o Matcher*: remove our Matcher design, and instead resort to a state-of-the-art graph matching strategy [57] to select a parent model with the most pairwise tensor mappings; and (2) *Keeper w/o Mapper*: disable our Mapper design, so only transform the parent model weight if and only if two tensors are identical. Figure 16 reports the improvement of these

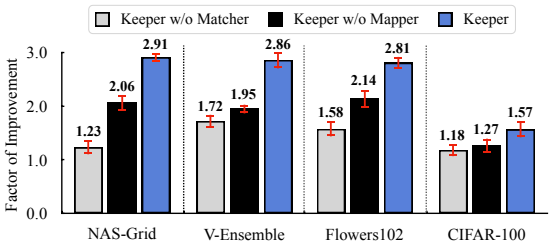


Figure 16: ***Breakdown of Keeper components.***

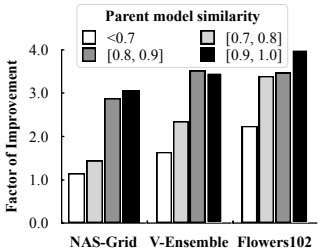


Figure 17: ***Faster training with higher model similarity.***

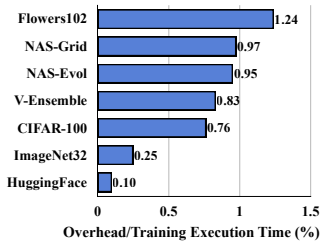


Figure 18: ***Keeper introduces negligible overhead.***

variants. We notice: (i) the classic GED solution, in *Keeper w/o Matcher*, achieves suboptimal performance, since model matching prefers to match prefixes, and partial matching allows better overall similarity. (ii) transforming weights only for identical tensors, in *Keeper w/o Mapper*, is inferior to Keeper information-preserving transformation. (iii) Matcher and Mapper contribute comparable improvements.

Breakdown of Improvement Characteristics Figure 17 reports the average improvement after categorizing training models by their similarity to the parent model. We note that: (1) Keeper tends to achieve better execution saving for models with a higher parent model similarity. This again supports our parent model selection criteria that prioritize models with higher architectural similarity. (2) Improvements of different similarity regimes (e.g., [0.7, 0.8] vs. [0.8, 0.9]) are often distinct, and this becomes vague as similarity over 0.8. Because most layers have been largely warmed up, and deeper layers are too specific to the parent model to be transferable [71].

Overhead Analysis Figure 18 reports Keeper's overhead, i.e., the time taken between initiating the query and starting to train, over the training execution time. We report the average of all jobs, and notice that Keeper introduces less than 1.5% overhead (< 43 s) across all workloads.

6.4 Sensitivity and Ablation Studies

Impact of Low-Accuracy Models As a cluster-wide service, ModelKeeper should be robust to unfavorable settings where the accuracy of user-registered zoo models can be low (e.g., due to insufficient training). We follow the popular early-stop design in ML [44] to simulate unfavorable setups, where model registration takes place when jobs run to at most X minutes. Figure 19 reports the improvement of execution time across different degrees of unfavorable settings. Here, the x-axis value 40% indicates X is set to be the 60th percentile

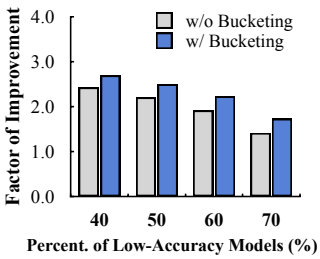


Figure 19: ***Keeper is robust in the presence of poor performance models (NAS-Grid).***

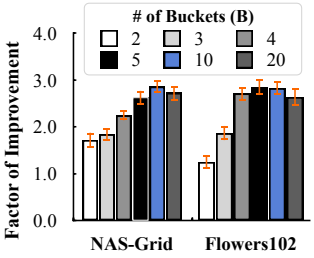


Figure 20: ***Keeper improves training execution time across the different numbers of buckets.***

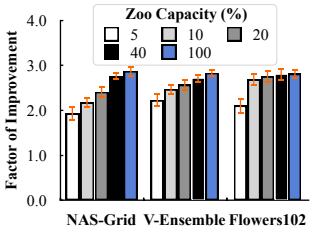


Figure 21: ***Impact of zoo capacity on execution time. Error bars report standard deviation.***

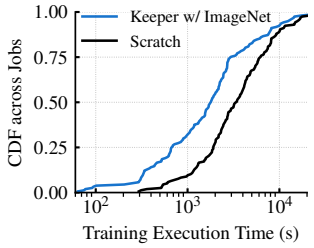


Figure 22: ***Keeper accelerates model training on CIFAR-100 using ImageNet32 model zoo.***

value in execution time distributions, so only 40% zoo models are trained to converge. We observe that: (i) improvement decreases as more zoo models have low accuracy. (ii) Keeper is more robust with our bucketing design as it exploits the similarity-accuracy sweet spot.

Impact of Bucketing Figure 20 reports that ModelKeeper delivers consistent improvement across a wide range of number of buckets B. Meanwhile, we notice that using the most accurate parent model (i.e., $\sim B$=2) or the most architecturally similar parent (i.e., $\sim B$=20) achieves suboptimal improvement, since it respectively undervalues the model similarity and accuracy in selecting the parent model.

Impact of Zoo Capacity Figure 21 reports the average improvement under different zoo capacities. The total size (i.e., 100%) of model zoos in NAS-Grid, V-Ensemble, and Flowers102 are 1.6GB, 17GB, and 31 GB, respectively. We observe that: (i) as expected, the improvement is more pronounced as we allocate more storage to ModelKeeper's model zoo, but (ii) we can still achieve $\sim$2$\times$ improvement under severe capacity limits (e.g., 5% capacity aka < 2GB storage), since Keeper adaptively evicts suboptimal zoo models.

Cross-Dataset Training Warmup Figure 22 reports that ModelKeeper can benefit DNN training across datasets. Here, we warm start the training of Imgclsmob-small models on CIFAR-100 using zoo models from the ImageNet32 workload, and notice 2.5$\times$ faster training on average. This is because front DNN layers capture general input features (e.g., color blobs of images), which are transferable to similar datasets [71]. While picking which dataset as the source for warmup is still an open ML problem [45, 78], ModelKeeper provides systems support for automated warmup transformation across ML tasks and datasets using the given model zoo.

7 Discussion and Future Work

Support for Hyperparameter Tuning ModelKeeper by default automatically searches and transforms the parent model for various training tasks. Meanwhile, the developer can specify which parent model to repurpose using the tag configuration in their request too (Figure 6), while enjoying the automated weights transformation. For example, we may want the same parent model for hyperparameter tuning jobs to eliminate the comparison bias and/or to ensure reproducibility. Moreover, as the training of the query model will be jumpstarted, it would be interesting to investigate how to adapt to better job configurations (e.g., scaling the learning rate in terms of the number of transformed layers [56, 66]) to further improve the training convergence.

Model Sharing in the Wild ModelKeeper repurposes a zoo of trained models to warm start the new training job. These zoo models can be maintained by the cluster provider, and/or contributed by users. For example, AWS SageMaker offers hundreds of pre-trained models for tasks like object detection and natural language processing [8]; HuggingFace Model Hub has gathered $\sim$70K models shared by the community [2]. The former is more managed but expensive to include extensive models and tasks, while the latter has good extensibility but can exhibit great uncertainties (e.g., low-accuracy models). To the best of our knowledge, ModelKeeper moves the first step to *automatically* warm start the cluster-wide model training. However, further investigations on how to democratize it in the wild, such as for privacy and security concerns, are needed. To this end, one possible approach is to develop differential privacy-like solutions [11] (e.g., adding noise to the weights of the contributed models), which naturally leads to an interesting trade-off between privacy and the model quality.

8 Related Work

Deep Learning Frameworks Recent ML efforts have made considerable progress toward efficient inter-job scheduling [28, 49, 56, 74], intra-job computation placement [43, 50], communication optimization [40, 55], specialized execution backend [9, 18, 42], and timely inference [29]. However, they are mostly in-execution optimizations, and/or the total amount of training remains the same. Different from transforming tensors for faster computation (e.g., TASO [38] and PET [64]), ModelKeeper operates on model weights, and acts as a complementary service to accelerate cluster-wide DNN training.

AutoML Systems Retiarii [77] and AutoKeras [41] rely on the lineage of graph mutation to repurpose trained models, whereas they are limited to NAS tasks within individual jobs. Experiment Graph [22] identifies the reusable ML scripts and artifacts in platforms to speed up repeated executions,

so it focuses on the same job execution. As recent AutoML platforms, such as AzureML [59], Amazon SageMaker [1] and MLflow [75], provide a collaborative environment to simplify ML deployments, reusing artifacts can greatly speed up repeated executions (e.g., reuse scripts [22]). ModelKeeper is the first automated training warmup system to accelerate cluster-wide DNN training jobs across users, and improves Retiarii and AutoKeras further (§6.2).

Transfer Learning Transfer learning today mostly transfers the weight of the same model [71], from one source task to another target to alleviate the need for large training data. For a given parent model, network morphism [17, 66] introduces function-preserving transformation to construct child models while preserving the parent information. MotherNet [65] further applies the network morphism to warm start model training, but is limited to ensemble training tasks. ModelKeeper tackles a more challenging scenario for various tasks in the wild, and achieves better performance (§6.2).

Graph Matching Graph matching is one of the NP-hard fundamental problems in graph analysis [61]. To speed up the matching, DAF [31] decomposes the graph into forests. Similarly, AED [57] divides global matching into local matching, and then aggregates the local matching decisions. However, they are insufficient due to the novel properties of DNN graphs, where pairwise matching prefers ordered alignment and allows partial weights transformation. ModelKeeper outperforms them in training speedup and throughput (§6.3).

9 Conclusion

In this paper, we introduce ModelKeeper to enable automated warmup of DNN training jobs at the cluster scale. ModelKeeper manages a collection of already-trained models from different developers and/or frameworks. Before training a model, it selects a high-quality trained parent model and performs structure-aware transformation of parent model weights to warm up the weights of new training models. Our evaluations across thousands of CV/NLP models show that ModelKeeper achieves 1.3×-4.3× faster training completion.

Acknowledgments

Special thanks go to the entire CloudLab team for making ModelKeeper experiments possible. We would also like to thank the anonymous reviewers, our shepherd, Neeraja J. Yadwadkar, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CNS-1909067, CNS-1900665, and CNS-2106184.

References

[1] Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[2] HuggingFace Model Hub. https://huggingface.co/models?sort=downloads.

[3] Kaggle Competition. https://www.kaggle.com/docs/competitions.

[4] Microsoft NNI. https://github.com/microsoft/nni.

[5] MLflow. https://mlflow.org/.

[6] Model Zoo: Discover open source deep learning code and pretrained models. https://modelzoo.co/.

[7] Open Neural Network Exchange (ONNX). https://github.com/onnx/onnx.

[8] Pre-trained machine learning models available in AWS Marketplace. https://aws.amazon.com/marketplace/solutions/machine-learning/pre-trained-models/.

[9] PyTorch. https://pytorch.org/.

[10] Sandbox for training deep learning networks. https://github.com/osmr/imgclsmob.

[11] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, 2016.

[12] Zeyuan Allen-Zhu and Elad Hazan. Variance reduction for faster non-convex optimization. In *ICML*, 2016.

[13] Jordan Ash and Ryan P Adams. On warm-starting neural network training. *NeurIPS*, 33, 2020.

[14] Brian R. Bartoldson, Ari S. Morcos, Adrian Barbu, and Gordon Erlebacher. The generalization-stability tradeoff in neural network pruning. In *NeurIPS*, 2020.

[15] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. abs/2005.14165, 2020.

[17] Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *ICLR*, 2016.

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[19] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets. *CoRR*, abs/1707.08819, 2017.

[20] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC*, 2020.

[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *NSDI*, 2017.

[22] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. Optimizing machine learning workloads in collaborative environments. In *SIGMOD*, 2020.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[24] Xuanyi Dong and Yi Yang. NAS-Bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*, 2020.

[25] Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *AISTAAS*, 2009.

[26] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander J. Smola. Autogluon-tabular: Robust and accurate automl for structured data. *CoRR*, abs/2003.06505, 2020.

[27] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13, 113–129 (2010), 2010.

[28] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, 2019.

[29] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *OSDI*, 2020.

[30] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *NSDI*, 2022.

[31] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*, 2019.

[32] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. NIPS'15, 2015.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[34] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[35] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[36] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[37] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *ATC*, 2019.

[38] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[39] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *ATC*, 2018.

[40] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *OSDI*, 2020.

[41] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *SIGKDD*, 2019.

[42] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Sol: Fast distributed computation over slow networks. In *NSDI*, 2020.

[43] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, 2021.

[44] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 2018.

[45] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I. Jordan. Deep transfer learning with joint adaptation networks. In *ICML*, 2017.

[46] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, 2020.

[47] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016.

[48] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. Ablation studies in artificial neural networks. *CoRR*, abs/1901.08644, 2019.

[49] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, 2018.

[50] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP*, 2019.

[51] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Structured bayesian pruning via log-normal multiplicative noise. In *NeurIPS*, 2017.

[52] James Newling and François Fleuret. K-medoids for k-means seeding. In *NeurIPS*, 2017.

[53] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? *NeurIPS*, 2020.

[54] M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, 2008.

[55] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*, 2019.

[56] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI*, 2021.

[57] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959, 2009. 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007).

[58] Kevin Roth, Yannic Kilcher, and Thomas Hofmann. The odds are odd: A statistical test for detecting adversarial examples. In *ICML*, 2019.

[59] AzureML Team. Azureml: Anatomy of a machine learning service. In *PAPIs*, 2015.

[60] Nilesh Tripuraneni, Michael I. Jordan, and Chi Jin. On the theory of transfer learning: The importance of task diversity. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *NeurIPS*, 2020.

[61] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[62] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: A system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[64] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.

[65] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, 2020.

[66] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *ICML*, 2016.

[67] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *NSDI*, 2022.

[68] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[69] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, 2020.

[70] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. In *MLSys*, 2021.

[71] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *ArXiv*, abs/1411.1792, 2014.

[72] Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *AAAI*, 2019.

[73] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. In *MLSys*, 2020.

[74] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. Fluid: Resource-aware hyperparameter tuning engine. In *MLSys*, 2021.

[75] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[76] Ke Zhang, Miao Sun, Tony X. Han, Xingfang Yuan, Liru Guo, and Tao Liu. Residual networks of residual networks: Multilevel residual networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 28(6):1303–1314, Jun 2018.

[77] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A deep learning exploratory-training framework. In *OSDI*, 2020.

[78] Han Zhao, Remi Tachet des Combes, Kun Zhang, and Geoffrey J. Gordon. On learning invariant representation for domain adaptation. ICML, 2019.

A ModelKeeper Analysis

A.1 Design Criteria

At its core, ModelKeeper performs informed weight initialization for DNN models by repurposing a well-trained model's weights. Intuitively, we note that

- This can be viewed as an instance of existing transfer learning (TL), where we transform the weight of a model on one dataset to train on "another" dataset (i.e., the warmup model has not viewed that dataset before its training takes place) [13]. More subtly, it is a simplified and complementary TL scenario under homogeneous data distribution and features, so existing TL theories can be applied to validate our effectiveness too.
- ModelKeeper transformation is an informed weight initialization, thus a special case of random initialization. As the rest of the training remains the same, the model should be able to reach similar final accuracy when the model converges.

Why ModelKeeper Can Help Convergence? We next present the theoretical analysis of model convergence to show why ModelKeeper can achieve faster convergence.

Corollary A.1. *(Theorem 1 in [72]). Under widely-used DL assumptions (1) Smoothness: loss function $f(\boldsymbol{w})$ is L-Lipschitz smooth; (2) Bounded gradient variances: with constants $G > 0$, $\sigma > 0$, we assume $\mathbb{E}[\|\nabla f(\boldsymbol{w})\|^2] \leq G^2$ and $\mathbb{E}[\|\nabla F(\boldsymbol{w})\|^2 - \nabla f(\boldsymbol{w})\|^2] \leq \sigma^2$; and (3) Unbiased estimation: on mini-batch ξ, we have $\mathbb{E}_{\xi|\boldsymbol{w}}[\nabla f(\boldsymbol{w})] = \nabla F(\boldsymbol{w})$.*

With learning rate γ to $0 < \gamma \leq \frac{1}{L}$, then for iteration T, the model training convergence rate is:

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(\boldsymbol{w}^{t-1})\|^2] \leq \frac{2}{\gamma T}(f(\boldsymbol{w}^0) - f^*) + 4\gamma^2 I^2 G^2 L^2 + \frac{L}{N}\gamma\sigma^2$$

Where N is the number of workers in synchronized data-parallel training, f^* is the optimal training loss.

From Corollary A.1, we can notice that, for the same model, training achieves faster convergence with a smaller initial loss value $f(\mathbf{w}^0)$ in theory (similar to [53, 71]). Indeed, existing gradient variance reduction techniques in the ML community report a similar theory analysis [12]. Here, we empirically show that the initial training loss of the warmup query model, $f(\mathbf{w}^0)$, will start from some basin of loss curvature (e.g., better accuracy in Figure 3 and smaller gradient variance in Figure 4), and theoretically analyze why this enables starting from the loss basin in Appendix A.2.

Admittedly, weight transformation can be lossy (e.g., due to incomplete matching), which breaks the parent model information. We note that capturing the exact convergence comparison herein is extremely challenging, which indeed is a fundamental open problem even in today's transfer learning [25]. Nevertheless, many empirical analyses have reported consistently encouraging improvement [71], and transfer learning is widely-used. Intuitively, for front tensors that enjoy full information-preserving transformation, we can consider them as a prefix subnet, and this subnet holds the same output as the corresponding parent subnet. Therefore, these tensors can still potentially achieve faster convergence according to Corollary A.1.

How to Select Parent Models? In selecting the parent model, ModelKeeper prioritizes the model with (1) *better model accuracy*: this is because parent models with better accuracy enable smaller initial loss $f(\mathbf{w}^0)$, thus allowing better convergence speed (Corollary A.1); and (2) *larger architectural similarity* and *prefix preference*: If we dive to the fundamental of model training, the output activations of a specific model tensor i is $\mathbf{y}^i = f_i(\mathbf{y}^{(i-1)^T}\mathbf{w}_i + \mathbf{b}_i)$. Here, assuming the front $l-1$ tensor are warmup, while $\mathbf{w}_i$ is randomly initialized. The front subnet still enjoys better convergence, so we prefer a model with architectural similarity to maximize this potential. In the forward training propagation, $\mathbf{w}_i$ leads to cascading information loss to subsequent tensors, so we prioritize the match of prefixes to minimize this loss. On the other hand, training front tensors is more difficult but more transferable, because gradient information becomes less informative as it is backpropagated through more subsequent tensors [25], which requires us to match subsequent tensors as many as possible to curb this divergence to the front tensors in backward propagation.

A.2 Information-Preserving Transformation

ModelKeeper employs width and depth operators to perform structure-aware weight transformation, wherein expanding the parent model performs the same to Net2Net [17]. Net2Net theoretically grounds that expanding transformation (e.g., more convolution channels or new convolution tensors) can preserve the parent model information for a wide range of tensors. Specifically, the depth operator tries to deepen a tensor $\mathbf{y}^i = f_i(\mathbf{y}^{(i-1)^T}\mathbf{w}_i + \mathbf{b}_i)$ using two tensors $\mathbf{y}^i = f_i\big(\mathbf{U}^{(i)^T} f_i\big(\mathbf{y}^{(i-1)^T}\mathbf{w}_i + \mathbf{b}_i\big)\big)$, where f_i, $\mathbf{w}_i$, $\mathbf{b}_i$ are the activation function, tensor weights, and bias vectors, respectively. When matrix $\mathbf{U}$ is initialized to an identity matrix, adding $\mathbf{U}$ preserves the same output of its input tensor if f_i is chosen such that $f_i(\mathbf{U}f_i(\mathbf{v})) = f_i(\mathbf{v})$ for all vectors $\mathbf{v}$. This property, f_i, holds for widely-used rectified linear activation in today's DNN models. For example, to insert a new convolution tensor, we should set the convolution kernels to be identity filters. Readers can refer to Net2Net [17] for the theoretical analysis for the width operator. As such, in expanding the parent model, we may preserve the full parent model information.

SHEPHERD: Serving DNNs in the Wild

Hong Zhang
University of Waterloo

Yupeng Tang
Yale University

Anurag Khandelwal
Yale University

Ion Stoica
UC Berkeley

Abstract

Model serving systems observe massive volumes of inference requests for many emerging interactive web services. These systems need to be scalable, guarantee high system goodput and maximize resource utilization across compute units. However, achieving all three goals simultaneously is challenging since inference requests have very tight latency constraints (10–500ms), and production workloads can be extremely unpredictable at such small time granularities.

We present SHEPHERD, a model serving system that achieves all three goals in the face of workload unpredictability. SHEPHERD uses a two-level design that decouples model serving into planning and serving modules. For planning, SHEPHERD exploits the insight that while individual request streams can be highly unpredictable, aggregating request streams into *moderately-sized groups* greatly improves predictability, permitting high resource utilization as well as scalability. For serving, SHEPHERD employs a novel online algorithm that provides guaranteed goodput under workload unpredictability by carefully leveraging preemptions and model-specific batching properties. Evaluation results over production workloads show that SHEPHERD achieves up to 18.1× higher goodput and 1.8× better utilization compared to prior state-of-the-art, while scaling to hundreds of workers.

1 Introduction

Model inference has grown to become a critical component of many interactive applications [1–11]. Facebook, for instance, serves tens of trillions of inference requests per day [12]. Compared to model training, model inference dominates production costs: on AWS, inference accounts for over 90% of the machine learning infrastructure cost [13]. This has driven significant effort in the design of model serving systems to serve inference requests from several applications with deep neural network (DNN) architectures, often using hardware accelerators like graphics processing units (GPUs) to meet tight per-request latency service-level objectives (SLOs), *e.g.*, 50–500ms. These systems typically group requests with the same SLO and target model into separate request streams, and must make two types of scheduling decisions across them to meet system goals. First, they make request serving decisions to maximize *system goodput*, i.e., the number of requests that meet their SLO deadlines per unit time. Second, they make resource provisioning decisions in order to *scale* to a massive number of request streams using large pools of GPUs, while ensuring *high utilization* for the GPU pool for cost-efficiency.

We find that meeting these goals is challenging due to *short-term workload unpredictablity*: our analysis of both synthetic and production workloads (§2.2) indicates that while the average request arrival rates are predictable over longer timescales (i.e., hours), they are bursty and unpredictable at smaller time granularities (i.e., millisconds) that must be considered when scheduling requests to meet their SLO deadlines. As such, existing solutions [3–11] fail to meet one or more of the above goals due to two key reasons.

First, existing systems expose a hard tradeoff between resource utilization and scalability under short-term unpredictability, as they typically employ one of two classes of scheduling policies: (1) *periodic per-stream* policies [3–9], which make scheduling decisions (i.e., resource provisioning, batch sizing, load balancing, etc.) for each stream of requests separately in a periodic manner, and (2) *online global* policies, which make scheduling decisions in an online manner by time-multiplexing the entire pool of resources (*e.g.*, GPUs) across all request streams [10, 11]. On one hand, while the periodic and per-stream nature of scheduling for the former permit scaling to many request streams and compute resources, these systems must over-provision resources to handle unpredictable bursts of requests during each period, resulting in poor resource utilization. On the other hand, online global policies can achieve higher resource utilization by adapting the amount of resources allocated to each stream in an online fashion, but scale poorly with the number of request streams and size of the resource pool due to the increased complexity of online scheduling decisions.

Second, existing approaches are fundamentally unable to provide any *guarantees on system goodput* under unpredictable workloads. We establish several important theoretical results to show why this is fundamental (§5). First, making the optimal scheduling decisions (*e.g.*, executing, deferring or dropping a request) requires future knowledge of request arrival patterns, and even with perfect knowledge, the problem is NP-complete. Second, no online algorithm can achieve goodput that is even within a constant factor of the optimal with perfect knowledge *without using preemption*. Since existing approaches [10] employ simple heuristics without considering preemption, their performance can be arbitrarily worse than the optimal under unpredictable workloads (§2.2).

This raises the question: *Is it possible to design a model*

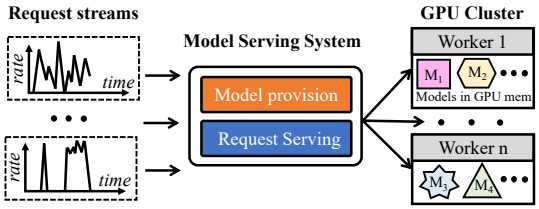


Figure 1: High-level architecture of model serving system

serving system that is scalable, achieves high utilization, and provides guaranteed high goodput under unpredictable serving workloads? In this paper, we answer the above question in affirmative with SHEPHERD, a model serving system resilient to workload unpredictability.

To break the utilization-scalability trade-off exposed by existing solutions, we make an important observation: while individual request streams can be highly unpredictable, aggregating them into *groups* permits accurate resource provisioning. Moreover, our analysis show that even moderately-sized groups comprising hundreds to thousands of streams can already offer reasonable predictability (§3.1). SHEPHERD realizes this insight into a *two-level* design that decouples model serving into a *periodic planning* phase and an *online serving* phase. For the planning phase, we introduce HERD, a planner that periodically classifies inference request streams, DNN models, and GPUs into several *serving groups*. Then based on the planning results, the serving phase employs an online algorithm FLEX to serve requests across streams within each serving group *independently*. HERD solves an ILP to efficiently balance utilization and scalability (§4): on one hand, HERD limits the size of each group, restricting the online scheduling algorithm's decision space to a limited number of streams and GPUs within each group. On the other hand, HERD provisions a sufficient number of streams and GPU workers for each serving group to maximize utilization.

To achieve guaranteed high goodput, we design FLEX (§5), an online scheduling algorithm that leverages preemption and model-specific batching properties. First, we note that while preemption permits correcting for sub-optimal scheduling decisions in the online setting, preempting too often can result in significant amount of wasted work. As such, FLEX carefully weighs the utility of the currently running batch of requests against pending candidate requests to decide whether or not the running batch should be preempted. Second, FLEX leverages a model-specific relationship between the batch size in batched inference execution and its execution latency to determine appropriate batch sizes and the order of execution across request streams. We show that both techniques work in concert to achieve SHEPHERD's goodput guarantee.

We implement SHEPHERD (§6) and evaluate it using a combination of testbed experiments and large-scale emulations with both production and synthetic workloads (§7). Our results show that (1) SHEPHERD achieves up to 18.1× higher goodput and 1.8× higher utilization than periodic per-stream solutions, (2) SHEPHERD achieves up to 5.2× higher goodput compared to heuristic-based online approaches, and (3) SHEPHERD's goodput scales linearly with the number of workers.

2 Background and Motivation

We begin with an overview of model serving systems (§2.1) and short-term workload unpredictability (§2.2).

2.1 System Model and Goals

We focus on Deep Neural Network (DNN) model serving systems [3–11] deployed on GPU clusters (Figure 1). Users issue inference requests, which the system must serve using a specific DNN model on one of its GPU workers within a latency SLO specified for the request, typically 10–500ms [6]. Requests for the same model and with the same latency SLO are typically grouped into a *request stream*, with arbitrary request arrival patterns within each stream. In serving these streams, serving systems can benefit significantly by batching requests on GPUs — on an NVIDIA GTX1080, batching together 32 inference requests improves model serving throughput by 4.7–13.3× for VGG, ResNet and Inception models relative to serving them individually [6]. Taking the above constraints into account, the system makes two scheduling decisions: *model provisioning* decisions to determine which models should be loaded on which and how many GPUs, and *request serving* decisions to determine:

- *batch size:* how many requests to be executed in a batch,
- *batch priority:* which batch should be executed first, and,
- *target GPU:* which GPU to execute the batch on.

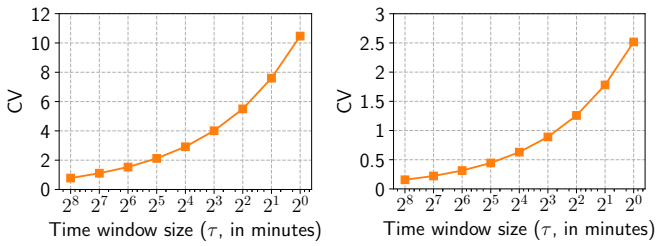
Note that although multiple batches can be executed on one GPU worker concurrently, their execution time becomes nondeterministic due to poor performance isolation on GPUs. As such, most model serving systems [3–11] execute one batch at a time for performance predictability.

The key performance goal for a model serving system is to maximize the system goodput, or the number of served requests that meet their SLO requirements per unit time; requests that fail to meet them often hold no utility for the user. Since serving systems must cater to thousands of requests streams [1, 12], the system should also scale to large clusters with thousands of GPUs in order to serve them. Finally, since inference pipelines comprise the majority of the machine learning infrastructure costs in production settings [13], serving systems should target high resource utilization of the GPU clusters to maximize cost-efficiency.

2.2 Short-term Workload Unpredictability

We find that a key challenge in achieving all three of the goals outlined above is *short-term workload unpredictability*[1] — while the average request arrival rates are predictable over longer timescales (*e.g.*, hours), they can be quite unpredictable at smaller time granularities (*e.g.*, milliseconds) that must be

[1] Unpredictability in request arrival patterns is orthogonal to *performance predictability* demonstrated in prior works [6,10], where the execution latency for inference requests on GPUs is often quite predictable.



(a) Production workload (b) Synthetic workload

Figure 2: **The coefficient of variance (CV) over the number of request in each time window vs window size (τ, in minutes).** The CV value increases dramatically as time window size decreases.

Solutions	Utilization	Scalability	Goodput
Periodic, per-stream policies [3–9]	✗	✓	✗
Online, global policies [10,11]	✓	✗	✗
SHEPHERD	✓	✓	✓

Table 1: Existing solutions under short-term unpredictability

considered to meet per-request SLO deadlines. Next, we show the presence of short-term unpredictability and its impact for both production and synthetic application workloads. Since we are unaware of any publicly available production traces for inference workloads, we use Microsoft's recently released traces for Azure Functions [14] for our production workload, which is noted by recent work to be representative of real-world inference workloads in terms of both diurnal patterns and short-term burstiness [4, 10]. The trace contains the number of function invocations performed at minute granularities across $\sim 46k$ applications over a two-week period. Our synthetic workload simulates $1k$ user request streams as Poisson processes with average arrival rate following an exponential distribution, a commonly-used approach in approximating human-generated invocations [3, 4, 14].

To study workload unpredictability, we divide the entire time period into non-overlapping time windows of size τ, and compute the number of requests $r_{t,s}$ in each time window t for every stream s. We quantify unpredictability in each stream using the coefficient of variance — the ratio of the standard deviation to the mean across $r_{t,s}$. Note that meeting 10–500ms request SLOs requires optimizing scheduling decisions in time window sizes (τ) of hundreds of milliseconds. Figure 2 shows the average coefficient of variance across all streams for different values of τ: for both synthetic and production workloads, coefficient of variance increases drastically as τ decreases. Clearly, while statistical models may be able to estimate average arrival patterns at hours time-scales, the high coefficient of variance at even minute-granularity makes sub-second request arrival patterns nearly impossible to predict.

Under short-term workload unpredictability, existing solutions [3–11] are unable to meet one or more of the three performance goals outlined in §2.1 (Table 1):

Poor resource utilization. Many existing approaches [3–9] make *periodic* provisioning and serving decisions for each user stream *independently*. Within each period (typically a few minutes to hours), inference requests are served following a fixed schedule determined at the beginning of the period. Since scheduling decisions are computed per-stream and updated only periodically, such approaches can scale to many streams over massive pools of GPUs. However, these approaches also tend to over-provisioning GPUs in order to maximize the number of request SLOs met in the presence of short-term burstiness, resulting in poor resource utilization.

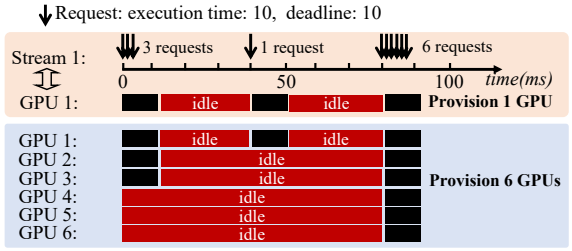
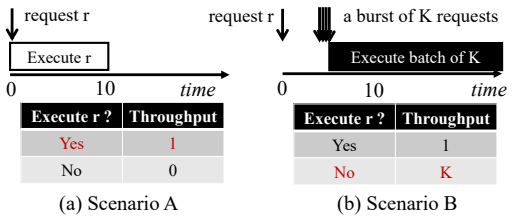


Figure 3: **Periodic per-stream policies observe poor utilization.** Request arrival pattern is shown at the top, with each request's execution time as well as latency SLO being 10ms. Provisioning one GPU (top) based on average load causes 70% of the requests to miss their deadline. Provisioning six GPUs (bottom) allows all requests to meet their SLOs, but reduces resource utilization to 17%.

As a concrete example, Figure 3 shows a user stream with average arrival rate of 1 request every 10ms, with each request's execution time and latency SLO being 10ms as well. The bursty nature of the workload causes three requests to arrive at t=0ms, one at t=40ms and six at t=80ms. Provisioning one GPU for the stream based on the average load would cause 7 out of 10 requests to miss their SLO deadlines — two from the first burst and five from the last. Provisioning six GPUs permits all request latency SLOs to be met, but reduces the resource utilization to 17%, since the GPUs are collectively idle for 500ms out of 600ms cumulative runtime.

Poor scalability. An alternate approach employed by other serving systems is to time-multiplex the GPU cluster across different user streams to achieve better resource utilization [10, 11]. Instead of provisioning and scheduling request for each stream independently and periodically, the system scales the number of GPUs allocated to each stream in an online manner in response to workload fluctuations. While this results in better resource utilization, it also limits system scalability — scheduling decisions to maximize system goodput grow super-linearly in computational complexity with both the number of request streams as well as the number of GPUs they are served over. Our scalability evaluation of Clockwork [10], a recent model serving system that employs such an approach, shows that its goodput does not scale beyond a hundred GPU workers, saturating at $\sim 50k$ requests/second (§7.1). In contrast, real-world inference serving load at Facebook can be as high as 2.3 billion requests/second [12].

Lack of goodput guarantees. Maximizing goodput is challenging under short-term unpredictability. To see why, consider the example in Figure 4, where a request r with an exe-



(a) Scenario A (b) Scenario B

Figure 4: **Example highlighting challenges in online scheduling with short-term unpredictability.** The optimal scheduling decision for request r at time $t = 0$ depends on future arrivals: the performance can be far from optimal depending on the scenario and the scheduling decision to either execute request r or to drop it.

cution duration of 10ms arrives at time $t = 0$. The request has a tight SLO deadline that necessitates its immediate execution for the deadline to be satisfied. The scheduling algorithm has two choices: to schedule the request, or drop it. Unfortunately, the optimal decision to maximize system goodput depends on the future arrival pattern. Specifically, in Scenario A, since no other request arrives during r's execution, the optimal choice is to serve the request. In scenario B, however, where a large burst of K requests with equally tight deadlines arrive at time $t = 5$, the optimal decision is drop r, since it would prevent K request SLOs from being satisfied in favor of one. Note that if the SLO deadline for r was not as tight, the scheduler would have yet another choice to consider — whether or not to defer r's execution so that it may be batched with future requests.

Since future arrival patterns cannot be accurately predicted in the short-term, making the right scheduling choice is inherently hard. Existing solutions rely on simple heuristics, which provides no guarantees on how far the performance could be from the optimal. While they perform well on certain workloads, their performance can be arbitrarily worse than the optimal under unpredictable workloads, similar to the above example. We validate this observation experimentally in §7.2.

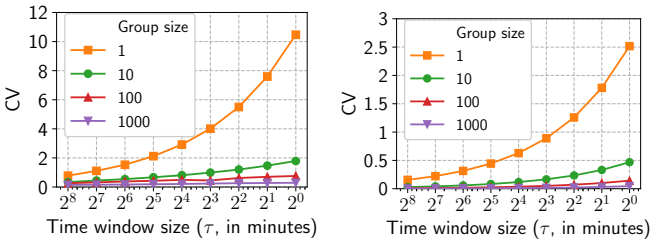
3 SHEPHERD Design

We now outline SHEPHERD's key design elements.

3.1 Overcoming Short-term Unpredictability

We leverage three key observations to overcome the challenges introduced by short-term unpredictability (§2.2):

Group-level predictability and group multiplexing. We observe that while the short-term arrival pattern for individual request streams are hard to predict, the aggregated arrival pattern across a group of request streams tends to be much more predictable. We validate this observation by considering the same workloads in Figure 2, but randomly classifying the request streams into *serving groups* of different sizes and measuring the coefficient of variance *per-group* instead of *per-stream*. Figure 5 shows that increasing the group sizes drastically reduces the coefficient of variance even at smaller window sizes. Note that the networking community has long made similar observations for bursty network flows, where statistical multiplexing can drastically improve utilization



(a) Production workload (b) Synthetic workload

Figure 5: **Coefficient of variance (CV) *for groups of streams* vs window size (τ, in minutes).** The CV increase with decreasing window sizes is much slower for larger group sizes.

by dynamically sharing a network link across network flows based on their instantaneous demands [15–18].

However, unlike multiplexing a network link across a few flows, multiplexing thousands of GPU workers across tens of thousands of SLO-bound request streams in real time presents a significant scalability challenge. To address it, we observe that even at moderate group sizes (100–1000), the per-group coefficient of variance is small enough to make its arrival pattern highly predictable (Figure 5). This motives a *group multiplexing* approach that first partitions the GPU cluster and request streams into moderately-sized serving groups (§4), then applies statistical multiplexing per-group to perform online scheduling (§5). This approach offers a means to break the tradeoff between resource utilization and scalability faced by existing systems: moderately sized groups are predictable enough even in the short-term to accurately provision their resources for high resource utilization. At the same time, restricting the online scheduling algorithm's decision space to streams and GPUs assigned to each group drastically limits its computational complexity, allowing the system to scale to much larger number of request streams and GPUs.

Preemption to correct for scheduling errors. As noted in the example from Figure 4, the optimal scheduling decision often depends on future arrival patterns, which can be hard to predict. As such, any non-clairvoyant online algorithm is bound to occasionally make sub-optimal scheduling decisions. We find that the ability to correct such decisions when its sub-optimality becomes apparent via preemptions is *necessary* for achieving performance guarantees for an online scheduling algorithm. For instance, in the example of Figure 4, a solution can correct for a sub-optimal scheduling decision in both scenarios by simply preempting r if a burst of requests arrives later. Preemptions in online scheduling algorithms are not a new concept; they have been used in a variety of scheduling contexts [19, 20] to achieve bounds on the algorithm's *competitive ratio* — the ratio between its performance and that of an optimal offline algorithm. Leveraging insights from recent work on context switching for DNN training on GPUs [21] allows us to realize preemptions efficiently for inference workloads (§6), and combining it with model-specific batch-latency relationships (described next) permits bounding

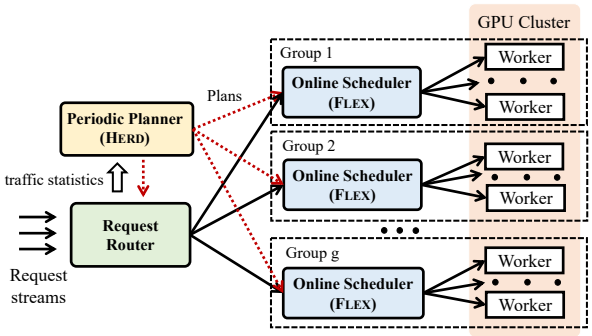


Figure 6: Overview of SHEPHERD design

the competitive ratio for online model serving.

Model-specific batch-latency relationships. Empirical measurements in prior work [6] indicate that a simple linear model can accurately describe the execution latency for varying request batch sizes in model serving workloads. In particular, for a batch B of size $|B|$ being executed on a model m, the execution latency $\ell_m(B)$ is given by:

$$\ell_m(B) = \alpha_m \cdot |B| + \beta_m \tag{1}$$

where β_m is the baseline execution latency for executing an empty batch on the model, while α_m is the latency for each additional request in the batch.

We find exploiting this relationship helps make better scheduling *and* stream grouping decisions. First, larger batches help amortize the fixed cost β_m and achieve higher throughput, but too large a batch may miss the SLO deadline altogether. As such, making scheduling and preemption decisions that leverage the batch-latency relationship to prioritize appropriately large batches that are likely to meet their deadline, permit better performance guarantees for the online scheduling algorithm. Second, when scheduling requests across streams in a serving group of certain models, we find that the online algorithm can achieve better performance guarantees if the models have similar α and β values (§5.2).

We next describe how SHEPHERD incorporates all of these insights into an end-to-end design.

3.2 Design Overview

SHEPHERD leverages group-level predictability in a two-level design that comprises a periodic planning and an online serving component. At a high-level, the periodic planning component leverages long-term load statistics to partition the entire GPU cluster into several serving groups, and determines how models and request streams querying them are mapped to these groups to optimize both resource utilization and system scalability. The online serving component, on the other hand, schedules requests from streams in each serving group across the group's allocated GPUs, and ensures that its goodput is always within a constant factor of the optimal schedule.

SHEPHERD's architecture (Figure 6) comprises four key components: a planner (HERD), a request router, a scheduler (FLEX) per serving group and multiple GPU workers. HERD executes periodic planning, and informs each GPU worker which serving group it belongs to and which models it must serve. HERD also assigns a group-level scheduler to each serving group — the total number of group-level schedulers can be scaled based on the number of models being served by the system and the aggregate load across them. The request router forwards client inference requests to group-level schedulers based on their target model, and collects statistics regarding their arrival patterns that HERD employs to compute group-level mappings. The group-level schedulers, in turn, execute our online scheduling algorithm, FLEX, to schedule inference requests across GPU workers in their own serving group.

HERD (§4). While even random assignment of models and GPU workers to serving groups can achieve decent workload predictability (§3.1), achieving high utilization and guaranteed goodput requires considering a number of additional constraints. To this end, HERD formulates this assignment problem as an Integer Linear Program (ILP) incorporating all such constraints. In particular, as noted in §3.1, colocating models with similar α, β values (Eq. 1) in the same serving group yields better goodput guarantees in FLEX. Consequently, HERD also incorporates model-affinity — a measure of similarity across α, β values — in its ILP.

FLEX (§5). FLEX's goal is to provide guaranteed high goodput for each group under short-term unpredictability. To this end, we answer three key theoretical and practical questions:

- **What performance guarantees are possible?** We first establish two impossibility results. We show that determining an optimal solution is NP-hard, even in the offline setting. In the online setting, we show that no online algorithm can achieve performance *competitive* with the optimal offline solution without using preemption. Since prior model serving systems do not employ preemption, they are fundamentally unable to provide any performance guarantees.
- **What performance guarantees can FLEX provide?** FLEX ensures that for each serving group, the aggregated goodput achieved is guaranteed to be at most $12.62 \cdot K\times$ worse than the optimal offline schedule with complete knowledge of the future. K is a model-affinity parameter that reduces to one if all models in the serving group have the same α and β, and increases if they diverge (§5.2).
- **How does FLEX achieve this guarantee?** FLEX leverages two key insights outlined in §3.1: preemption to correct for scheduling errors, and model-specific batch-latency relationships. First, preempting a scheduled batch requires carefully weighing the utility brought by the scheduled batch of requests against the utility of the new batch to be scheduled — the threshold beyond which preemption is performed significantly impacts the performance bound FLEX can achieve. Second, FLEX leverages the model-specific relationship in Eq. 1 to determine appropriate batch sizes

Decision variables	Definition
$x_{ij} \in \{0,1\}$	Is stream i mapped to group j?
$y_{cj} \in \{0,1\}$	Is affinity-set c mapped to group j?
$z_{kj} \in \{0,1\}$	Is model k mapped to group j?
$size_j \in \mathcal{N}+$	# of GPUs allocated to group j
Input parameters	**Definition**
mem	GPU memory capacity
$\overline{G}$	Scalability limit for # of GPUs per group
N	# of GPUs in cluster
$h_{ki} \in \{0,1\}$	Does stream i use model k?
$q_{ck} \in \{0,1\}$	Does affinity-set c include model k?
Optimization goal	**Definition**
$bt(i)$	The burst tolerance metric for stream i

Table 2: Variables used in HERD's ILP.

and their order of execution across request streams.

4 Periodic Planner: HERD

HERD operates in two steps. It first determines the number of GPUs n_i that would be needed to sustain the average load $rate_i$ for each request stream separately. To do so, HERD empirically measures the maximum goodput T_i each stream i can achieve on a single GPU, and uses it to compute n_i as $\frac{rate_i}{T_i}$. It uses n_i to define a new *burst tolerance* metric (bt) that captures the increase in load that the stream can tolerate if assigned to a particular serving group relative to the average-load based assignment of GPUs. More formally,

$$bt(i) = \frac{\text{\# GPUs } i \text{ can use for its peak load}}{\text{\# GPUs } i \text{ needs for its average load}} = \sum_j \frac{size_j \cdot x_{ij}}{n_i}$$

where x_{ij} is 1 if stream i is assigned to group j (0 otherwise), and $size_j$ is the number of GPUs assigned to group j.

Second, HERD uses an Integer Linear Program (ILP) to combine streams into serving groups to *maximize the minimum burst tolerance across all streams*; this captures the goal of ensuring every stream can tolerate as heavy a burst as possible, subject to a certain set of constraints:

(a) **Cluster-size limit** ensures that the total number of GPUs assigned across all serving groups is no larger than the cluster-size N (in number of GPUs).

(b) **Group-worker limit** ensures that the total number of GPUs $size_j$ assigned to each group j does not exceed the maximum scalability limit $\overline{G}$ of the online algorithm.

(c) **GPU-memory limit** ensures that the sum of model sizes assigned a serving group j does not exceed the GPU memory capacity mem.

(d) **Group surjectivity** ensures that every stream i is assigned to a single group j, and only if its associated model is also assigned to group j.

(e) **Affinity-set surjectivty** ensures that models assigned to the same group j have similar α, β values (as defined in Eq. 1) to ensure better performance guarantees in FLEX.

We capture the divergence in model α, β values as K (defined in §5), and pre-compute affinity-sets $c_1, c_2, \ldots$ as a partitioning of models such that K between any two models in an affinity set is $\leq \overline{K}$; this simplifies our ILP constraint to only picking models from the same cluster.

Our ILP is presented below, with variables listed in Table 2:

$$\textbf{maximize} \quad \min_i \{bt(i)\} \tag{2}$$

$$\textbf{s.t.} \quad \sum_j size_j \leq N, \qquad \text{((a) Cluster-size limit)}$$

$$size_j \leq \overline{G}, \quad \forall j \qquad \text{((b) Group-worker limit)}$$

$$\sum_k z_{kj} \cdot |m_k| \leq mem, \quad \forall j \qquad \text{((c) Memory limit)}$$

$$\left.\begin{aligned} &\sum_j x_{ij} = 1, \quad \forall i \\ &h_{ki} \cdot x_{ij} \leq z_{kj}, \quad \forall i,j,k \end{aligned}\right\} \qquad \text{((d) Group surjectivity)}$$

$$\left.\begin{aligned} &\sum_c y_{cj} = 1, \quad \forall j \\ &q_{ck} \cdot z_{kj} \leq y_{cj}, \quad \forall i,j,k \end{aligned}\right\} \qquad \text{((e) Affinity-set surjectivity)}$$

Note that the above formulation is not linear due to the non-linear optimization goal, which contains: (1) a max-min term, and, (2) a product between binary and non-negative variables ($x_{ij} \cdot size_j$). However, both can be linearized using standard techniques [22] — we omit the linearized ILP for brevity. Similar to prior work [6], HERD ensures that all models to be served by a worker in the subsequent online serving phase are present in GPU memory, with some memory set aside for the operation of the online algorithm, FLEX. We discuss additional challenges due to memory constraints in §8.

HERD complexity and periodicity. Since solving HERD's ILP is NP-hard, and we must scale to millions of streams and thousands of workers, we first aggregate streams using the same model into a single "model-stream", then apply the ILP to optimize the burst tolerance metric across the model-streams. The burst tolerance metric of the model-stream the lower bound of the burst tolerance metric for each stream in it. Note that different streams in the model-stream may have different SLOs, but this will not affect the correctness of our ILP, since none of the constraints (a) – (e) depend on per-stream SLO. Instead, FLEX incorporates the impact of SLOs across different streams during online serving.

Also, note that we only need to ensure that the ILP solver is much faster than HERD's periodicity, which, in turn, depends on how frequently the workload characteristics change enough to require recomputing group assignments. Fortunately, our analysis of Microsoft's Azure Function trace [14] shows that the workloads within moderately-sized serving groups remain stable for tens of minutes or more, while our solver can compute a plan within a few seconds (§7.3).

Input variables	Definition
$S = \{r_1, r_2, ...\}$	A request stream from one application.
a_r, d_r, m_r	Arrival time, deadline, model for request r.
$a(B), d(B), m(B)$	Arrival time, deadline and model for batch B.
$\mathbb{B} = \{B_1, B_2, ...\}$	Set of all possible batches.
Decision variables	**Definition**
$I(B,t,n) \in \{0,1\}$	Is batch B is executed at time t on GPU n?

Table 3: Notations for online batch scheduling.

5 Online Serving Algorithm: FLEX

We first formulate the online serving problem (§5.1) and then present the FLEX algorithm to provide guaranteed goodput under short-term unpredictability (§5.2).

5.1 Problem Formulation

Our online serving setting focuses on scheduling inference requests across models and GPUs assigned to a single serving group. Requests within each stream query the same model with the same latency SLO. Each request r has an arrival time a_r, deadline d_r and queries model m_r. Requests are served in batches; for a batch B, arrival time $a(B)$ is the arrival time of the most recent request in B, and deadline $d(B)$ is the earliest deadline of all requests in B. Let $\mathbb{B}$ be the set of all possible batches of requests; the online serving algorithm decides whether to execute batch $B \in \mathbb{B}$ at time t on GPU n, which we capture as the decision variable $I(B,t,n) \in \{0,1\}$. The goal of online serving is to maximize the overall goodput: the number of requests that meet their SLOs per second. Table 3 summarizes the notations for our problem formulation.

Optimal offline serving algorithm. We find that the offline serving problem where the scheduler has access to the complete future can be formulated as the following Zero-one Integer Linear Program (ZILP):

$$\textbf{maximize} \sum_t \sum_n \sum_{B\in\mathbb{B}} |B| \cdot I(B,t,n) \tag{3}$$

$$\textbf{s.t.} \sum_t \sum_n \sum_{\{B|r\in B\}} I(B,t,n) \le 1, \quad \forall r \tag{a}$$

$$\sum_{B\in\mathbb{B}} \sum_{\{t'|t'\le t\le t'+\ell_{m_B}(B)\}} I(B,t',n) \le 1, \quad \forall t,n \tag{b}$$

$$a(B)\cdot I(B,t,n) \le t, \quad \forall B,t,n \tag{c}$$

$$(\ell_{m_B}(B)+t)\cdot I(B,t,n) \le d(B), \quad \forall B,t,n \tag{d}$$

$$I(B,t,n) \in \{0,1\}, \quad \forall B,t,n \tag{e}$$

Intuitively, the ZILP maximizes the total number of requests that meet their latency SLOs across all selected batches ($I(B,t,n) = 1$), which in turn maximizes the total goodput. The ZILP constraints correspond to:

(a) Each request can be executed in at most one batch,

(b) A GPU can only execute one batch at a time,

(c) No selected batch can start before its arrival time,

Algorithm 1 FLEX Algorithm

```
1: Initialize:
2: for each model m do
3:     Q_m ← Priority queue of m's requests sorted by deadlines.
4: Event: On completion of a batch on any GPU n:
5: B_{g,n} ←BATCHGEN(n)  # Largest feasible batch across all Q_m
6: Execute B_{g,n} and dequeue requests in B_{g,n} from model queue
7: for each GPU n do
8:     B_{g,n} ←BATCHGEN(n)  # Update candidate batch
9: Event: On arrival of request r:
10: Enqueue r to corresponding queue
11: for each GPU n do
12:     B_{c,n} ← The batch currently being executed on GPU n
13:     B_{g,n} ←BATCHGEN(n)
14:     if B_c = ∅ then
15:         Execute B_{g,n} and dequeue requests in B_{g,n}
16:     else if |B_{g,n}| ≥ λ × |B_{c,n}| then  # Preemption rule
17:         Preempt B_{c,n}
18:         Execute B_{g,n} and dequeue requests in B_{g,n}
19:         Treat requests in B_{c,n} as new arrivals (go to Line 11)
```

(d) Every selected batch must finish before its deadline, and

(e) The decision variable $I(B,t,n)$ must either be 1 or 0.

Clearly, the optimal solution to the above ZILP is also the optimal offline schedule. Obtaining such an optimal is unrealistic — not only is it impractical to have access to the complete future (or even a reasonable prediction of it, §2), computing the optimal solution to the ZILP is NP-hard [23].

Achievable guarantees. However, the optimal offline schedule provides us with a baseline of the best schedule possible, and permits us to reason about how close an online algorithm can get to such a solution. More formally, the performance guarantee an online algorithm can achieve is typically captured by the *competitive ratio*: the worst-case ratio of the ZILP's goodput to the online algorithm's goodput over all possible inputs. Note that our focus is on online request serving decisions, so we assume both algorithms have the same resources provisioned to them. We establish the following important result regarding the competitive ratio:

Theorem 5.1 *No non-preemptive, deterministic algorithm can achieve a bounded competitive ratio for online serving.*

We defer the proof to Appendix A, but note that since existing online serving algorithms [6, 10, 11] are non-preemptive, they are incapable of achieving a bounded competitive ratio.

5.2 FLEX Algorithm

Algorithm 1 presents our FLEX algorithm that achieves a bounded competitive ratio for online serving. During initialization, FLEX creates a priority queue Q_m for each model m, which holds requests sorted by tightness of their deadlines. The algorithm reacts to two key events: (1) *completion event* of a batch on any GPU, and, (2) *arrival event* of a new request.

For a completion event, FLEX simply generates a new batch B_g and executes it; all requests in B_g are dequeued from corresponding model queue Q_m. To generate the new batch, FLEX finds the largest feasible batch across all queues, such that all requests in the batch can meet their latency SLOs.

For an arrival event, FLEX generates a candidate batch for each GPU as outlined above, and compares it with the currently running batch. If the generated batch is λ times larger than currently running batch, the current batch is preempted. If preemption occurs, requests in preempted batch that can still meet their SLOs are re-enqueued to their corresponding priority queues. The re-enqueued requests will be treated as newly arrived requests so they can be scheduled again.

We now dive deeper into salient features of the algorithm.

Choice of λ. The preemption threshold λ plays a crucial role in bounding FLEX's competitive ratio. A conservative preemption policy with larger λ can result in a poor competitive ratio, while an aggressive preemption policy with smaller λ can waste GPU resources, since the preempted work does not contribute to system goodput. As such, we express the competitive ratio in terms of λ, and formulate the problem of finding the optimal competitive ratio as an optimization problem. Solving this problem yields the optimal value of $\lambda \approx 3.03$ (Theorem 5.2). Note that while a worker may experience cascading preemptions if batches keep arriving with sizes $\lambda\times$ than the currently executing batch, our choice of λ ensures that the total wasted work is always much less than the additional useful work performed post-preemption. In practice, the effect of cascading preemptions is bounded due to our maximum batch size limit (128 by default). We defer the description of our preemption implementation to §6.

Prioritizing batches for a single model. Online job scheduling algorithms [19, 20, 24–27] tend to consider one of two key metrics as optimization goals: a job's *value*, and its *value density*. In the online model serving context, the value of a job (batch) corresponds to the number of requests it contains (i.e., its batch size), while the value density corresponds to its contribution to system goodput (i.e., $\frac{\text{batch size}}{\text{batch latency}}$). Traditional online job scheduling algorithms often fail to achieve a bounded competitive ratio since optimizing these two goals are often at odds with each other, i.e., optimizing total value density comes at the cost of optimizing total value across jobs, and vice versa. Fortunately, Eq. 1 establishes a linear relationship between value density and value for batches of inference requests: for a single model, *larger batches always contribute more to system goodput*. As such, our preemption and batch generation criteria always favor larger batches to maximize total value and value density simultaneously, enabling FLEX to achieve a bounded competitive ratio. In contrast, prior slack-based prioritization schemes (*e.g.*, tightest deadline first [10]) are unable to provide such guarantees. In fact, our evaluation (§7) shows that prioritizing larger batches over those with tigher deadlines leads to higher goodput under high load.

Extending FLEX to multiple models. While the above prioritization scheme is straightforward when a single model is involved, extending FLEX's competitive ratio analysis to a multi-model scenario is challenging, since the linear relationship between batch value and value density no longer holds *across* models. However, the batch-latency relationship in Eq. 1 still allows us to bound the batch value and value density across models using the model-specific parameters α and β. More precisely, we define an affinity metric $A(m_i, m_j)$ between two models m_i and m_j as:

$$A(m_i,m_j) = \begin{cases} \frac{\alpha_i+\beta_i}{\alpha_j}, & \text{if } \alpha_j+\beta_j-\beta_i \le 0 \\ \min(\frac{\alpha_i+\beta_i}{\alpha_j}, \max(\frac{\alpha_i}{\alpha_j+\beta_j-\beta_i}, \frac{\alpha_i}{\alpha_j})), & \text{otherwise} \end{cases}$$

where $\alpha_i, \alpha_j, \beta_i$ and β_i are the model-specific parameters for models m_i and m_j respectively. While its specific formulation is devised to establish FLEX's competitive ratio (Theorem 5.2), we note that $A(m_i, m_j)$ is close to 1 if m_i and m_j have similar α and β, and deviates from 1 as the α and β values for the models diverge. For a set of models $\mathbf{M}$, we show that the competitive ratio is a multiple of K, the largest affinity value $A(m_i, m_j)$ across all pairs of models (m_i, m_j) in $\mathbf{M}$, i.e.,

$$K = \max_{i,j \in \mathbf{M}} A(m_i, m_j) \tag{4}$$

FLEX properties. Our analysis in Appendix B shows that:

Theorem 5.2 *Algorithm 1 is* $12.62 \cdot K$*-competitive with preemption threshold* $\lambda \approx 3.03$*, with K defined in Eq. 4.*

We note that FLEX is the first algorithm that achieves guaranteed performance for online model serving to the best of our knowledge. We validate FLEX's performance empirically over a wide range of representative workloads in §7. Finally, while we defer the complexity analysis to Appendix C the following result establishes FLEX's complexity:

Theorem 5.3 FLEX *has a worst-case complexity of O(G), where G is the number of GPUs in the serving group.*

6 SHEPHERD Implementation

Our SHEPHERD implementation follows the architecture described in Figure 6. The periodic planner (HERD), request router and online scheduler are implemented as C++ processes, while the GPU workers support configurable model execution runtimes like PyTorch [28] and Apache TVM [29].

Supporting preemptions. While recent hardware-based preemptions on newer GPUs [31] may enable better performance, we opt for software-based preemptions adapted from Pipeswitch [21] in SHEPHERD due to its general applicability to commodity GPUs. Pipeswitch supports preemption of DNN training tasks by inserting exit points between the training phases of different DNN layers: when a preemption is requested, the execution of the current training task can be terminated at the next exit point. Since PipeSwitch currently supports preemptions for PyTorch only, we use the PyTorch

Model	α (ms)	β (ms)	# Exit points
ResNet18 (RN18)	0.22	3.74	40
ResNet34 (RN34)	0.38	5.78	46
ResNet50 (RN50)	0.75	7.96	46
ResNet101 (RN101)	1.25	13.57	39
ResNet152 (RN152)	1.77	18.98	84
ResNeSt50 (RS50)	1.18	15.39	78
ResNeSt101 (RS101)	1.91	29.21	57
ResNeSt200 (RS200)	3.35	45.43	96
ResNeSt269 (RS269)	4.37	74.20	128
DenseNet121 (DN121)	0.69	19.96	129
DenseNet161 (DN161)	1.74	23.10	171
DenseNet169 (DN169)	0.83	27.47	120
DenseNet201 (DN201)	1.12	32.33	142
GoogLeNet (GN)	0.25	8.41	44
Inception v3 (I3)	0.96	11.77	122
R-CNN (RCNN)	2.59	14.90	51
BERT (BERT)	40.98	5.67	43

Table 4: DNN Models evaluated in SHEPHERD. BERT [30] is a popular NLP model, while the rest are popular CV models from six different model families.

runtime by default in our implementation, although the same approach could be implemented for Apache TVM as well.

Adapting the preemption approach from training to inference pipelines introduces a key challenge: while the overhead for preemption is not a major concern for long-running model training tasks, it is quite crucial to minimize preemption overheads for model inference. On one hand, adding too few exit points to an inference task introduces unacceptable *preemption delay* — the time between from the preemption being requested and actually being completed — since the preempted task may still execute for tens of milliseconds before the reaching next exit point. On the other hand, adding too many exit points slows down the normal execution of inference tasks, as each exit point introduces non-negligible *execution delay*. To better navigate the trade-off, we evaluate the preemption and execution delay overheads with different number of exit points for different DNN models via comprehensive profiling, and determine the optimal number of exit points for each individual model (§7.3). Table 4 shows the DNN models used in SHEPHERD, with their α, β values (Eq. 1) and the number of exit points. Note that adding exit points incurs a one-time offline profiling cost during model registration; this can be implemented as a part of the DNN framework, making it completely transparent to users.

7 Evaluation

We evaluate SHEPHERD to answer the following questions:

- How does SHEPHERD compare against state-of-the-art schemes for real-world workloads? (§7.1)
- How does each design component in SHEPHERD contribute to its performance gains? (§7.2)
- What overheads do SHEPHERD's preemption and periodic planning components introduce? (§7.3)

Setup. All our experiments were run on Amazon EC2. For GPU workers, our testbed experiments use 12 p3.2xlarge instances each with 8 vCPUs, 61GB RAM, and one NVIDIA Tesla v100 GPU with 16GB memory, while our large-scale emulations use m4.16xlarge instances with 64 vCPUs and 256GB RAM. The request router, periodic planner, and online schedulers are deployed on separate m4.16xlarge instances.

Metrics. We focus on goodput, utilization and scalability as our key metrics. Goodput and utilization values are averaged over 5 runs, while scalability is measured as the increase in system goodput on increasing the number of workers.

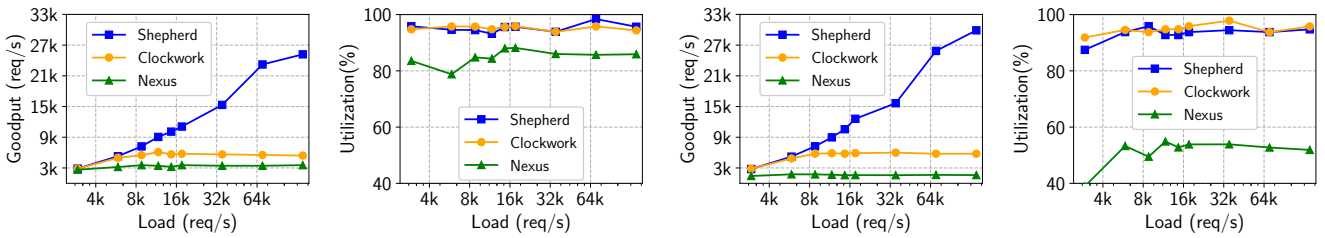
Compared schemes. We compare SHEPHERD against Clockwork [10] and Nexus [6]. Clockwork is representative of online global scheduling policies, while Nexus is a representative of the periodic per-stream approach (§1). We implement all evaluated policies in our SHEPHERD prototype and use a PyTorch-based runtime to ensure that the performance differences are solely due to the scheduling decisions rather than choices in system implementation or the underlying runtime.

For Nexus, we set the reconfiguration period to 60 seconds as recommended in [6]. Moreover, since Nexus is designed for predictable workloads, we adapt their algorithm to provision for the peak demand in every 60-second window of the workload to ensure it can sustain the provided load. For SHEPHERD, we set the GPU group-worker limit to 12, since we found it to be large enough to ensure workload predictability (due to a large enough group size) while being well within our scheduler's scalability limit. The GPU memory limit for p3.2xlarge instances is large enough to fit all 13 DNN models. Finally, we place all the models in a single affinity-set.

DNN Models. We evaluate SHEPHERD with 17 DNN models widely used for model inference (Table 4), taken from PyTorch Hub [32]. For Clockwork and Nexus, we use models without any exit points (needed for preemption in SHEPHERD, §6) to ensure they do not suffer any performance penalties for execution delays. We ensure the models remain in GPU memory for the duration of all our experiments to eliminate performance impacts of loading models into GPU memory.

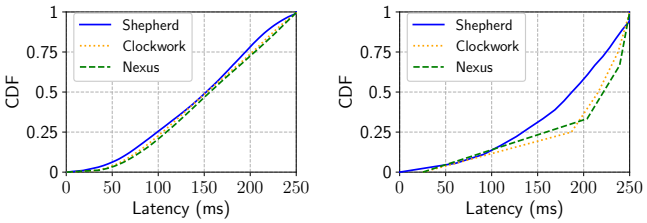
Workloads. Similar to prior work [10], we use the Microsoft's publicly-released production traces from Azure Functions (MAF) [14] as a representative production model serving workload. MAF interleaves a wide range of workloads, including heavy-sustained, low-utilization, bursty and fluctuating workloads. For our 13 profiled DNN models, we assign the 46,000 streams from MAF to models in a round-robin manner, and configure all streams with a default SLO of 250ms[2], unless otherwise specified. The MAF trace only contains the aggregated number of requests per one-minute interval for each request stream. Therefore, we generate two request ar-

[2]We use a relatively relaxed SLO compared to [10] since the PyTorch runtime used in our implementation observes longer inference latencies compared to the TVM runtime used in [10].



(a) Goodput (MAF-stable) (b) Utilization (MAF-stable) (c) Goodput (MAF-bursty) (d) Utilization (MAF-bursty)

Figure 7: **Performance variation with load.** Under high load, SHEPHERD achieves 4.6 (5.2)× and 7.1 (18.1)× higher goodput than Clockwork and Nexus under the MAF-stable (MAF-bursty) workload, respectively. SHEPHERD and Clockwork achieve high system utilization while Nexus's utilization remains under 89% (55%) across different arrival rates under the MAF-stable (MAF-bursty) workload.



(a) Low load (3*k* req/s) (b) High load (144*k* req/s)

Figure 8: **Latency CDFs for MAF-stable workload with 250ms SLO.** The latency CDF is presented for the set of requests admitted by each approach. At high load, large portions of Clockwork and Nexus request latencies are close to the SLO, while SHEPHERD's request latencies are distributed more evenly. See §7.1 for details.

rival patterns within each one-minute interval: (1) a Poisson process to model stable workloads, similar to [10] ("MAF-stable"), and (2) a more bursty Markov-modulated Poisson process (MMPP) similar to [9] ("MAF-bursty").

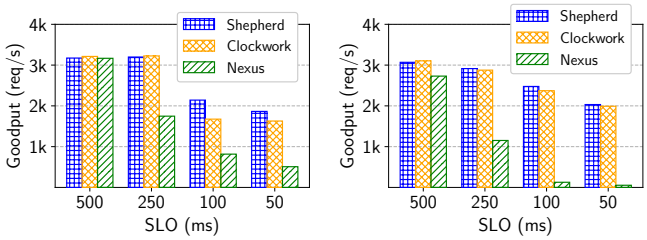
7.1 SHEPHERD in the Wild

We first evaluate the compared systems for real-world workloads on a testbed comprising 12 GPU workers and large-scale emulations that mimic work done by a GPU on CPU cores.

Performance variation with load (Figure 7). For the MAF-stable workload, with a low request arrival rate (*e.g.*, at ∼ 3*k* requests/second), all systems can meet the SLO deadlines for most requests in the workload. As such, both SHEPHERD and Clockwork achieve high system utilization (over 95%) and high goodput. At higher loads, while both systems are consistently busy serving requests (resulting in high utilization) neither SHEPHERD nor Clockwork can satisfy all request deadlines; however, since Clockwork prioritizes requests based on how close their deadline is, it greedily schedules many small batches of requests with tight deadlines, resulting in a reduced goodput. In contrast, SHEPHERD always prioritizes execution of larger batches, while the use of preemption ensures that large batches never get blocked by small batches scheduled before them. SHEPHERD can therefore efficiently utilize limited GPU resources to maximize goodput under high load, and while Clockwork's goodput starts to saturate beyond a load of 6*k* requests/second, SHEPHERD's goodput keeps increasing, outperforming Clockwork by up to 4.6× at 144*k* requests/second. We confirm that SHEPHERD's gains stem from its preemption and prioritization design choices in §7.2. We observe similar trends for Clockwork and SHEPHERD under the MAF-bursty workload.

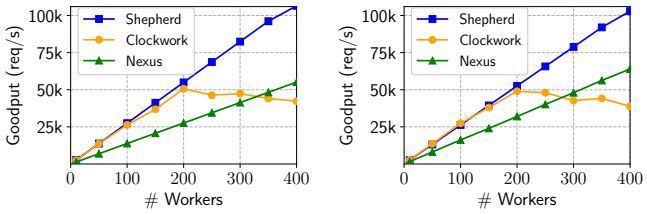
For Nexus, we find that the goodput largely remains the same as we increase the load under both MAF-stable and MAF-bursty workloads, with a goodput that is up to 7.1× and 18.1× lower than SHEPHERD. Moreover, Nexus's utilization remains under 89% for the MAF-stable workload and 55% for the MAF-bursty workload — even under high load. These observations can largely be attributed to Nexus's offline approach — during its periodic planning phase, Nexus takes the arrival rate as input and calculates the number of GPU workers required along with an offline schedule for each worker. With a fixed number of workers, Nexus can only make its planning decision assuming a specific arrival rate that it can completely satisfy, which ends up being much lower than the applied load. Moreover, during online serving phase, Nexus is unable to adjust its planning decisions dynamically based on the increased arrival rates. This impact is even more severe for the MAF-bursty workload, where predetermined execution plan is unable to adapt to periodic bursts of requests, resulting in even lower utilization (1.8× worse than SHEPHERD) and goodput relative to the MAF-stable workload.

Figure 8 plots per-request latency CDFs for SHEPHERD, Clockwork, and Nexus at low (3*k* requests/second) and high load (144*k* requests/second) for the MAF-stable workload. Note that while Figure 7(a) shows the proportion of requests admitted by each system, the CDF only depicts the latency of requests admitted by each solution. All systems observe similar latency distributions at low load (Figure 8(a)). At high load, however, a large portion of requests in Clockwork observe latency close to the SLO, since Clockwork prioritizes serving requests closer to their deadlines. Nexus also shares a similar CDF pattern, as its periodic scheduler tries to batch together as many requests as it can based on request deadlines. In contrast, SHEPHERD's request latencies are distributed more evenly; this is because SHEPHERD priortizes requests based on their batch sizes rather than their deadlines, and the evaluated workload results in batches of widely varying sizes at different times. We observe similar trends under the



(a) MAF-stable (b) MAF-bursty

Figure 9: **Goodput with varying request SLOs.** SHEPHERD outperforms Nexus and Clockwork by up to 38× and 1.3×, respectively, under tight SLOs. We omit SLOs ≤ 10ms since some of our evaluated models have higher execution latencies (Table 4).



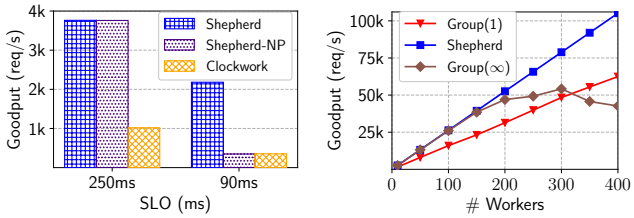
(a) MAF-stable (b) MAF-bursty

Figure 10: **Scheduler scalability with emulated workers.** For both workloads, Clockwork does not scale beyond 200 workers; Nexus scales linearly but observes 40–50% lower goodput than SHEPHERD. SHEPHERD observes both high goodput and linear scaling.

MAF-bursty workload.

Goodput with varying request SLOs (Figure 9). To understand the impact of request SLOs, we fix the arrival rate to ∼ 3*k* requests/second and measure the goodput for the compared approaches with varying SLO values. All approaches achieve high goodput with 500ms SLO, since almost all request deadlines can be met with a relaxed SLO. On reducing SLO from 500ms to 50ms, all approaches observe reduced goodput; Clockwork's reduction is smaller due to its online algorithm that prioritizes requests with tighter deadlines, while Nexus observes higher reduction, especially for the MAF-bursty workload. This is because its periodically computed static execution plan is unable to adapt to small bursts of requests, resulting in even fewer requests meeting their deadlines. However, SHEPHERD's online FLEX algorithm is able leverage prioritization and preemption to maximize the number requests that meet the stringent SLOs, outperforming both Clockwork and Nexus by up to 1.3× and 38× respectively.

Scheduler scalability (Figure 10). Due to the limited number of GPUs in our testbed, we were unable to evaluate the scalability of SHEPHERD and existing systems beyond a point. We therefore complement our testbed experiments with large-scale emulations with up to 400 *emulated* workers. As in prior work [10], an emulated worker is identical to a real SHEPHERD worker, except an inference request triggers no meaningful work; instead, they wait for a period of time determined by the corresponding model's batch-latency characteristics (Table 4), before returning a response. We run the



(a) Benefits of FLEX (b) Benefits of HERD

Figure 11: **Understanding SHEPHERD benefits.** (a) Prioritization and preemption in SHEPHERD results in 3.7× and 6.2× improvement in goodput, respectively; SHEPHERD-NP refers to a non-preemptive variant of SHEPHERD. (b) SHEPHERD achieves both high goodput and scalability with group-worker limit $\overline{G} = 12$.

MAF-stable and bursty workloads with varying number of emulated workers (N), scaling up the total load applied to the system with the number of workers. We apply a low enough load per worker to ensure any requests dropped in SHEPHERD and Clockwork are solely due to the scheduler's failure to scale to large number of workers.

Clockwork's goodput scales linearly with smaller N, slows down around $N = 150$, and saturates at 50*k* request/second around $N = 200$ since its centralized scheduler becomes the bottleneck[3] (Figure 10(a)). Nexus goodput, on the other hand, scales almost linearly with N; this is expected since Nexus's scheduling decisions are computed per-stream and updated only periodically. However, its periodically computed schedule results in ∼ 40% lower goodput than SHEPHERD. This is because Nexus's computed schedule conservatively provisions for a load that a given number of workers can sustain without adapting to any changes due to workload unpredictability, as discussed in the results for Figure 7. Finally, SHEPHERD observes both consistently high goodput and linear scaling. The linear scaling is attributed to SHEPHERD dividing its workers into groups, each with a group-worker count of 12, which is below the scalability limit of our online scheduler. The high goodput, on the other hand, is attributed to each group being large enough for efficient multiplexing across request streams. As such, SHEPHERD outperforms Clockwork and Nexus by 2.5× and 1.8× respectively in terms of goodput at $N = 400$ workers. We note, however, that SHEPHERD employs multiple schedulers — specifically, $\lceil \frac{N}{12} \rceil$ schedulers for N workers — in contrast to Clockwork's single centralized scheduler to achieve its linear scaling. We observe similar trends with the MAF-bursty workload in Figure 10(b).

7.2 Understanding SHEPHERD Benefits

We now dig deeper into how each design component in SHEPHERD contributes to its overall performance gains.

Benefits of FLEX (Figure 11(a)). To demonstrate the effec-

[3] This trend is consistent with the scalability results reported in the Clockwork paper [10] albeit with a higher peak goodput due to differences in the system implementation and execution runtime.

tiveness of batch prioritization and preemption in FLEX, we create a synthetic workload with two streams. Stream A is bursty and issues requests to the low-latency model ResNet18 ($\sim$ 4ms for batch size $= 1$, $\sim$ 32ms for batch size $= 128$). Requests in stream A arrive periodically in bursts of 1024 requests at $t = 5$ms, 125ms, 245ms, ..., i.e., with a period of 120ms. Stream B is stable and issues requests to the high-latency model ResNet269 (79ms for batch size $= 1$), and has individual requests arriving at $t = 0$ms, 1ms, 2ms ...; note that in the absence of other queued requests from stream B, any approach would schedule batches of size 1 for it every 1ms.

We provision one GPU worker for both streams, and compare the performance for SHEPHERD and Clockwork for this workload. To decouple the contributions of preemption from prioritization, we also evaluate a non-preemptive variant of SHEPHERD that retains all the properties of FLEX except preemption. We run the experiments under two different SLOs (250ms and 90ms) to separate the contributions of prioritization and preemption in SHEPHERD, as described next.

For 250ms SLO, both SHEPHERD and non-preemptive SHEPHERD outperform Clockwork by 3.7×. Since Clockwork prioritizes requests with tighter deadlines, it always ends up prioritizing high-latency requests of stream B over low-latency requests of stream A. In contrast, SHEPHERD's batch generation prioritizes larger batches — since stream A's low-latency requests can accumulate much larger batches under the 250ms SLO (*e.g.*, 128 sized batches with 32ms latency) and achieve much higher goodput. Prioritizing stream A's requests allows SHEPHERD to leverage the limited GPU resource to complete more requests in the same time span. In more detail, after a batch of stream B (comprising a single request) scheduled at time $t = 1ms$ completes after 79ms, SHEPHERD prioritizes stream A's queued requests over the remaining requests of stream B. With an SLO of 250ms, most requests in stream A can meet their SLO deadlines, permitting SHEPHERD to achieve high goodput even without preemption.

However, with a reduced SLO of 90ms, non-preemptive SHEPHERD cannot complete executing larger batches of stream A's requests within their SLO deadline since it *waits* for stream B's batch to finish (i.e., at $t = 80$ms). Thus, the performance for non-preemptive SHEPHERD is similar to Clockwork — most of stream A's requests fail to meet their deadline. With preemption, a large batch of stream A's requests preempts the scheduled (much smaller) batch of stream B's requests, allowing most requests of stream A to finish within the deadline. As such, SHEPHERD outperforms both its non-preemptive variant and Clockwork by 6.2×. Note that the performance for heuristic-driven and non-preemptive approaches can be made arbitrarily worse than SHEPHERD by increasing stream A's burst size and reducing its request execution latency, as discussed in §2.2 and Theorem 5.1, respectively.

Benefits of HERD (Figure 11(b)). We use the same setting as the large-scale emulation in §7.1 and vary the number of group-worker limit $\overline{G}$ for HERD (§4). With a group-worker limit of $\overline{G} = \infty$, SHEPHERD always chooses a group size equal to the number of workers. As such, it reduces to the online global approach, observing the same scalability limit as Clockwork (Figure 10). With a group-worker limit of $\overline{G} = 1$, SHEPHERD cannot efficiently multiplex across streams, leading to constantly lower goodput compared to SHEPHERD with multiple workers. As such, HERD allows SHEPHERD to achieve a goodput that is 2.5× and 1.7× higher than the two grouping alternatives, respectively, at 400 workers.

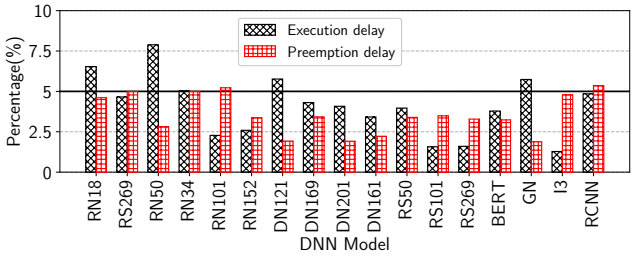


Figure 12: **Preemption overheads.** The preemption delay and additional execution delay relative to the normal batch executions for most of our evaluated models remains below 5%.

# streams	# models	# workers	solver	network	loading
200,000	200	200	0.55	0.19	0.71
400,000	400	400	2.51	0.35	1.23
600,000	600	600	4.28	0.51	1.84
800,000	800	800	8.53	0.62	2.41
1,000,000	1,000	1,000	13.26	0.90	3.14

Table 5: Components of the periodic planning latency (in seconds).

7.3 Understanding SHEPHERD Overheads

Finally, we evaluate the preemption and periodic planning overheads in SHEPHERD to show that neither impact SHEPHERD's performance benefits in any significant manner.

Preemption overheads (Figure 12). As discussed in §6, efficient preemption should minimize two overheads: (1) *preemption delay*, or the time between from the preemption being requested and actually being completed, and (2) *execution delay*, the additional latency introduced by exit points for normal batch execution. We achieve a reasonable trade-off between these two overheads by specifically tailoring appropriate number of exit points for each model listed in Table 4.

We measure the *relative* preemption overheads introduced by SHEPHERD, i.e., the preemption and execution delay relative to normal batch execution time, averaged over batch sizes 1–128. For most models, both the preemption delay and the extra execution delay are well below 5%.

Periodic planning overheads in HERD (Table 5). The periodic planning latency in HERD consists of three parts: (1) the solver latency for solving the ILP (Eq. 2), (2) the network latency for broadcasting the plan to schedulers and workers, and, (3) the loading latency for loading the models from CPU memory to GPU on each worker. We run large-scale emula-

tion to divide the system into 10 serving groups, and measure these latencies with different number of streams, models, and emulated workers. The solver latency accounts for most of the planning time, taking 13.26 seconds for 1 million streams and $1k$ workers. Network latency is always less than a second, while model loading time increases with the number of models. Even so, the total planning latency is always much smaller than HERD's scheduling period, which is tens of minutes. Moreover, the solver and network latency for the next planning phase can be pipelined with the current online serving phase, ensuring that planning is never a bottleneck.

8 Discussion and Caveat

We now outline avenues of future research in SHEPHERD.

Group predictablility under different workloads. Although we have demonstrated group predictability using two representative workloads, we note that the number of streams (i.e., group size) to achieve sufficient group predictability may be different for real-world workloads. With insufficient predictability, HERD may under or overprovision resources for some groups, although FLEX would still provide the same performance guarrantee within each group since it does not rely on predictability. Moreover, group predictability itself is rooted in statistical multiplexing theory, and holds when a large enough number of request streams in the workload have statistical independence [33–35]. While well-exploited in the networking community to achieve high utilization under bursty network traffic patterns [15–18], more in-depth quantitive analysis of group predictability for real-world inference serving workloads is important future work.

Model affinity *vs.* degree of multiplexing. Recall from §4 that HERD includes an affinity-set surjectivty constraint, which requires that models assigned to the same group j have divergence less than $\overline{K}$. With a small $\overline{K}$, HERD will break models into more groups, with each group containing fewer but more similar models, i.e., models with similar model affinity values. While this enables tighter performance guarantees in FLEX, it also reduces the degree of multiplexing within each group, since GPU workers in each group can serve streams across a smaller set of models. Although a single affinity group (i.e., $\overline{K}$=∞) yields a looser competitive ratio, our evaluation shows that it still results in high empirical performance for the MAF workload. Finding an optimal value of $\overline{K}$ is promising future work.

Fairness across request streams. Similar to prior serving system designs [6, 10], we focus on the isolated GPU cluster settings where fairness across request streams and models is not a major concern. Fairness can be an important metric to extend our design to multi-user or cloud scenarios.

Dynamic model swapping. Similar to prior work [6], SHEPHERD only loads models onto GPU memory at the start of a planning period. An alternative solution is to dynamically swap models between GPU and CPU memory on-demand during online serving [10]. However, since such swaps are likely to take much longer than serving a request, its cost must be weighed against the potential performance gains from swapping in a new model. We leave incorporating this decision as a part of online serving as future work.

Large DNN models. If a DNN model is so large that it cannot be co-located with other models in GPU memory, HERD must place it in an isolated group with reduced degree of multiplexing. It is possible, however, to break such large models into smaller partitions [36] to group them with other models for better multiplexing.

9 Related Work

We discussed existing model serving systems in §2; we now discuss prior work related to SHEPHERD in other areas.

Preemption for ML workloads. PipeSwitch [21] allows preempting a training tasks to execute an inference task. Irina [11] applies preemption to improve average latency for inference tasks. LazyBatching [8] is an inference system that can preempt and stall the currently ongoing batch. SHEPHERD leverages preemption approaches outlined in these works to achieve guaranteed high goodput. Concurrent to our work, REEF [31] leverages ISA support for preemptions [37, 38] in recent AMD GPUs to enable μs-scale preemptions. While our current implementation still implements preemptions in software, it can readily incorporate hardware-based preemptions. Future improvements in this space will only improve SHEPHERD's performance further.

Online job scheduling. The theory community has long considered issues of prioritization and preemption in online job scheduling [19, 20, 24–27]. Its adaptation to model serving, however, has a few nuances — the scheduler for model serving must also decide how to optimally execute requests across batches while taking into account model-specific batch-latency relationships. Our scheduling algorithm exploits both to achieve strong performance guarantees.

10 Conclusion

We have presented SHEPHERD, a distributed a DNN model serving system. SHEPHERD employs a periodic planner that aggregates request streams into moderately-sized groups for high utilization and scalability, and an online scheduler that employs a novel online algorithm to provide guaranteed goodput. Evaluation over production workloads shows that SHEPHERD achieves 17.2× higher goodput and 1.8× higher utilization than prior approaches and scales to hundreds of workers.

Acknowledgement

We thank our shepherd Ravi Netravali and the anonymous NSDI reviewers for their insightful feedback. This research is supported by NSF Awards 2112562, 2047220, 1730628 and gifts from AWS, Ant Group, Ericsson, Futurewei, Google, Intel, Meta, Microsoft, NetApp, Scotiabank, and VMware.

References

[1] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA*, 2018.

[2] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[3] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC*, pages 477–491, 2020.

[4] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated Model-less Inference Serving. In *ATC*, pages 397–411, 2021.

[5] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Middleware*, pages 109–120, 2017.

[6] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[7] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[8] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 493–506. IEEE, 2021.

[9] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *ATC*, pages 1049–1062, 2019.

[10] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 443–462, 2020.

[11] Xiaorui Wu, Hong Xu, and Yi Wang. Irina: Accelerating DNN Inference with Efficient Online Scheduling. In *4th Asia-Pacific Workshop on Networking*, pages 36–43, 2020.

[12] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[13] AWS. Deliver high performance ML inference with AWS Inferentia. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf., 2019.

[14] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *ATC*, pages 205–218, 2020.

[15] Ravi R Mazumdar. Performance modeling, stochastic networks, and statistical multiplexing. *Synthesis Lectures on Communication Networks*, 6(2):1–211, 2013.

[16] Basil Maglaris, Dimitris Anastassiou, Prodip Sen, Gunnar Karlsson, and John D Robbins. Performance models of statistical multiplexing in packet video communications. *IEEE transactions on communications*, 36(7):834–844, 1988.

[17] Kavitha Chandra. Statistical multiplexing. *Wiley Encyclopedia of Telecommunications*, 5:2420–2432, 2003.

[18] Hiroshi Saito, Masatoshi Kawarasaki, and Hiroshi Yamada. An analysis of statistical multiplexing in an atm transport network. *IEEE Journal on Selected Areas in Communications*, 9(3):359–367, 1991.

[19] S. Goldman, Jyoti Parwatikar, and S. Suri. Online scheduling with hard deadlines. *J. Algorithms*, 34:370–389, 2000.

[20] Richard J. Lipton and Andrew Tomkins. Online interval scheduling. In *In Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 302–311, 1994.

[21] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 499–514, 2020.

[22] Boris N Pshenichnyj. *The linearization method for constrained optimization*, volume 22. Springer Science and Business Media, 2012.

[23] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[24] Ran Canetti and Sandy Irani. Bounding the power of preemption in randomized scheduling. *SIAM Journal on Computing*, 27(4):993–1015, 1998.

[25] Xujin Chen, Xiaodong Hu, Tie-Yan Liu, Weidong Ma, Tao Qin, Pingzhong Tang, Changjun Wang, and Bo Zheng. Efficient mechanism design for online scheduling. *Journal of Artificial Intelligence Research*, 56:429–461, 2016.

[26] Sanjoy Baruah, Gilad Koren, Decao Mao, Bhubaneswar Mishra, Arvind Raghunathan, Louis Rosier, Dennis Shasha, and Fuxing Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992.

[27] Sally A Goldman, Jyoti Parwatikar, and Subhash Suri. Online scheduling with hard deadlines. *Journal of Algorithms*, 34(2):370–389, 2000.

[28] Pytorch. https://pytorch.org/.

[29] Apache tvm: An end to end machine learning compiler framework for cpus, gpus and accelerators. https://tvm.apache.org/.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[31] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[32] Pytorch hub. https://pytorch.org/hub/.

[33] Kavitha Chandra. Statistical multiplexing. *Wiley Encyclopedia of Telecommunications*, 5:2420–2432, 2003.

[34] Basil Maglaris, Dimitris Anastassiou, Prodip Sen, Gunnar Karlsson, and John D Robbins. Performance models of statistical multiplexing in packet video communications. *IEEE transactions on communications*, 36(7):834–844, 1988.

[35] Ward Whitt. Tail probabilities with statistical multiplexing and effective bandwidths in multi-class queues. *Telecommunication Systems*, 2(1):71–107, 1993.

[36] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023 (OSDI)*, 2022.

[37] GPUOpen. Amd gpu isa documentation. https://gpuopen.com/documentation/amd-isa-documentation, 2021.

[38] Nathan Otterness and James H Anderson. Amd gpus as an alternative to nvidia for supporting real-time workloads. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[39] Richard Hamming. *Numerical methods for scientists and engineers*. Courier Corporation, 2012.

A Competitive Ratio without Preemption

Theorem A.1 *No non-preemptive, deterministic algorithm can achieve a bounded competitive ratio for online serving.*

Proof Consider a batch X_1 with $|X_1| = 1$, $\ell_{X_1} = 1$ and $d_{X_1} = 1$ that arrives at time $t = 0$. A deterministic non-preemptive algorithm $\mathcal{B}$ serves X_1 with probability $p = \{0, 1\}$. We consider two scenarios after the scheduling decision at $t = 0$: (A) no request arrives afterwards, and therefore, the optimal solution has a total value of 1, and algorithm $\mathcal{B}$ has a total value of p; (B) another batch X_2 arrives at time $t = \varepsilon$ with $|X_2| = \omega \to \infty$, $\ell_{X_2} = 1$ and $d_{X_2} = 1 + \varepsilon$: the optimal solution can achieve a total value of ω by ignoring X_1, and algorithm $\mathcal{B}$ has a total value of $p + (1-p) \cdot \omega$, since it is non-preemptive.

Note that the competitive ratio should be no less than the ratio between the optimal solution over algorithm $\mathcal{B}$ in either scenario. As such, by combining both cases we show the competitive ratio c of algorithm $\mathcal{B}$ should be no less than:

$$c \geq \max_{p=\{0,1\}} \left(\frac{1}{p}, \frac{\omega}{p + (1-p) \cdot \omega}\right) \to \infty \tag{5}$$

which completes the proof ■

B Competitive Ratio Analysis for FLEX

We define a *schedule* σ to be a sequence of batch executions (B, t_B), where t_B is the start time of batch B in the schedule σ. Note that since we allow preemption, some batches may get preempted and never complete; we use $\sigma^c \subset \sigma$ to denote the set of completed batches in σ and $\sigma^p \subset \sigma$ to denote the set of preempted batches. We say a schedule σ is *feasible* if (1) at any time, at most one batch $B \in \sigma$ is executing, and, (2) a request is completed (i.e., executed without being preempted) in at most one batch $B \in \sigma^c$. Let $v(\sigma) = \sum_{B \in \sigma} v(B, t)$ denote the aggregated value of all batches in σ. We have,

$$v(\sigma) = v(\sigma^c) + v(\sigma^p) \tag{6}$$

We use standard competitive analysis to evaluate our algorithm. We denote the schedule due to an algorithm $\mathcal{A}$ as $\sigma_{\mathcal{A}}$, and the optimal schedule constructed by a computationally unbounded offline algorithm as σ_*. We say that algorithm $\mathcal{A}$ is *c-competitive* if for any request stream we have:

$$c \cdot v(\sigma^c_{\mathcal{A}}) \geq v(\sigma_*) \tag{7}$$

To better differentiate batch sequences (B, t) between schedule $\sigma_{\mathcal{A}}$ and σ_*, we denote the batch sequences in $\sigma_{\mathcal{A}}$ as (I, t_I) and batch sequences in σ_* as (J, t_J). Moreover, for a batch $I \in \sigma_{\mathcal{A}}$, we denote its (1) start time as t_I; (2) value (batch size) as $|I|$; and (3) duration as ℓ_I. The same notation rules apply to $J \in \sigma_*$.

We prove our main result in Theorem 5.2 in three steps. First, we consider a simplification of the online batch scheduling algorithm that only considers online batch scheduling for a single model running on a single GPU (§B.1). We then extend the setting to include multiple models deployed on a single GPU (§B.2), and finally consider the general case of multiple models deployed across multiple GPUs (§B.3).

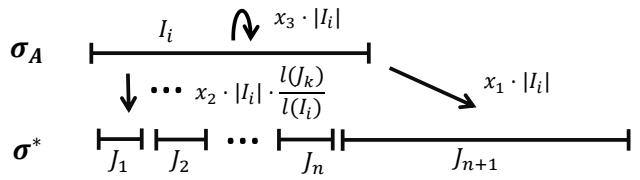
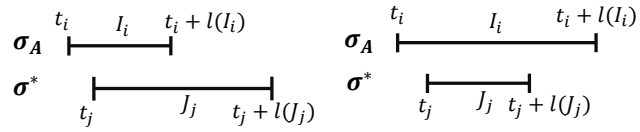


Figure 13: **The value assignment rule from one $I \in \sigma_{\mathcal{A}}$ to Js $\in \sigma_*$.**



(a) $J_j \in \sigma^*$ **blocked by** $I_i \in \sigma_A$ **(b)** $J_j \in \sigma^*$ **covered by** $I_i \in \sigma_A$

Figure 14: **The block and cover relationship between batches in $\sigma_{\mathcal{A}}$ and σ_*.** Note that if J is identical to I then J is covered by I (by our definition of blocking).

B.1 Single-GPU Single-Model Setting (sgsm)

For the single GPU, single model setting our key result is:

Theorem B.1 *Algorithm 1 is* 10.81*-competitive with a single model on a single GPU with preemption threshold* $\lambda \approx 2.38$.

Proof To prove the above theorem, we bound the value of batches in the optimal schedule (σ_*) by the value of completed batches in $\mathcal{A}$'s schedule ($\sigma^c_{\mathcal{A}}$). To this end, our analysis builds on the *value assignment approach* employed in [19, 20]. This approach operates in two steps:

1. Mapping. First, we *map* each batch in $\sigma_{\mathcal{A}}$ to a set of batches in σ_* in a manner that ensures each batch in σ_* is matched to at least one batch in $\sigma_{\mathcal{A}}$. This mapping identifies batches in σ_* that are related to batches in $\sigma_{\mathcal{A}}$, either because they overlap in their execution durations, or the batches contain common requests. More formally we define three relationships to compare a batch $J \in \sigma_*$ with a batch $I \in \sigma_{\mathcal{A}}$ (Figure 14):

M1. J is *blocked* by I if $t_I \leq t_J < t_I + \ell_I \leq t_J + \ell_J$.

M2. J is *covered* by I if $t_I < t_J$ and $t_I + \ell_I > t_J + \ell_J$.

M3. J is *intersected* by I if neither **R1** or **R2** hold, and $\exists r$ such that, $r \in I$ and $r \in J$.

We say J is temporally related to I if either **R1** or **R2** holds, and spatially related if **R3** holds.

2. Assignment. We assign values from each batch $I \in \sigma_{\mathcal{A}}$ to its mapped batches $J \in \sigma_*$, which we denote as $v_a(I, J)$. This assignment must satisfy two properties. First, it should be *feasible*, i.e., for any $I \in \sigma_{\mathcal{A}}$ its total assignment to all batches in σ_* should be equal to the value of I:

$$\sum_{J \in \sigma_*} v_a(I, J) = |I|, \quad \forall I \in \sigma_{\mathcal{A}} \tag{8}$$

Second, the assignment should be bounded, i.e., the total value assigned from all $I \in \sigma^c_{\mathcal{A}}$ to to all $J \in \sigma_*$ must be greater than or equal to a constant portion of the aggregated value of $J \in \sigma_*$:

$$\sum_{J \in \sigma_*} \sum_{I \in \sigma^c_{\mathcal{A}}} v_a(I,J) \geq r \cdot \sum_{J \in \sigma_*} |J| \tag{9}$$

where $r \in [0,1]$ is a constant. Note that for an assignment that is both feasible and bounded, we have:

$$\begin{aligned} v(\sigma^c_{\mathcal{A}}) &= \sum_{I \in \sigma^c_{\mathcal{A}}} |I| \\ &= \sum_{I \in \sigma^c_{\mathcal{A}}} \sum_{J \in \sigma_*} v_a(I,J) \\ &= \sum_{J \in \sigma_*} \sum_{I \in \sigma^c_{\mathcal{A}}} v_a(I,J) \\ &\geq \sum_{J \in \sigma_*} r \cdot |J| \\ &\geq r \cdot v(\sigma_*) \end{aligned} \tag{10}$$

Based on the definition of competitive ratio in Eq. 7, Eq. 10 suggests a competitive ratio of $c = \frac{1}{r}$.

The key tasks that remain are defining a feasible and bounded assignment v_a, and quantifying the bound r achieved by this assignment.

Defining the value assignment v_a: We are now ready to define our value assignment; as Figure 13 shows, a batch I in $\sigma_{\mathcal{A}}$ may cover n batches J_{c1} to J_{cn} (see **M2**), block at most one batch J_b (see **M1**) and intersect m batches J_{i1} to J_{im} (see **M3**). Our value assignment rules are as follows:

- **A1.** For a batch J_b that I blocks, $v_a(I,J_b) = x_1 \cdot |I|$.
- **A2.** For a batch J_c that I covers, $v_a(I,J_c) = x_2 \cdot |I| \cdot \frac{\ell_{J_c}}{\ell_I}$, i.e., the assigned value is proportional to the duration of J_c. Moreover, since the total duration of all covered batches J_{c1} to J_{cn} is no more than ℓ_I, the total assignment across J_{c1}, ..., J_{cn} is no more than $x_2 \cdot |I|$.
- **A3.** For a batch J_i that I intersects, we assign a value of x_3 to J_i for every request that is common between I and J_i, i.e., $v_a(I,J_i) = x_3 \cdot |I \cap J_i|$. Since each request will be executed at most once in σ_*, the total assigned value from I across all J_i is no larger than $x_3 \cdot |I|$.
- **A4.** If the total assigned value from I is less than $|I|$, we assign the residual value of I to any arbitrary $J \in \sigma_*$.

It is clear to see that the above assignment ensures that the total assignment from any batch $I \in \sigma_{\mathcal{A}}$ to all $J \in \sigma_*$ equals $|I|$, i.e., satisfies the feasibility constraint Eq. 8, if $(x_1 + x_2 + x_3) \leq 1$. Next, we quantify for each batch $J \in \sigma_*$, the lower-bound r to satisfy the boundedness constraint (Eq. 9).

3. Determining the bound r: A key challenge in determining the bound r for value $|J|$ relative to the value assigned to it, as per the boundedness constraint Eq. 9, is that a batch $I \in \sigma_{\mathcal{A}}$ and a batch $J \in \sigma_*$ can be related both temporally and spatially as outlined in our mapping step. As such, each such case requires analysis for the bound. As a concrete example, consider a batch J blocked by a batch I (as per **M1**). One possible reason J is not executed in $\sigma_{\mathcal{A}}$ is because a subset $J^E \subset J$ of requests may already have been dequeued from Q_m in $\sigma_{\mathcal{A}}$ and thus will not be executed again. Based on the dequeue condition in Algorithm 1, J^E is the subset of all requests in J that have already completed in $\sigma_{\mathcal{A}}$ at time t_J. On the other hand, it may be the case that J is not executed in $\sigma_{\mathcal{A}}$, because the value added by subset of requests $J^R \subset J$ that still remain to be executed (i.e., $J^R = J \setminus J^E$) is less than twice the value of the batch executed by $\sigma_{\mathcal{A}}$ in its place, namely I.

To accurately capture the impact of both of the above effects in determining the bound r, we define *virtual batches* J^R and J^E for each batch $J \in \sigma_*$ as above (see Figure 15). We denote the fraction of requests in J which belong to J^R as p, so that J^E contains the remaining $1-p$ fraction of requests. Note that p can take different values in $[0,1]$ for different J in σ_*. Since the value of a batch equals batch size, we have:

$$\begin{aligned} |J^R| &= p \cdot |J| \\ |J^E| &= (1-p) \cdot |J| \end{aligned} \tag{11}$$

Our next step is to determine the bound r based on J^R and J^E independently (r_R and r_E, respectively), and take the tighter of the two as our final lower bound, i.e., $r = max(r_R, r_E)$.

Determining r_R based on J^R: We first consider the lower-bound bound imposed on the value of J by only considering the virtual batches J^R. To this end, we confine ourselves to assignment rules **A1** and **A2** corresponding to blocked and covered batches, respectively.

- **Case 1: $I \in \sigma_{\mathcal{A}}$ blocks J^R.** Since J^R is blocked by I, it must be the case that $|J^R| \leq \lambda \cdot |I|$; otherwise J^R would have preempted I in $\sigma_{\mathcal{A}}$ at time t_J. Combined with the assignment rule **A1**, this gives us:

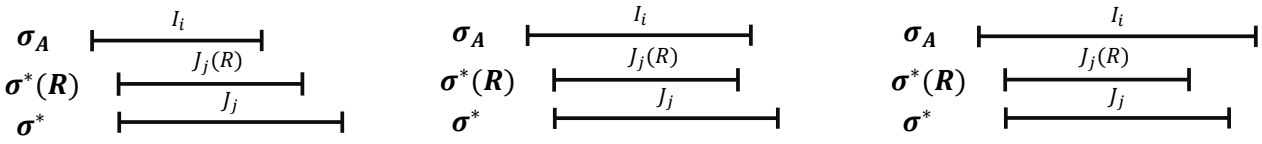
$$\begin{aligned} \sum_{I \in \sigma_{\mathcal{A}}} v_a(I,J) &\geq x_1 \cdot |I| \\ &\geq x_1 \cdot (\frac{1}{\lambda} \cdot |J^R|) \\ &= x_1 \cdot \frac{1}{\lambda} \cdot p \cdot |J| \end{aligned} \tag{12}$$

- **Case 2: $I \in \sigma_{\mathcal{A}}$ covers J^R.** To determine the lower bound in this case, we exploit two properties. First, since I covers J^R, $\ell_I > \ell_{J^R}$, i.e.,

$$\ell_I > \ell_{J^R} \tag{13}$$

Second, we exploit the property that a given model can always execute larger batches with smaller latency per record. Since I covers J^R,

$$\frac{|I|}{\ell_I} > \frac{|J^R|}{\ell_{J^R}} \tag{14}$$



(a) Case 1 (b) Case 2a (c) Case 2b

Figure 15: All possible conditions in $\sigma_{\mathcal{A}}$ for a batch $J \in \sigma_*$. Here schedule $\sigma_*(R)$ denotes the batch sequence of $(J(R), t_J)$.

Finally, as Figure 15(b)-(c) shows, this case can be further broken down into the following two sub-cases based on the relation between I and the real batch J:

– **Case 2a: I blocks J.** Assignment **A1** gives us:

$$\begin{aligned}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) &\geq x_1 \cdot |I| \\ &> x_1 \cdot \frac{|J^R|}{\ell_{J^R}} \cdot \ell_I \\ &= x_1 \cdot \frac{p\cdot|J|}{\ell_{J^R}} \cdot \ell_I \\ &> x_1 \cdot p \cdot |J|\end{aligned} \tag{15}$$

– **Case 2b: I covers J:** Assignment **A2** combined with Eqs. 13 and 14 applied to J gives us:

$$\begin{aligned}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) &\geq x_2 \cdot |I| \cdot \frac{\ell_J}{\ell_I} \\ &> x_2 \cdot \frac{|J|}{\ell_J} \cdot \ell_I \cdot \frac{\ell_J}{\ell_I} \\ &> x_2 \cdot |J|\end{aligned} \tag{16}$$

Note that in this case we do not need consider J^R to determine the bound since I directly covers J.

- **Case 3:** If neither of the above cases occur, then $\sigma_{\mathcal{A}}$ must be idle at time t_J. This implies that J^R must have been empty (i.e., $p = 0$), otherwise J^R would have been scheduled in $\sigma_{\mathcal{A}}$. Therefore, the following trivial bound holds:

$$\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \geq p\cdot|J| = 0 \tag{17}$$

Combining all the cases (Eq. 12, Eq. 15, Eq. 16 and Eq. 17), we have for any $J \in \sigma_*$:

$$\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \geq \min(\frac{p\cdot x_1}{\lambda}, x_2)\cdot|J| \tag{18}$$

Note that we omit the term from **Case 3**, since the corresponding inequality is dominated by $\frac{1}{2}\cdot x_1 \cdot p$ with $x_1 \leq 1$. Similarly, the term from **Case 2a** is also omitted since it is dominated by Case 1.

Aggregating both sides of Eq. 18 over all $J \in \sigma_*$, we get:

$$\begin{aligned}\min(\frac{p\cdot x_1}{\lambda}, x_2)\cdot\sum_{J\in\sigma_*}|J| &\leq \sum_{J\in\sigma_*}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \\ &= \sum_{I\in\sigma_{\mathcal{A}}}\sum_{J\in\sigma_*} v_a(I,J) \\ &= \sum_{I\in\sigma_{\mathcal{A}}}|I| \\ &= v(\sigma_{\mathcal{A}})\end{aligned} \tag{19}$$

Next, we show how we can upper-bound $v(\sigma_{\mathcal{A}})$ by $v(\sigma^c_{\mathcal{A}})$. Note that batches in $\sigma_{\mathcal{A}}$ can form a chain based on the preemption relation. For each chain, the next batch on the chain preempts the previous one, and each chain must ended with a batch in $\sigma^c_{\mathcal{A}}$. We denote the chain which ends with batch $|I|$ as $chain(I)$. Denote $v(chain(I))$ as the value of all the batches in $chain(I)$, since each batch in $\sigma_{\mathcal{A}}$ will be covered by exactly one chain, we have

$$\sum_{I\in\sigma^c_{\mathcal{A}}} v(chain(I)) = \sum_{I\in\sigma_{\mathcal{A}}}|I| = v(\sigma_{\mathcal{A}}) \tag{20}$$

Moreover, based on the preemption rule we have that for each chain, the value of the ith batch in the chain must be no less than $\lambda\times$ the value of the $i-1$th batch. As such, $v(chain(I))$ must be no higher than $\frac{\lambda}{\lambda-1}\times|I|$, which indicates that:

$$\begin{aligned}v(\sigma^c_{\mathcal{A}}) &= \sum_{I\in\sigma^c_{\mathcal{A}}}|I| \\ &\geq \sum_{I\in\sigma^c_{\mathcal{A}}}\frac{\lambda-1}{\lambda}\cdot v(chain(I)) \\ &= \frac{\lambda-1}{\lambda}\cdot\sum_{I\in\sigma_{\mathcal{A}}}|I| \\ &= \frac{\lambda-1}{\lambda}\cdot v(\sigma_{\mathcal{A}})\end{aligned} \tag{21}$$

Combined with Eq. 6, we have,

$$\begin{aligned}v(\sigma_{\mathcal{A}}) &\leq \frac{\lambda}{\lambda-1}\cdot v(\sigma^c_{\mathcal{A}}) \\ &= \frac{\lambda}{\lambda-1}\cdot\sum_{I\in\sigma^c_{\mathcal{A}}}|I| \\ &= \frac{\lambda}{\lambda-1}\cdot\sum_{J\in\sigma_*}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J)\end{aligned} \tag{22}$$

Combining Eqs. 19 and 22, we get:

$$\sum_{J\in\sigma_*}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \geq \frac{\lambda-1}{\lambda}\cdot\min(\frac{p\cdot x_1}{\lambda},x_2)\cdot\sum_{J\in\sigma_*}|J| \quad (23)$$

This gives us a bound $r_R = \frac{\lambda-1}{\lambda}\cdot\min(\frac{p\cdot x_1}{\lambda},x_2)$.

Determining r_E based on J^E: Note that all requests in J^E must have been completed in $\sigma_{\mathcal{A}}$. So based on our assignment rule **A3**, any request in J^E must have been assigned a value of x_3 from one *completed* batch from $\sigma^C_{\mathcal{A}}$ (the set of completed batches in $\sigma_{\mathcal{A}}$). The above observation permits bounding $|J|$ based on J^E as follows:

$$\begin{aligned}\sum_{I\in\sigma^c_{\mathcal{A}}} v_a(I,J) &\geq \sum_{I\in\sigma^c_{\mathcal{A}}} x_3\cdot|I\cap J| \\ &\geq x_3\cdot|J^E| \\ &= (1-p)\cdot x_3\cdot|J|\end{aligned} \quad (24)$$

Aggregating both sides of Eq. 24 over all $J\in\sigma_*$, we get $r_E = (1-p)\cdot x_3$.

4. Determining the optimal competitive ratio: Combining the bounds r_R and r_E, we get the final competitive ratio:

$$\begin{aligned}c_{sgsm} &= \frac{1}{\max(r_R,r_E)} \\ &= \frac{1}{\min\limits_{p\in[0,1]}\{\max\{\frac{\lambda-1}{\lambda}\cdot\min\{\frac{p\cdot x_1}{\lambda},x_2\},(1-p)\cdot x_3\}\}}\end{aligned} \quad (25)$$

To minimize the value of c_{sgsm}, we can select appropriate values of x_1, x_2 and x_3 subject to the feasibility constraint $x_1+x_2+x_3\leq 1$, and select appropriate value of $\lambda\in(1,\infty)$. Note, however, that we cannot select p — it can take arbitrary values in $[0,1]$; as such, we have to consider the worst-case value for p to compute the competitive ratio. This provides us the following optimization problem:

$$\begin{aligned}&\min_{x_1,x_2,x_3,\lambda} c_{sgsm} \\ &\texttt{s.t. } x_1+x_2+x_3\leq 1 \\ &1<\lambda<\infty\end{aligned} \quad (26)$$

Solving the above optimization problem (via numerical methods [39]) yields the minimal value for $c_{sgsm} =\sim 10.81$ with preemption threshold $\lambda\approx 2.38$. ■

B.2 Single-GPU Multi-Model Setting (sgmm)

We now extend our analysis to the setting with k models $\{m_1,...,m_k\}$ deployed on a single GPU. The competitive analysis for the multi-model case also leverages the linear relationship between batch size and batch execution latency: for model m_i, the execution latency for a batch B is $\alpha_i\cdot|B|+\beta_i$, where α_i and β_i are model-specific constants

Theorem B.2 *Algorithm 1 is $10.81\cdot K$-competitive with multiple models on a single GPU, with preemption threshold $\lambda\approx 2.38$ and K defined in Eq. 4*

Proof The proof shares a similar structure with the single-GPU, single-model case, and is identical until we determine r_R based on J^R. Even so, the analysis for **Case 1** is still the same, since the preemption rule remains unchanged. For **Case 2**, however, Eq. 14 no longer holds, since with multiple models, a batch with larger length may have a smaller value density than a batch for a different model. However, with Eq. 1 we can replace Eq. 14 with:

$$K\cdot\frac{|I|}{\ell_I} > \frac{|J^R|}{\ell_{J^R}} \quad (27)$$

Now we show why Eq. 27 always holds. Assume I and J are batches for models m_1 and m_2 respectively. We have

$$\begin{aligned}\frac{|J^R|}{\ell_{J^R}}\cdot\frac{\ell_I}{|I|} &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\frac{\beta_1}{|I|}) \\ &\leq \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\beta_1) \\ &= \frac{|J^R|\cdot(\alpha_1+\beta_1)}{\alpha_2\cdot|J^R|+\beta_2} \\ &\leq K\end{aligned} \quad (28)$$

Note that Line 1 to Line 2 is based on the implicit constraint that $|I|\geq 1$ since it can only take integer values.

To further improve the bound, we notice that as $\ell_I > \ell_{J^R}$ always holds in Case 2, with Eq. 1 we have

$$\begin{aligned}&\alpha_1\cdot|I|+\beta_1 > \alpha_2\cdot|J^R|+\beta_2 \\ \rightarrow\ &|I| > \frac{\alpha_2\cdot|J^R|+\beta_2-\beta_1}{\alpha_1}\end{aligned} \quad (29)$$

On one hand, if $\alpha_2+\beta_2-\beta_1>0$, we have $\alpha_2\cdot|J^R|+\beta_2-\beta_1>0$. Then we can replace I in the first line of Eq. 28 with Eq. 29:

$$\begin{aligned}&\frac{|J^R|}{\ell_{J^R}}\cdot\frac{\ell_I}{|I|} = \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\frac{\beta_1}{|I|}) \\ &< \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\frac{\beta_1\cdot\alpha_1}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\frac{\alpha_1\cdot(\alpha_2\cdot|J^R|+\beta_2-\beta_1)+\beta_1\cdot\alpha_1}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\frac{\alpha_1\cdot\alpha_2\cdot|J^R|+\alpha_1\cdot\beta_2}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\frac{\alpha_1\cdot(\alpha_2\cdot|J^R|+\beta_2)}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{\alpha_1\cdot|J^R|}{\alpha_2\cdot|J^R|+\beta_2-\beta_1} \\ &\leq K\end{aligned} \quad (30)$$

Then for Case 2a we will have:

$$\begin{aligned}\sum_{I\in\sigma_A} v_a(I,J) &\geq x_1\cdot|I| \\ &> \frac{x_1\cdot p\cdot|J|}{K}\end{aligned} \quad (31)$$

For Case 2b we have:

$$\sum_{I\in\sigma_A} v_a(I,J) \geq x_2\cdot|I|\cdot\frac{\ell_J}{\ell_I} > \frac{x_2\cdot|J|}{K} \tag{32}$$

The analysis for Case 3 and J^E is the same as the single model case. Similar to Eq. 25, by combining all cases we can calculate the competitive ratio c_{sgmm} for the multi-model case:

$$c_{sgmm} = \frac{1}{\min\limits_{p\in[0,1]}(\max(\frac{\lambda-1}{\lambda}\cdot\min(\frac{p\cdot x_1}{\lambda},\frac{p\cdot x_1}{K},\frac{x_2}{K}),(1-p)\cdot x_3))} \tag{33}$$

Since $K\geq 1$, combining Eqs. 33 and 25 gives us:

$$c_{sgmm}\leq K\cdot c_{sgsm} \quad \forall x_1,x_2,x_3,p,\lambda \tag{34}$$

As such, Algorithm 1 can always achieve a competitive ratio of $10.81\cdot K$ for the single-GPU, multi-model setting. ■

B.3 Multi-GPU Multi-Model Setting (mgmm)

Finally, we extend our analysis to the general case with k models $\{m_1,...,m_k\}$ and N GPUs. The major difference lies in the per-GPU preemption rule for the request arrival event — the new preemption rule ensures that at any time, no available batch will have a value $\lambda\times$ higher than the value of the currently running batches on *any* GPU. Moreover, the modified dequeue rule ensures that in the multi-GPU case, a request is completed in *at most one* batch in $\sigma_{\mathcal{A}}$.

We have the following theorem for the general case.

Theorem B.3 *For the multi-GPU, multi-model case, Algorithm 1 is $12.62\cdot K$-competitive with preemption threshold $\lambda\approx 3.03$, with K defined in Eq. 4.*

Proof The proof follows the same structure as the single-GPU, single-model setting as well. Define the schedule $\sigma_{\mathcal{A}}(u)$ as the schedule of Algorithm $\mathcal{A}$ on GPU $u\in[1,N]$ and $\sigma_*(v)$ as the optimal schedule on GPU $v\in[1,N]$. We have $\sigma_{\mathcal{A}}=\bigcup_u\sigma_{\mathcal{A}}(u)$ and $\sigma_*=\bigcup_v\sigma_*(v)$. Moreover, we define (u,v) as a GPU pair between the schedule $\sigma_{\mathcal{A}}(u)$ and $\sigma_*(v)$.

Value assignment rule between GPU pair (u,v) We apply a similar value assignment rule in the basic case *for each GPU pair* (u,v) in the general case. The major difference lies in assignment rules **A1** and **A2**, where we evenly spread the value for I from each $\sigma_{\mathcal{A}}(u)$ to all $\sigma_*(v)$ with different v.

- **A1.** For a batch $J_b\in\sigma_*(v)$ that $I\in\sigma_{\mathcal{A}}(u)$ blocks, $v_a(I,J_b)=\frac{x_1}{N}\cdot|I|$.
- **A2.** For a batch $J_c\in\sigma_*(v)$ that $I\in\sigma_{\mathcal{A}}(u)$ covers, $v_a(I,J_c)=\frac{x_2}{N}\cdot|I|\cdot\frac{\ell_{J_c}}{\ell_I}$.
- **A3.** For a batch $J_i\in\sigma_*(v)$ that $I\in\sigma_{\mathcal{A}}(u)$ intersects, we assign a value of x_3 to J_i for every request that is common between I and J_i, i.e., $v_a(I,J_i)=x_3\cdot|I\cap J_i|$.
- **A4.** If the total assigned value from $I\in\sigma_{\mathcal{A}}(u)$ is less than $|I|$, we assign the residual value of I to a random $J\in\sigma_*(v)$.

Similar to the basic case, the above pair-wise assignment rule ensures the following property:

Feasibility: For any GPU $u\in[1,N]$, with $(x_1+x_2+x_3)\leq 1$, the total assignment from any batch $I\in\sigma_{\mathcal{A}}(u)$ to all J in all $\sigma_*(v)$ equals $|I|$. That is:

$$\sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)} v_a(I,J)=|I|, \quad \forall I\in\sigma_{\mathcal{A}}(u) \tag{35}$$

Boundedness: Similar to the basic case (Eq. 9), the assignment should be bounded. Here we want to show that the total value assigned from all I in all $\sigma^c_{\mathcal{A}}(u)$ to all batches J in all $\sigma_*(v)$ must be greater than or equal to a constant portion of the aggregated value of J in all $\sigma_*(v)$. That is:

$$\sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)}\sum_{u\in[1,N]}\sum_{I\in\sigma^c_{\mathcal{A}}(u)} v_a(I,J)\geq r\sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)}\cdot|J| \tag{36}$$

where $r\in[0,1]$ is a constant. Note that similar to the basic case (Eq. 10), for an assignment that is both feasible and bounded, we have:

$$\begin{aligned} v(\sigma^c_{\mathcal{A}}) &= \sum_{u\in[1,N]}\sum_{I\in\sigma^c_{\mathcal{A}}(u)}|I| \\ &= \sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)}\sum_{u\in[1,N]}\sum_{I\in\sigma^c_{\mathcal{A}}(u)} v_a(I,J) \\ &\geq \sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)} r\cdot|J| \\ &\geq r\cdot v(\sigma_*) \end{aligned} \tag{37}$$

Eq. 37 suggests a competitive ratio of $c=\frac{1}{r}$.

Determining the bound r: The key task that remains is to quantify the bound r achieved by the assignment. Similar to the basic case, this is done by bounding the values of J for each $\sigma_*(v)$ by values of I for each $\sigma_{\mathcal{A}}(u)$, based on both the J^E and J^R parts.

Determining r_R based on J^R: We can apply the same analysis as in the basic case for each GPU pair (u,v). Note that for **Case 1**, the modified preemption rule ensures that at any time, no available batch in $\sigma_{\mathcal{A}}$ will have a value $\lambda\times$ higher than the value of the currently running batches on *any* GPU. As such, the J^R from any GPU u must have a value no higher than $\lambda\times$ the value of the I blocks it in any u, which indicates:

$$\begin{aligned} \sum_{I\in\sigma_{\mathcal{A}}(u)} v_a(I,J) &\geq \frac{x_1}{N}\cdot|I| \\ &\geq \frac{x_1}{N}\cdot(\frac{1}{\lambda}\cdot|J^R|) \\ &= \frac{x_1}{N}\cdot\frac{1}{\lambda}\cdot p\cdot|J| \end{aligned} \tag{38}$$

Moreover, the analysis for **Case 2** and **Case 3** follows the same logic. Formally, for any u and $J \in \sigma_*(v)$ we have:

$$\sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) \geq \begin{cases} \frac{x_1 \cdot p}{\lambda} \cdot \frac{|J|}{N}, & \text{Case 1} \\ \frac{x_1 \cdot p \cdot |J|}{K \cdot N}, & \text{Case 2a} \\ \frac{x_2 \cdot |J|}{K \cdot N}, & \text{Case 2b} \\ p \cdot |J|, & \text{Case 3} \end{cases} \tag{39}$$

Since the above equation holds for each $\sigma_{\mathcal{A}}(u)$, we have:

$$\sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) \geq \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K}) \cdot |J| \tag{40}$$

Aggregating both sides of Eq. 39 over all J in all $\sigma_*(v)$, we get:

$$\begin{aligned} & \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K}) \cdot \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} |J| \\ \leq & \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) \\ = & \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} |I| \\ = & v(\sigma_{\mathcal{A}}) \end{aligned} \tag{41}$$

Next, we bound $v(\sigma_{\mathcal{A}})$ by $v(\sigma^c_{\mathcal{A}})$. We apply the same chain analysis as we did for the basic case for each $\sigma_{\mathcal{A}}(u)$. More specifically we have:

$$\sum_{I \in \sigma^c_{\mathcal{A}}(u)} v(chain(I)) = \sum_{I \in \sigma_{\mathcal{A}}(u)} |I| \qquad \forall u \in [1,N] \tag{42}$$

Then based on the preemption rule we have:

$$\begin{aligned} v(\sigma^c_{\mathcal{A}}) &= \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} |I| \\ &\geq \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} \frac{\lambda - 1}{\lambda} \cdot v(chain(I)) \\ &= \frac{\lambda - 1}{\lambda} \cdot \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} |I| \\ &= \frac{\lambda - 1}{\lambda} \cdot v(\sigma_{\mathcal{A}}) \end{aligned} \tag{43}$$

Combining Eqs. 41 and 43, we get:

$$\begin{aligned} & \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} v_a(I,J) \\ \geq & \frac{\lambda - 1}{\lambda} \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K}) \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \cdot |J| \end{aligned} \tag{44}$$

This gives us a bound $r_R = \frac{\lambda - 1}{\lambda} \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K})$.

Determining r_E based on J^E: Note that based on the dequeue and preemption rule in Algorithm 1, some request in J^E may not have been completed in $\sigma_{\mathcal{A}}$. Instead, it only ensures that for any $J \in \sigma_*(v)$, all requests in the corresponding J^E must *have been (or being) executed* in some $\sigma_{\mathcal{A}}(u)$. Since each of the requests in J^E gets assigned a value of x_3 (based on **A3**), we have the following bound:

$$\begin{aligned} \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) &\geq \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} |I \cap J_i| \\ &\geq x_3 \cdot |J^E| \\ &= (1-p) \cdot x_3 \cdot |J| \end{aligned} \tag{45}$$

Note that Eq. 45 is in the exact same form as Eq. 40. So following the same procedure from Eq. 41 to Eq. 44 we can get:

$$\begin{aligned} & \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} v_a(I,J) \\ \geq & \frac{\lambda - 1}{\lambda} (1-p) \cdot x_3 \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \cdot |J| \end{aligned} \tag{46}$$

This gives us a bound $r_E = \frac{\lambda - 1}{\lambda}(1-p) \cdot x_3$.

Determining the optimal competitive ratio c_{mgmm}: Combining the bounds r_R and r_E, we get the final competitive ratio:

$$\begin{aligned} c_{sgsm} &= \frac{1}{\max(r_R, r_E)} \\ &= \frac{1}{\frac{\lambda - 1}{\lambda} \cdot \min\limits_{p \in [0,1]} \{\max\{\min(\frac{p \cdot x_1}{\lambda}, \frac{p \cdot x_1}{K}, \frac{x_2}{K}), (1-p) \cdot x_3\}\}} \end{aligned} \tag{47}$$

Similar to the basic case, we can select appropriate values of x_1, x_2, x_3 and λ to minimize c_{mgmm}.

$$\begin{aligned} & \min_{x_1, x_2, x_3, \lambda} c_{mgmm} \\ & \texttt{s.t.} \;\; x_1 + x_2 + x_3 \leq 1 \\ & 1 < \lambda < \infty \end{aligned}$$

■

Solving the above optimization problem yields the maximal value for $c_{mgmm} =\sim 12.62 \times K$ with preemption threshold $\lambda \approx 3.03$.

C Complexity Analysis for FLEX

Theorem C.1 FLEX *has a worst-case complexity of O(G), where G is the number of GPUs in the serving group.*

Proof Batch generation (Algorithm 2) has a complexity of $O(|\mathbb{M}| \cdot |Q|)$ where $|\mathbb{M}|$ is the number of models queues with newly enqueued requests since last update, and $|Q|$ is the largest queue size among these model queues. For each request arrival event, batch generation is triggered $O(G)$ times. Moreover, between every two invocations, at most one model queue changes. Therefore, $|\mathbb{M}|$ is at most 2 for each invocation (Line 2 in Algorithm 2). In addition, since each preemption will increase the size for the running batch by at least $\lambda\times$, each GPU can only be preempted by at most $O(log_\lambda(|Q|))$ times.

Algorithm 2 Batch generation algorithm in FLEX

```
1: procedure BATCHGEN(n)
2:     M ← models with newly enqueued requests or currently running on GPU n
3:     for each model m ∈ M do
4:         Dequeue requests passing their deadlines from Q(m)
   # Line 5-13: Find largest feasible batch B_g(n,m) for model m
5:         Candidate request set S ← Q(m)
6:         if B_c(n) uses model m then
7:             S ← Q(m) ⋃ B_c(n)
8:         B_g(n,m) ← ∅
9:         for request r in S with ascending deadline do
10:            if r can meet SLO with batch size |B_g(n,m))| then
11:                Add r to B_g(n,m)
12:            else
13:                Break
14:    B_g(n) ← B_g(n,m) with largest batch size among all models
15:    Return B_g(n)
```

As such, the re-enqueue event (Line 19 and 11) will be triggered by at most $O(log_{\lambda}(|Q|))$ times for each GPU. The complexity of enqueue operation is $O(log(|Q|))$ (Line 10), and the complexity of re-enqueue operation is $O(|Q|+|B_c(n)|)$ (Line 19). Note that $|B_c(n)|$ can never be larger than $|Q|$ by definition. Note that $|Q|$ and λ are constants. Based on the above analysis, the total complexity for each request arrival event and batch completion event is $O(G)$. Similar analysis applies for each batch completion event. Taken together, FLEX has an overall complexity of $O(G)$. ■

Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE

Kshiteej Mahajan*, Ching-Hsiang Chu†, Srinivas Sridharan†, Aditya Akella$
University of Wisconsin - Madison, Facebook†, UT Austin$*

Abstract: Emerging ML training deployments are trending towards larger models, and hybrid-parallel training that is not just dominated by compute-intensive all-reduce for gradient aggregation but also bandwidth-intensive collectives (e.g., all-to-all). These emerging collectives exacerbate the communication bottlenecks despite heterogeneous network interconnects with ample multipath opportunities. In this work, we propose SYNDICATE, a systematic, general framework to minimize communication bottlenecks and speed up training for both state-of-the-art and future large-scale models and interconnects. SYNDICATE proposes a novel abstraction, the motif, to break large communication work as smaller pieces as part of execution planning. SYNDICATE also does joint optimization of scheduling and execution planning by rethinking the interfaces in the networking systems stacks used for ML training. Motifs afford greater flexibility during scheduling and the joint optimizer exploits this flexibility by packing and ordering communication work so as to maximize both network utilization and overlap with compute. This improves the speed of training state-of-the-art large models by 21-74%.

1 Introduction

Training machine learning (ML) models is a common-case workload at any data-driven enterprise. To keep up with evolving data and maintain a competitive edge, enterprises are employing more sophisticated features and more complex model architectures, and attempting to train faster at ever larger scales and to deploy high-quality models frequently.

These trends are exemplified by the deep learning recommendation model (DLRM). DLRM is used in recommendation systems at several large organizations. These models use a mixture of continuous and categorical features obtained from user data. The model architectures, which are themselves rapidly evolving, uses a mixture of multi-layer perceptrons and embedding table lookups. The model capacity and compute is increasing exponentially year-on-year [24].

At production scale, such state-of-the-art models use a mixture of data and model parallelism to efficiently scale-out to a large number of machines in the training cluster. This induces rich communication collectives such as all-reduce, all-to-all, collective-permute, and all-gather [21, 24]. The resulting communication operations (comm-ops) are a major bottleneck to end-to-end training performance [24].

Evolution in networking infrastructure in training clusters [32, 35] (to include faster interconnects such as NVLink/NVSwitch, RoCE, Infiniband and support faster transports such as Remote Direct Memory Access (RDMA)) does not in itself help address these bottlenecks. These advancements need to be coupled with effective computation-communication scheduling and execution planning optimizations. These optimizations hide communication by maximizing overlap with compute and help maximize utilization of the networking infrastructure.

Unfortunately, existing scheduling optimizations [16, 18, 29] and execution planners [10, 11, 20, 33, 34] fall short. These works make several restrictive assumptions limiting their applicability to simplistic models, training settings, and networks. Communication schedulers make assumptions about the model and training architecture (simple layer-by-layer models [29] with data-parallel training) or deployment mode (Parameter Server-based [16, 18]), and the execution planners make simplifying assumptions about the nature of comm-ops (only all-reduce [10, 11, 20, 33, 34] or only push-pull [20]).

Moreover, existing solutions are point solutions in the optimization space and fail to *jointly* optimize scheduling and execution planning concerns. Schedulers today are unaware of the optionality during execution planning, such as parallel execution of two comm-ops over non-overlapping network communication channels, and might impose orders that fail to leverage such opportunities during execution planning. As a result, they leave significant room for optimization.

We seek a comm-op optimization framework that jointly optimizes planning and scheduling, applies to state-of-the-art large models with complex communication patterns, is generalizable to future large models and arbitrary network interconnects. Our framework should also encapsulates all possible optimization axes, and enables a systematic, thorough, automatic search through the space for optimal strategies.

Enabling systematic joint optimization of scheduling and execution planning is challenging. First, the communication systems stacks used for ML training today place scheduling and execution planning concerns in two different layers. The scheduler is co-located with ML training frameworks (such as PyTorch, TensorFlow) and the execution planner is co-located with communication libraries (such as NCCL, MPI). These are governed by two different developer communities and the scheduler interacts with the execution planner via a narrow, one-way API to just submit comm-ops. Moreover, the scheduler and the execution planner only accommodate fast, deterministic procedures so as to enable tight co-ordination across worker processes that peer with each other using parallel programming frameworks (such as MPI) during training.

Second, scheduling happens at the very coarse granularity of collectives submitted by the training application which limits scheduling flexibility as it leads to fewer opportunities to reorder communication work in time and efficiently pack communication work in space, i.e., over the heterogeneous mix of communication channels and bandwidth available in the networking infrastructure.

We propose SYNDICATE, a system for joint optimization of scheduling and execution planning with several innovations:

- SYNDICATE proposes a novel abstraction, the motif, to break large communication work in comm-ops into smaller units of communication work. Motifs afford greater flexibility, by helping pack and order communication work so as to maximize network multipath utilization and to maximize overlap with compute.
- Similar to query optimization backed by a well-defined relational algebra, we present a novel algebra atop motifs that systematically codifies the search space of correct, composable motif operators used to transform comm-ops into motifs and enables comm-op optimization.
- SYNDICATE rethinks the interfaces in the commmunication stack and enables joint optimization via the joint action of a control plane and a data plane. The former executes a time-intensive, non-deterministic joint optimization out-of-band without blocking the latter which enables fast execution of tightly co-ordinated motifs.
- We blend techniques used for optimal tensor operator fragmentation [19], DAG scheduling [15] and query replanning [22] to probabilistically search the joint optimization space to yield near-optimal comm-op optimization plans. We also introduces a novel shim-layer above existing communication libraries to enforce these plans.

We implement the enforcer atop existing communication libraries by extending the `torch-ucc` interface; the joint optimizer as a separate python process; and enable safe interaction between the central optimizer and the enforcer via a two phase commit protocol. We present the evaluation of several state-of-the-art models on a 128-GPU cluster with rich multipath opportunities. SYNDICATE demonstrates 21–74% faster training than the closest state-of-the-art [29] and is better than hand-optimized trainers.

2 Background

The compute and capacity of models has been increasing exponentially [1], with model training compute approaching 1000s of petaflop/s-days [9] and model capacity approaching trillions of parameters [24]. To train ever larger models, training clusters are scaling up to thousands of devices [21].

In this section, we give a short primer on the compute parallelization strategies used for ML training and the accompanying communication operations (comm-ops) that are issued. We also discuss how training network infrastructure is evolvong to deal with higher network loads.

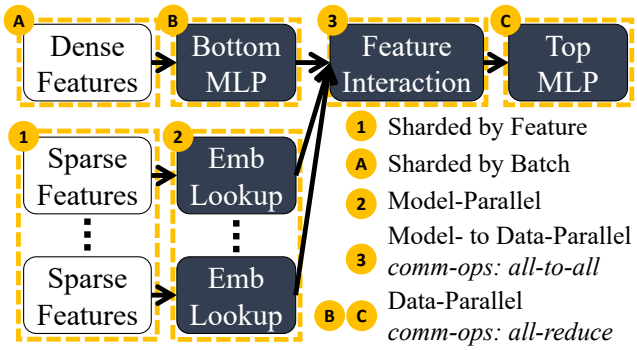


Figure 1: DLRM Model

2.1 Parallelization Strategies

We exemplify the different parallelization strategies via the Deep Learning Recommendation Model (DLRM). The largest DLRM used in production has trillions of parameters [12, 24, 26], making DLRM training especially challenging. DLRM uses a hybrid mix of parallelization strategies for different model parts (similar to BERT [13], Megatron [30], GPT [9]).

Figure 1 shows the DLRM model architecture. The training data comprises a mixture of dense continuous features and sparse categorical features (one-hot encoded or multi-hot encoded data), which are first mapped to a common embedding space using the bottom multi-layer perceptron (MLP) and the embedding table lookups respectively. The output embeddings go through a feature interaction phase and are then fed to the top MLP to get the recommender model output.

Data-Parallelism: With data-parallelism, all the model parameters are replicated across all the training devices and each device has a worker process computing parameter gradients in parallel. In the case of DLRM, the bottom MLP and top MLP use data-parallelism for training in production. These MLPs are compute intensive but not memory intensive and the MLP parameters fit within a single device memory.

Model-Parallelism: Data-Parallelism does not work for models with large capacity and with input datasets that cannot be trivially sharded into batches. With Model-Parallel training, the model is partitioned (and not replicated) across different devices. For DLRM, the embedding table lookup models and the input tables are large and memory-intensive, and as a result are parititioned across different devices during training resulting in model-parallel training.

Hybrid-Parallelism: As seen so far, different portions of DLRM training use different parallelization strategies. This is known as hybrid-parallelism. In the most general case, models can be replicated or partitioned in several different ways during training [19, 21, 25], resulting in hybrid-parallelism.

FSDP: Fully Sharded Data Parallelism [8] is a memory-efficient version of data-parallelism. It shards the model state (weights, gradients, optimizer state) for each layer of the model. During forward- or backward-propagation of a layer it enables data-parallel computation by first doing an all-gather of all the model state at all the devices and reshards the updated state post-computation by doing a scatter. This leads

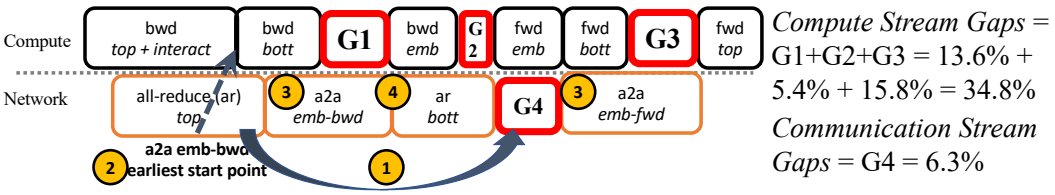


Figure 2: Gaps in DLRM Training Trace

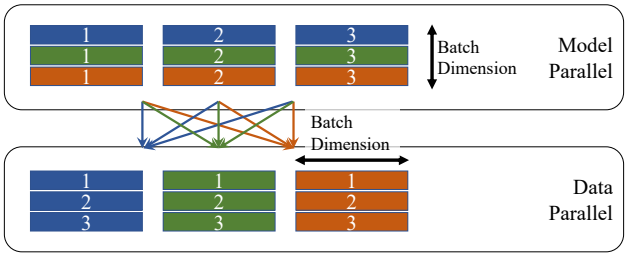


Figure 3: Illustration of all-to-all collective in DLRM

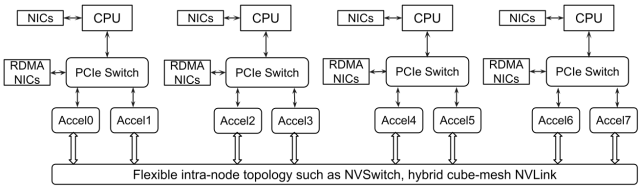


Figure 4: State-of-the-art system architecture of training cluster such as Nvidia DGX/HGX-like systems [24, 28]

to memory-efficiency and as a result allows to pack larger models in the same cluster resources.

2.2 Communication Operations (Comm-Ops)

Different parallelism strategies induce different comm-ops.

With data-parallel training, gradients computed at each worker process are aggregated layer-by-layer (during backward pass). Each aggregation yields an **all-reduce** collective.

After the embedding lookups in DLRM (② in Figure 1), each device has a vector for the table lookup models resident on those devices for all the samples in the batch, which needs to be reorganized and sharded along the batch dimension. This induces an **all-to-all** pattern of collective communication, as shown in Figure 3.

Collectives from the MPI standard [23]: In the general case, hybrid-parallel or FSDP model training [8, 14, 21] results in several types of comm-ops, ranging from all-reduce, all-to-all, collective-permute, all-gather, reduce-scatter to any collective defined in the MPI standard [23].

2.3 Evolving Network Infrastructure

The aforementioned comm-ops push increasing amounts of network traffic and the network infrastructure is adapting with fatter topologies and faster interconnects to ensure the needed throughput and latency.The network infrastructure in a state-of-the-art training cluster [24, 28] is shown in Figure 4. Each node has multiple CPU cores and accelerators such as GPUs, with frontend Network Interface Controllers (NICs) connected to the host CPUs and a dedicated RDMA NICs such as InfiniBand and RDMA over Converged Ethernet (RoCE) for each of the GPUs connected via PCIe switches. The RDMA NICs from across nodes can be connected with a dedicated network. The extensible design of this node allows to scale-out the network to interconnect thousands of nodes, forming a data-center scale training cluster. This cluster has heterogeneous mix of networking interconnects and protocols with varying throughput and latency guarantees. There are multiple communication channels between any two endpoints. At an intra-node level, a pair of GPUs can communicate via shared memory, NVLink, PCIe or the external network. At an inter-node level, any two GPUs can communicate via GPUDirect RDMA [27] or TCP/IP over Ethernet.

3 Motivation

Communication Bottleneck: Despite the networking infrastructure upgrades, the execution of comm-ops are a source of excessive delays in training. As an example of the issues that can arise in large model training, Figure 2 shows the execution of CUDA stream kernels on a randomly chosen GPU during a single iteration of production scale DLRM training [1]. The training creates a compute and a communication stream for serialized execution of tensor operator kernels and comm-op kernels, respectively. We note that there are several gaps during execution. A gap on a stream occurs when the stream is waiting for the result of kernel execution on the other stream. The compute stream gaps are wider (34.8%) and cumulatively larger than those in the communication stream (6.3%). This means that communication is a training bottleneck as it blocks compute for a third of the iteration. We now show that there are several opportunities to optimize comm-ops.

Better Scheduling Opportunity: Reordering of comm-ops improve compute/communication overlap. As shown in Figure 2 – ①: the top MLP all-reduce comm-op can be split judiciously and partially executed later to occupy the gap G4; ②: as a result the all-to-all backward comm-op can be pulled up to begin as soon as possible to reduce the gap G1.

Better Execution Planning Opportunity: Existing comm-ops do not efficiently utilize multiple communication channels available in heterogeneous network interconnects. We highlight this in Fig.2 – ③: both the all-to-all's can be broken up into smaller fragments of communication work and executed one fragment at a time to reduce incast and improve through-

[1] We show accurate percentages and hide low-level details.

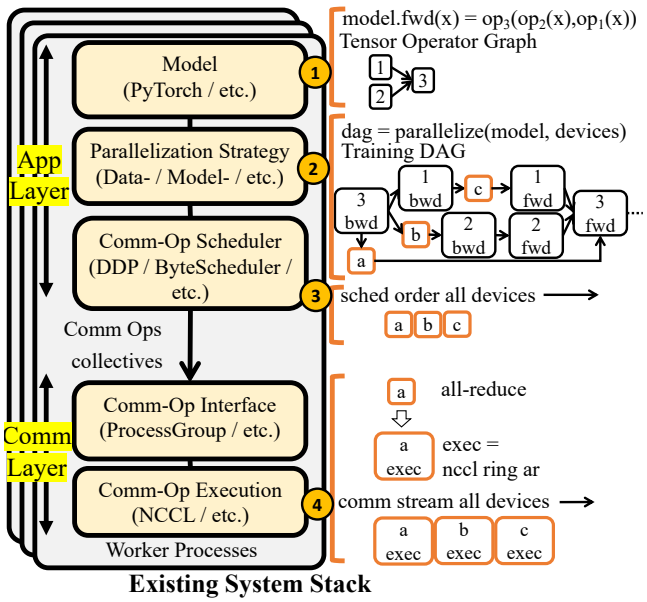


Figure 5: ML Training Communications Stack

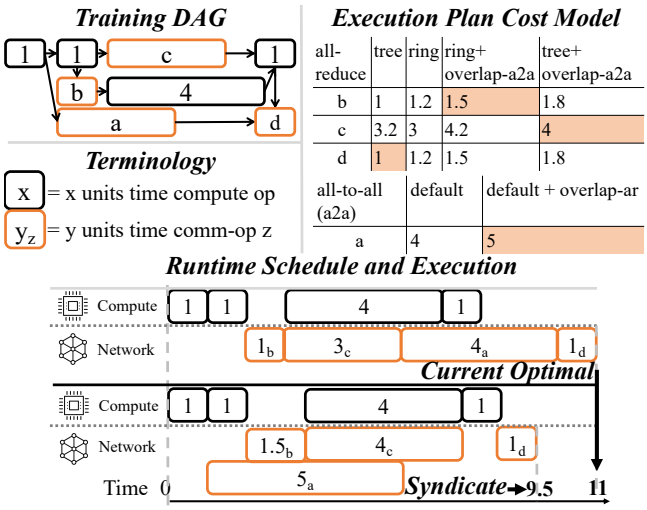


Figure 6: Motivating Example

put to reduce gaps G1 & G3; ④: all-to-all and all-reduce can be executed in parallel over communication channels with non-overlapping interconnects to start all-reduce sooner and drive higher network throughput to reduce gap G2.

Existing works to reduce communication overheads are optimal for specific training scenarios (PS architecture [16], layer-by-layer models [29], all-reduce collectives [20,34]; §7); and the scheduling and execution planning techniques proposed therein make restrictive assumptions, making it unclear as to how to compose and apply these different techniques towards hybrid-parallel training of large DLRM-like models.

A fundamental shortcoming of these works is that they do not explore *joint optimization* (§3.1) mainly because *existing interfaces in the communication stack used for ML training are not naturally amenable* (§3.2). We also note that *a collective is often too coarse-grained* to schedule communication work; breaking it up improves communication optimization flexibility (§3.3). We describe these issues next.

3.1 Disjoint Scheduling, Execution Planning

3.1.1 Communication Stack Overview

Figure 5 shows the two sets of layers in the communication stack used for ML training – the application (app) layer and the communication (comm) layer – and the four steps leading to execution of a comm-op over the network –

① **Model Definition:** The user defines a model by composing various tensor operations. The example shows a model declaration with three operators and its tensor operator graph.

② **Parallelization Strategy:** The parallelize module (e.g., `nn.DistributedDataParallel` in PyTorch) takes the model and the set of devices and converts the computation to a training DAG. The vertices are compute-ops or comm-ops and edges capture dependencies. Above we show the training DAG for a single iteration; the ops in the DAG are managed by spawning several worker processes on all the devices by using a parallel programming library such as MPI.

③ **Comm-Op Scheduling:** The communication scheduler takes the training DAG as input and decides a ordering for the comm-ops that maximizes compute/communication overlap. The scheduling procedure is deterministic and executes on all the worker processes so that the comm-ops are issued (and executed) every training iteration in *the same order on all the devices*. The default PyTorch order is FIFO.

④ **Comm-Op Execution Planning:** The comm layer at all the devices receive the comm-op from the app layer via an interface with a well-defined API (e.g., `ProcessGroup` in PyTorch). The execution planner on receiving each comm-op, assigns it an execution plan and queues it in the same order on the devices' i.e., GPU's communication stream for serialized execution. The example shows an all-reduce collective which binds to NCCL's ring all-reduce implementation during execution. Each collective typically has several options for its execution plan (e.g., ring or tree for all-reduce collective) and execution planners, such as NCCL, have network topology aware cost models that estimate the execution time for different options. Current execution planners are greedy and select the execution plan option with the least execution time. All worker processes bind to the same execution plan.

Thus, scheduling is an app-layer concern today, governed by schedulers in ML training frameworks as PyTorch, while execution planning is a comm-layer concern governed by execution planners in communication libraries as NCCL. As these are not jointly optimized, several inefficiencies arise, which we exemplify next.

3.1.2 Example to highlight suboptimality

Figure 6 illustrates the lost opportunities due to a lack of joint optimization of scheduling and execution planning. The network topology is similar to that illustrated in Figure 4. The training DAG in this example has four collectives: `a` is an all-to-all collective and `b`, `c`, `d` are all-reduce collectives.

There are several execution plan options for each collective. There is a cost associated with each option which measures the execution time over the network. For all-reduce, the execution plan options are tree all-reduce or the ring all-reduce, both using NVLink and GPUDirect RDMA. For all-to-all, there are two options: pairwise exchange between all processes either using NVLink and GPUDirect RDMA or using PCIe complex and TCP/IP over ethernet. With the latter option for all-to-all, all-to-all and all-reduce can be overlapped. We compare the iteration time of the current solution against SYNDICATE.

Current Solution: The execution planner greedily selects the fastest option for each collective resulting in a training iteration time of 11 units.

SYNDICATE Solution: SYNDICATE realizes that by jointly making changes to the scheduling order and execution plan choices there is opportunity to overlap all-to-all with all-reduce and speed-up communication by utilizing network heterogeneity. The current solution's scheduling order executes all-to-all last and does not allow overlap. SYNDICATE's scheduling order executes all-to-all collective at the very beginning and allows overlap. The execution plan choices made by SYNDICATE are shaded in the execution plan cost model in Figure 6. SYNDICATE's execution planner is not greedy and chooses a slower execution plan for all-to-all so as to allow for parallel execution of all-to-all and all-reduce over non-overlapping interconnects in the network. Overall, this results in a training iteration time of 9.5 units.

Joint optimization is beneficial but current interfaces are not amenable as we discuss next.

3.2 Interface constraints Joint Optimization

The training DAG scheduler in the ML processing frameworks is unaware of optionality (e.g., an all-reduce can be executed by as a ring all-reduce or a tree all-reduce) present lower down the stack during execution planning. A trivial extension of the existing interface is to expose the training DAG to the comm layer and push the scheduling concern down the stack to co-locate it with the execution planner. Exposing the training DAG down the stack is necessary to ensure that any reordering of comm-ops down the stack does not lead to *dependency violations*: a child comm-op cannot be ordered before a parent comm-op as otherwise it can lead to a *deadlock*. This enables joint optimization without dependency violations. However, the joint optimization problem is NP-hard and the joint optimization procedure requires a time-intensive, randomized algorithm (§4.3). As a result, this procedure can delay comm-op execution due to its time-intensive nature. To make matters worse, this randomized procedure, may lead to divergent scheduling orders across different processes. This can lead to *out-of-sync* issues, wherein if the collectives are not submitted in the same order across two different processes then it results in a *deadlock* where each process waits for the other process to issue the same collective as itself. As a result, the existing interfaces are unable to trivially accommodate joint optimization of these concerns.

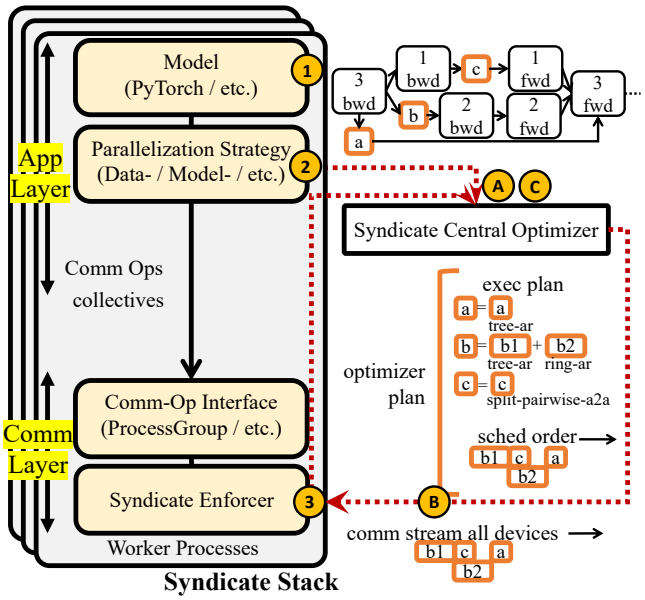


Figure 7: Overview of SYNDICATE's ML training Communication Stack

3.3 Issues with Coarse-Grained Scheduling

Scheduling today happens at the granularity of user-submitted comm-ops i.e., collectives. Communication libraries such as NCCL, submit each comm-op as a kernel on the GPU communication stream and a kernel cannot be context-switched during execution. This means that once a comm-op is scheduled for execution it cannot be stopped mid-execution. This leads to limited scheduling flexibility in space and time.

If the payload is very large then each network transfer in the comm-op, once scheduled for execution, occupies the network links for a long time. Likewise, if the pattern of network transfers is large (e.g., a clique of network transfers) then the comm-op gang schedules transfers on a large fraction of network interconnects. Comm-ops, if scheduled as-is, thus have large communication work orders and limit the ability to both context switch and efficiently pack communication work over available heterogeneous interconnects.

4 SYNDICATE Design

SYNDICATE changes the interfaces in the communication stack to enable joint optimization of scheduling and execution planning. It builds on the *motif* abstraction to enable deconstructing comm-ops into smaller work units along a few dimensions and allow finer-grained scheduling. In this section, we start with an overview of the new interfaces and the new modules in SYNDICATE's communication stack and how it enables joint optimization (§4.1). We then explain the motif abstraction (§4.2), the joint optimizer design (§4.3), and enforcement of the joint optimizer's decisions (§4.4).

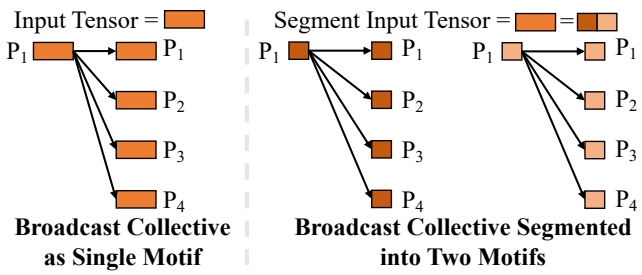


Figure 8: Example showing segmentation of broadcast collective. Left half shows the broadcast collective as a single motif. Right half shows broadcast collective segmented into two motifs, where each motif broadcasts one half of the bytes from the original tensor.

4.1 Overview

Figure 7 shows SYNDICATE's communication stack. Notably, we propose two new entities: a central optimizer and an enforcer. SYNDICATE co-locates scheduling and execution planning concerns in the centralized joint optimizer. The central optimizer generates an optimizer plan. This plan contains instructions on how to execute as well as how to schedule the comm-ops during training and is conveyed to the enforcer on each worker process. In this regards, *the central optimizer is the the control plane while the enforcer is the data plane*. We propose interfaces (Ⓐ, Ⓑ, Ⓒ) between the central optimizer and the communication stack. These interfaces are out-of-band and asynchronous, meaning that the data plane does not, in any circumstances, block execution of a comm-op waiting for control plane instructions.

We now go over the workflow in SYNDICATE. We divide it into control plane workflow and the data plane workflow.

Stepping through the control plane workflow –

Ⓐ **Joint Optimization:** The central optimizer pulls the training DAG from the app layer and the network topology from the comm layer. The optimizer uses these inputs to construct the execution plan cost model and does joint optimization to yield the optimizer plan (§4.3).

Ⓑ **Optimizer Plan Distribution:** The joint optimizer plan is sent to the enforcer on all the worker processes (§4.3).

Ⓒ **Feedback:** The central optimizer pulls comm-op performance statistics from the enforcer to help refine the execution plan cost model and potentially redo joint optimization (§4.4).

Stepping throught the data plane workflow –

① **Model Definition:** The user defines a model by composing tensor operators. This yields a computation graph (§3.1).

② **Parallelization Strategy:** The computation graph is converted to a training DAG (§3.1). The comm-ops from the training DAG are submitted every training iteration as-is to the comm layer in the default FIFO order without applying any scheduling optimizations.

③ **Optimizer Plan Enforcer:** The comm-ops are executed as instructed by the central optimizer (§4.3).

4.2 Motif Abstraction

A motif is a logical grouping of several point-to-point transfers over the network. The enforcer schedules and executes communication work at the granularity of motifs. A motif once issued to the device e.g., as a kernel on GPU communication stream is non-preemptible and occupies network resources until its communication work is completed.

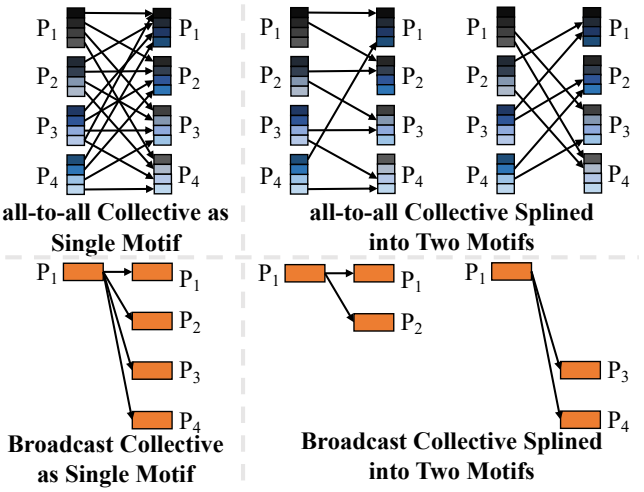


Figure 9: The left half shows all-to-all and broadcast collective as a single motif that bundles the transfers from all the source processes to all the destination processes. The right half shows both the collectives splined into two motifs, where each motif transfers the same payload from the source process to one half of the destination processes.

Conversion of comm-op to motifs: Each comm-op i.e., a collective has two attributes: a payload (typically tensors) and a pattern of network transfers. We propose two transformation operators to slice a comm-op either along the payload dimension or the pattern dimension into one or more motifs. As compared to the original comm-op, each motif represents a smaller unit of communication work (with reduced payload size and/or smaller pattern). Since SYNDICATE does scheduling at the granularity of motifs, this enables finer-grained scheduling with increased opportunities for making more frequent scheduling decisions in time to enable better overlap of compute/communication and packing smaller units of communication work more efficiently over the network resources.

4.2.1 Motif Transformation Operators

Segmentation and Splining: SYNDICATE proposes two transformation operators: segmentation and splining. Segmentation splits the payload into smaller payload segments. Splining splits the pattern of network transfers into smaller patterns. Figure 8 and Figure 9 illustrates these operators.

Transformation Operator Algebra: We now formalize the algebra for the motif transformation operators. The goal of this algebra is to state concrete rules for transforming comm-ops into motifs. This formalization succinctly encodes: (1) correct and admissible motif transformations, (2) correct and admissible transformation combinations, and (3) a structured space for all possible operator compositions. We denote the segmentation operator by $\stackrel{s}{=}$ and the splining operator by $\stackrel{p}{=}$. These rules are by no means exhaustive and are extensible.

We first present the symbols used in the algebra.

$\|$: Parallel Execution Permitted

$\overset{S}{=}$: Segmentation Transformation

$\overset{P}{=}$: Splining Transformation

`N` : Total number of Processes

`PG[IDs]` : Process `IDs` involved in a Collective

T_i`[0:N,0:D]` : `N` Tensors of size `D` on Process P_i with T_i`[j,0:D]` destined for Process P_j

M_{AA}`(`T_i`[0:N,0:D], PG[0:N])` : collective with all-to-all pattern of transfers with tensor T_i as payload executing on each Process P_i for all i in `0:N`

`M(`T_i`[`a_i`:`b_i`,`x_i`:`y_i`], PG[IDs])` : Motif `M` executing on each Process P_i for all i in `IDs` with payload = tensor T_i`[j, `x_i`:`y_i`]` from Process P_i destined to Process P_j for all j in a_i`:`b_i

Next, we present the algebraic rules that we use in the context of DLRM to transform all-to-all into motifs. The algebra for other comm-ops is in the Appendix §A.1.

Segmented All-To-All:

$$M_{AA}(T_i[0{:}N,0{:}D],\ PG[0{:}N]) \overset{S}{=} \|_{s=0}^{\frac{D}{d}-1} M(T_i[0{:}N,\ s{*}d{:}(s{+}1){*}d],\ PG[0{:}N])$$

With segmentation, the payload to be sent from a source process to all the destination processes is split into segments of size `d` (= T_i`[0:N, s*d:(s+1)*d]`). Segmentation of all-to-all in this way, yields $\frac{D}{d}$ motifs where each motif sends a payload of size `d` from a source process keeping the set of destination processes the same. `d` is a parameter and controls the number of motifs associated with the input all-to-all.

Splined All-To-All:

$$M_{AA}(T_i[0{:}N,0{:}D],\ PG[0{:}N]) \overset{P}{=} \|_{c=0}^{\frac{N}{n}-1} M(T_i[(i{+}c{*}n)\%N{:}(i{+}(c{+}1){*}n)\%N,\ 0{:}D],\ PG[0{:}N])$$

With splining, the pattern of network transfers in the all-to-all with each source process sending the payload to all the `N` destination processes is broken down into smaller patterns, where each source process P_i sends the same payload as before to `n` destination processes (= `(i+c*n)%N:(i+(c+1)*n)%N`). Splining of all-to-all in this way, yields $\frac{N}{n}$ motifs. Here, `n` parameterizes the all-to-all splining operator with larger `n` breaking the all-to-all into fewer motifs with larger sub-patterns.

Composition of Operators: Note that the all-to-all collective, M_{AA}`(:)`, is in fact a special case single motif (with `d` = `D` and `n` = `N`). These operators can be composed and recursively break a motif into several more finer-grained motifs. While fine-grained motifs are beneficial for scheduling flexibility, there is a fixed overhead associated with dispatching a motif as a kernel on GPU communication stream and launching it during execution and too fine-grained motifs are not desirable as these overheads can slow-down communication.

Physical Plan for Motif: Each motif bundles together several network transfers. Physical plan determines the physical interconnects that each network transfer is assigned to. Figure 10 shows an example of a physical plan for the broadcast collective. The broadcast collective is first broken into three motifs. The physical plan maps the motif to point-to-point network transfers over various interconnects available in the network. The figure also shows a toy network topology where the GPU for process $P1$ connects to all other GPUs via both PCIe and NVLink interconnects. The three motifs can be multiplexed over different interconnects. The physical plan for the first motif does a `memcpy` on process $P1$. The physical plan for the remaining two motifs use PCIe and NVLink in a mutually exclusive manner. This allows the point-to-point transfers in the three motifs to execute in parallel, maximizing utilization of multipath opportunities available in the network.

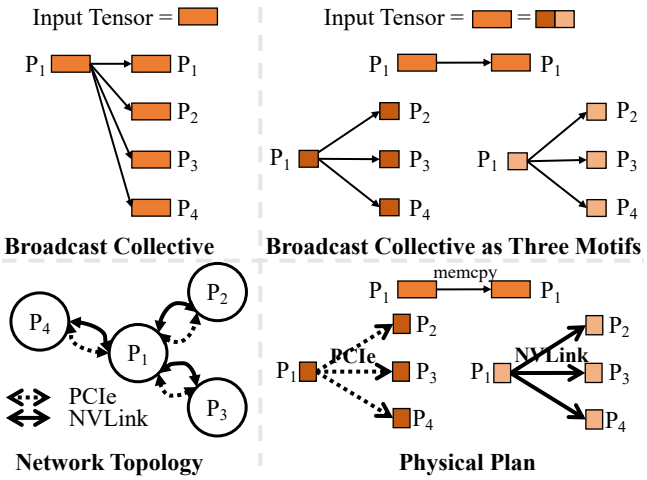


Figure 10: Physical Plan for Broadcast Collective

4.3 Central Optimizer

The central joint optimizer is responsible for minimizing training iteration time by minimizing communication overheads. The optimizer determines the optimizer plan by systematically navigating the vast space of potential schedules.

The optimizer plan has two pieces: the execution plan and the scheduling order, containing instructions regarding how to execute and how to schedule comm-ops respectively. The execution plan transforms each comm-op in the training DAG into one or more motifs. The scheduling order decides the order of execution of motifs.

Exponential Search Space: There is a lot of optionality in the execution plans for each comm-op. The transformation operators can be composed to break a comm-op into motifs in several different ways. Let us say that there are atleast `O` execution plan options for each comm-op and there are `C` comm-ops in the training DAG, then this results in O^C unique execution plan options for all the comm-ops in a DAG.

Cost of each Execution Plan: For a particular execution plan, there is an optimal scheduling order for the motifs that maximizes overlap of compute/communication and minimizes training iteration time. This training iteration time with the optimal scheduling order is the cost of the execution plan.

Problem Statement: The centralized joint optimizer takes a

Pseudocode 1 Probabilistic Search

```
1:  Training DAG with Greedy Execution Plan G*

2:  procedure MCMCSEARCH
3:      C*, sched_order* = optSched(G*)
4:      while true do
5:          G_temp = transform(G*) ▷ change execution plan for a comm-op at random
6:          C_temp, sched_order_temp = optSched(G_temp)
7:          α(C_temp | C*) = min(1, exp(β * (C* - C_temp)))
8:          G*, C*, sched_order* = G_temp, C_temp, sched_order_temp with α prob.
9:      end while
10:     return G*, sched_order*
11: end procedure

12: procedure OPTSCHED
13:     comm_q                          ▷ queue of ready communication motifs
14:     compute_q                       ▷ queue of ready compute tasks
15:     comp_time = 0
16:     comm_time = 0
17:     sched_order
18:     while comp_time ≤ comm_time and comp_q != ϕ do
19:         comp_task = fifoDequeue(comp_q)
20:         comp_time += comp_task.time()
21:         sched_order.schedule(comp_task) ▷ enqueue ready motifs, compute
22:     end while
23:     while comm_time ≤ comp_time and comm_q != ϕ do
24:         comm_motif, startTime = criticalPathDequeue(comm_q)
25:         comm_time = max(comm_time, startTime+comm_motif.time())
26:         sched_order.schedule(comm_motif)  ▷ queue ready motifs, compute
27:     end while
28:     return max(comm_time, comp_time), sched_order
29: end procedure
```

training DAG G and the network topology as inputs. We take a training DAG that unrolls compute-ops and comm-ops across two training iterations to enable cross-iteration optimizations. The aim of the joint optimizer is to take these inputs and find the execution plan with minimal cost. The joint optimizer outputs the optimizer plan, which has the execution plan and the scheduling order that minimizes overall cost.

4.3.1 Joint Optimization Procedure

The key idea in SYNDICATE is to do *probablistic search* over the exponentially large search space. We use MCMC search as outlined in Pseudocode 1.

MCMC Search: The joint optimizer starts with the default execution plan for the training DAG (denoted by G_*), wherein all the comm-ops are greedily assigned the execution plan choice with the minimum possible execution time. Thereafter, a comm-op is chosen at random and it is assigned a random execution plan option. This changes the motifs associated with this particular comm-op, keeping all the other motifs constant and yields a temporary execution plan for the training DAG (denoted by G_{temp}). The cost i.e., the execution time of this training DAG is calculated using the optSched (line 11 in Pseudocode 1) procedure which is a greedy scheduling heuristic to always dequeue motifs on the critical path in the DAG to maximize overlap of communication motifs with compute tasks or other communication motifs (in case the two communication motifs have non-overlapping physical plans). This update to the execution plan is probablistically sampled using the Metropolis-Hastings algorithm [17] and retained in G_* (line 8 in Pseudocode 1). This tends to behave as a greedy search over the search space with an ability to escape local minimas [17, 19].

Pseudocode 2 Distributed Optimizer Plan Enforcer

```
1:  Exec Plan E_colls = {..., coll_i^in : {motif_{i,j}^out}, ...}   ▷ optimal execution plan
2:  Scheduling Order S = {..., motif_{i,j}^out : seq_{i,j}^num, ...}   ▷ optimizer scheduling order
3:  Progress Queue pq   ▷ thread-safe priority queue containing ready motifs

4:  procedure ENFORCEEXECPLAN(coll^in)   ▷ app submits comm-op to comm layer
5:      {motif^out} = E_colls[coll^in]   ▷ comm-op is deconstructed into motifs
6:      for all motif^out ∈ {motif^out} do
7:          seq^num = S[motif^out]
8:          pq.INSERT(priority=seq^num, motif^out)
9:      end for
10: end procedure

11: procedure ENFORCEORDER   ▷ runs in a separate thread and enforces order
12:     nextMotifSeqNum = 0
13:     while true do
14:         while pq.TOP().priority != nextMotifSeqNum do
15:                                  ▷ busy loop until next in order motif is ready
16:         end while
17:         nextMotifSeqNum += 1
18:         {motif} = pq.POP()
19:         {motif_tensors} = {motif}.EXECUTE()
20:         REPACK({motif_tensors})
21:     end while
22: end procedure
```

Search Termination: MCMC search is terminated if the search procedure exceeds the time budget assigned for search or if the search procedure does not find a better joint optimizer plan for more than half of the total elapsed search time.

4.4 Enforcer

The central optimizer commits the same joint optimizer plan, comprising of the execution plan and the scheduling order to the enforcer on each worker process. The enforcer is responsible for *tightly co-ordinating this plan across all the worker processes during training* so as to avoid deadlocks and out-of-sync issues (§3.2). The application thread spawned by the ML processing framework at each worker process submits comm-ops to the comm layer every training iteration. With SYNDICATE, these comm-ops are submitted one-at-a-time in FIFO order. These comm-ops are intercepted by the enforcer. The enforcer is responsible for execution of these comm-ops and preparing the result of these comm-ops (tensors) to unblock the next application thread operation (compute-op or comm-op typically waiting on a CUDA stream) that is waiting on these tensors.

The enforcer takes the responsibility of executing these comm-ops as per the instructions of the optimizer plan and preparing the output tensors once ready. It does so in three steps. First, on intercepting a comm-op, it enforces the execution plan by breaking it into motifs. Second, it enforces the desired scheduling order of execution of motifs. Third, as and when motifs complete, it checks for completion of comm-ops and packages the output of individual motifs into the comm-ops output tensors. Pseudocode 2 shows the procedure to enforce the execution plan and the scheduling order contained in the joint optimizer plan. The repacking of tensors to comm-op output tensors happens after successful execution of each motif (line 20 in Pseudocode 2).

Enforcing Execution Plan: The enforcer is layered as a shim on top of existing comm-op execution layer i.e., differ-

Compute (TFLOPS)	120 (FP32)/ 1000 (FP16)
HBM	256 GB, 7.2 TB/s
DDR	1.5 TB, 200 GB/s
Scale-up bandwidth	1.2 TB/s (uni-directional)
Scale-out bandwidth	8 x 100 Gbps (uni-directional)
Host NW	2 × 100 Gbps

Table 1: Configuration of each node in our cluster

ent communication libraries such as NCCL, MPI, UCX. The app layer submits comm-ops to the comm layer using the interface between them and is immediately trapped by the `enforceExecPlan` procedure (line 4 in Pseudocode 2). This procedure deconstructs the comm-op to one or more motifs, each assigned a sequence number that captures the priority of this motif in the current training iteration. This sequence number is contained in the scheduling order of the joint optimizer plan. These motifs are enqueued into a priority queue using the sequence number as the priority.

Enforcing Scheduling Order: The `enforceOrder` procedure (line 11 in Pseudocode 2) enforces the scheduling order and runs in a thread separate from the `enforceExecPlan` procedure. This procedure maintains a priority counter that is incremented sequentially and is reset at the end of each training iteration. This counter maintains the priority of the next expected motif(s). In case of overlapping motifs, two or more motifs can be assigned the same priority. The `enforceOrder` procedure busy loops until the priority of the motif at the top of the priority queue matches the value in the priority counter. It busy loops until the `enforceExecPlan` enqueues the next expected motif. This ensures that the enforcer on all the worker processes executes motifs in the same order.

Replanning: We measure the wait time in the busy loop and send it as feedback to the central optimizer. If wait times in every iteration consistently add up to more than a threshold (= 5% of iteration time), then we redo joint optimization at the optimizer to explore a different optimizer plan.

5 Implementation

We implement the central optimizer as a separate module in python. The central optimizer simulates the execution of different execution plan choices as part of the MCMC search procedure until the search procedure terminates and yields a joint optimizer plan. The central optimizer runs on one of the machines in the training cluster and interacts with the various enforcers on the worker processes via RPCs. We build a two-phase commit (2PC) protocol using RPCs so that the same joint optimizer plan is safely committed by the central optimizer to all the enforcers. After the 2PC protocol is complete, the enforcers switch to the new joint optimizer plan from the subsequent training iteration. This ensures that out-of-sync issues are avoided whenever transitioning to a new joint optimizer plan.

We implement the enforcer in the Unified Collective Communication (UCC) library interface [6]. We implement the enforcer routines: `enforceExecPlan` routine in the `main thread` and the `enforceOrder` routine in the `progressLoop thread` in the `torch-ucc` interface [5].

Model	A1	A2	A3	A4
Num parameters	95B	793B	845B	332B
MFLOPS per sample	89	638	784	60
Num of emb tables	∼ 100s	∼ 1000s	∼ 1000s	∼ 1000s
Emb table dim ([min, max], avg)	[4, 192] avg: 68	[4, 384] avg: 93	[4, 960] avg: 231	[32, 128] avg: 72
Avg pooling size	27	15	17	49
Num MLP layers	26	20	26	43
Avg MLP size	914	3375	3210	682
Batch Size	512	1024	512	4096
Parallel Paradigm	Hybrid	Hybrid	Hybrid	FSDP [8]

Table 2: Models in our workload. Model A5 and A6 descriptions are in §6.2.

6 Evaluation

6.1 Testbed

We experimented with our prototype on a production-scale cluster using off-the-shelf NVIDIA HGX-2 based systems. Specifically, each node hosts dual-socket CPUs, 8 NVIDIA V100 GPUs that are fully-connected using NvSwitch, 2 front-end host NICs, and 8 back-end RoCE NICs to allow RDMA communication between GPUs across nodes. Table 1 summarizes the node configuration; we deployed 16 such nodes.

The testbed runs CentOS-8 and CUDA 11.4 with NVIDIA driver 470.57.02. For distributed training of DLRM models, we used PyTorch 1.11 (nightly) with the extension of Process Group UCC [5] and the latest UCC library [6], which can take advantage of various transports such as NCCL 2.10.3 [2] and UCX-based collectives [31] for dynamically selecting optimal execution planing of collective operations.

6.2 Workloads

We tested SYNDICATE in production across a breadth of scenarios; see Table 2.

Vary Model Architectures: We experiment with three different model architecture families. A1-A4 are Recommendation Models (DLRM [24]), A5 is an NLP Model (XLM-R-XL [14]), A6 is a CV Model (RegNetZ [7]).

Vary Model Sizes: We have progressively wider MLPs and higher number of embedding tables from model A1 to A3.

Vary Parallelization Strategies: Models A1-A3 are Hybrid Parallel. Model A4 is Hybrid Parallel with Fully Sharded Data Parallel (FSDP) for data parallelism [8]. Model A5 is Model Parallel. Model A6 is Data Parallel.

Vary Topologies: We vary the number of nodes (and hence GPUs) used from our testbed.

6.3 Metrics

We measure the following metrics

(1) **Training Throughput:** We measure the training throughput in terms of recommendation queries per second (A1-A4) or words per sec (A5) or images per sec (A6). Higher throughput is desirable.

(2) **Compute Idling:** We measure the idle gaps in the

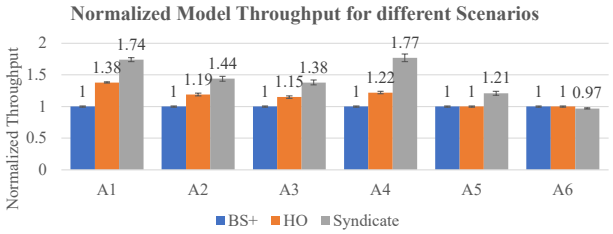


Figure 11: Training performance for SYNDICATE compared against baselines

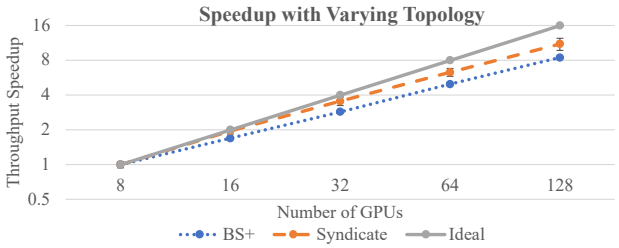


Figure 12: Speedup with varying topology sizes

GPU's compute stream and report it as a percentage of the total iteration time. Lower compute idling is desirable.

(3) **Normalized Metric:** We normalize the metric (such as throuput) against baseline using the formula – $\frac{\text{Metric with SYNDICATE}}{\text{Metric with Baseline}}$.

We run each experiment 5 times and plot the mean and standard deviation.

6.4 Baselines

We compare SYNDICATE against the following baselines.

- ByteScheduler+ (BS+): ByteScheduler [29] proposes LIFO scheduling policy for maximizing overlap of compute and communication. Their implementation only supports all-reduce in layer-by-layer models and does not have support for all-to-all collectives. We emulate ByteScheduler (BS) via our own implementation that is co-located with PyTorch framework. To emulate the bayesian optimizer used in ByteScheduler for tensor partitioning, we aid our BS scheduler with an oracle (BS+) that optimally segments tensors in all collectives.
- Hand Optimized Model (HO): Existing execution planners do not optimize all-to-all collectives and existing schedulers do not have support for DLRM-like models. We hand optimize both the scheduling policy in PyTorch (and also provide it the benefit of the segment oracle) and the choice of execution plan for each collective (including all-to-all) in the UCC library. In this regards, HO models the best possible solution with today's placement of scheduling and execution planning concerns in exiting stacks.
- SYNDICATE-Exec (S-Exec): We disable scheduling optimizations in SYNDICATE. We do so by using PyTorch's default scheduler that does FIFO scheduling to estimate the cost of each execution plan during joint optimization.
- SYNDICATE-Sched (S-Sched): We disable execution planning optimizations in SYNDICATE. We do so by disabling the MCMC search procedure and find the optimal scheduling order using SYNDICATE's scheduling heuristic (and also provide it the benefit of the segment oracle to optimally segment collectives).

6.5 Evaluation on Testbed

Figure 11 compares training performance for SYNDICATE against the BS+ and HO baselines for all the models. The Y-axis measures the model throughput normalized against that of BS+.

Vary Model Sizes: SYNDICATE outperforms BS+ by a factor of 1.74x, 1.44x, 1.38x for Models A1, A2, A3, respectively. SYNDICATE outperforms HO baseline by a factor of 1.26x, 1.21x, 1.2x for Models A1, A2, A3, respectively. Note that gains diminish with increased model sizes. The embedding tables are not compute-intensive and do not contribute to increasing the MFLOPs per sample but have a high memory footprint (and contribute to higher number of parameters) and induce progressively more communication bandwidth-hungry all-to-all's to transfer a large number of embeddings. On the other hand, MLPs are compute intensive and increase the model compute (MFLOPS per sample). On detailed analysis, we found that the larger embedding table sizes amplify the amount of time spent in all-to-all in a training iteration to a higher degree than the contribution of increased MLP compute time; which skewed the overall ratio of communication to compute and diminished the opportunity to overlap communication and compute with SYNDICATE.

Vary Model Architectures and Parallelization Strategies: SYNDICATE is effective across a range of parallelization strategies and outperforms the BS+ baselines for A2 (hybrid-parallel, recommendation model) by 1.44x, A4 (FSDP, recommendation model) by 1.77x, and A5 (model-parallel, NLP model XLM-R-XL) by 1.21x. SYNDICATE is slightly worse-off for A6 (data-parallel, RegNetZ) by 0.97x. For this model there are no opportunities to overlap comm-ops as they have linear dependency and BS+ solution (LIFO with oracle tensor partitions) is the optimal solution (similar to other CV model families, e.g., ResNet [29]). SYNDICATE is slightly worse-off due to the overheads of SYNDICATE's enforcer. The gains are for A4 are significantly higher than that for A2 despite both the models having similar model sizes and model architecture. We found that FSDP parallel strategy for A4 offers a richer set of collectives: reduce-scatter and all-gather in addition to all-to-all and all-reduce. This allows SYNDICATE to find a schedule and an execution plan that overlaps atmost three comm-ops for A4 at the same time (compared to atmost two for A2). For Model A5, we find that BS+'s LIFO schedule is optimal and hence HO does not yield any improvements. For A5, the 1.21x gains with SYNDICATE are due to better execution plan with overlap of allgather and reduce-scatter during backward pass. For Model A6, we observe no improvements with SYNDICATE. This is primarily because A6 is data

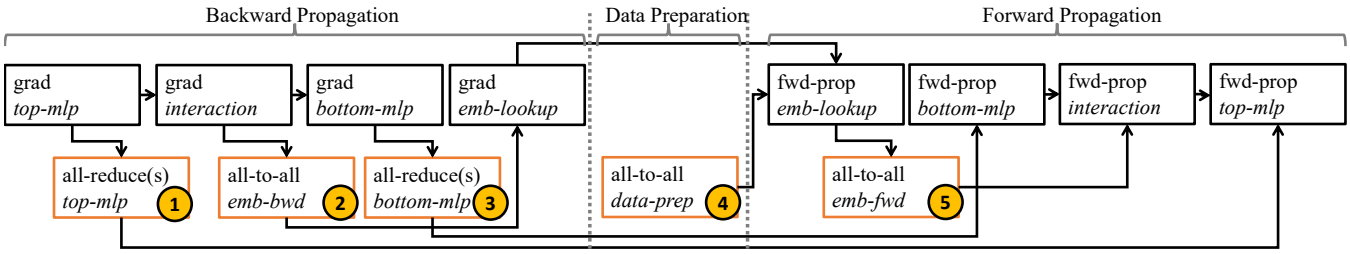


Figure 13: DLRM Training DAG. The numbers represent the order in which the PyTorch modules (`nn.DistributedDataParallel` and `nn.EmbeddingBag`) submit these collectives.

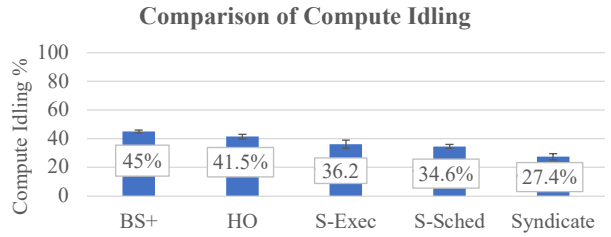


Figure 14: Comparison of Compute Idling

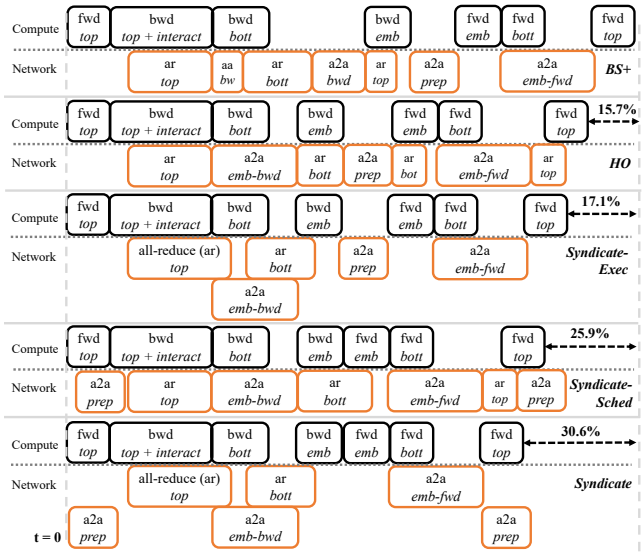


Figure 15: Zooming in on every DLRM iterations for different systems

parallel and BS+ LIFO schedule is optimal for data parallel models. Furthermore, the collectives in training DAG for A6 have serial dependencies across themselves with no room to optimize execution plan by overlapping collectives.

Vary Topologies: Figure 12 compares throughput speedup for Model A2 with SYNDICATE against BS+. We report the speedup relative to throughput on a single node. We note that SYNDICATE scales better than BS+ and is closer to ideal speedup line. SYNDICATE is better at opportunistically utilizing the increasing communication bandwidth as the cluster size scales out.

6.5.1 Sources of Improvement

Compute Idling:

Figure 14 compares the compute idling metric for SYNDICATE against baselines for Model A2. We observe that compute idling with SYNDICATE is 27.4% and is 1.64x, 1.51x, 1.32x, 1.26x less than the BS+, HO, S-Sched, S-Exec baselines, respectively. This shows that SYNDICATE is better at overcoming communication bottlenecks and achieves higher overlap of compute and communication than any other baseline. It also highlights that joint optimization is beneficial as it outperforms the S-Sched and S-Exec baselines. Next, we zoom-in on each training iteration to better understand the reasons for lesser compute idling.

Zooming in on an Iteration: We collect traces for execution of DLRM with different systems using PyTorch Kineto [4]. We illustrate these traces[2] to zoom-in on the execution of compute and communication events on the GPU streams for a single training iteration for Model A2 in Figure 15. We also show the training DAG in Figure 13 for reference. We explain these traces one system at a time.

BS+: BS+ prioritizes the execution of the most recently submitted collective (LIFO). For reference, Figure 13 shows the order of submission of collectives by DLRM PyTorch trainer. To achieve LIFO, tensors in collectives need to be optimally segmented and as explained before, we use a segment oracle to do so. BS+ is the worst-performing baseline. BS+ prioritizes execution of a2a-emb-bwd over ar-top-mlp[3], which is beneficial. However, to its detriment, it also prioritizes execution of ar-bottom-mlp over a2a-emb-bwd despite a2a-emb-bwd being on the critical path. Delaying a2a-emb-bwd also delays bwd-emb compute, which delays a2a-data-prep.

HO: To amend the drawbacks of BS+, we hand optimize the scheduling order to prioritize the execution of a2a-emb-bwd as well as a2a-data-prep before ar-bottom-mlp. We also add support for greedy execution planning for a2a collective (ar greedy optimization is available out-of-the-box). We observe that HO improves the iteration time by 15.7% as compared to BS+. We observe that the key reason for this improvement is that HO unblocks bwd-emb and fwd-emb compute sooner

[2]We hide low-level events and absolute timing information for confidentiality and legal reasons.

[3]We use a2a and ar as short hand for all-to-all and all-reduce, respectively.

and enables better overlapping of ar-bottom-mlp with these compute blocks.

S-Exec: With S-Exec, we observe that the iteration time is further improved and is 17.1% better as compared to BS+. We observe that despite placing limiting constraints on scheduling (default FIFO scheduling), the joint optimizer in SYNDICATE finds an execution plan that assigns two different communication channels to a2a and ar and enables better communication-communication overlap by leveraging heterogeneity in the network. The ar's use a communication channel over NVLink (for intra-node) and GPUDirect RDMA (for inter-node). The a2a's use a non-intersecting communication channel over PCIe (for intra-node) and TCP/IP over Ethernet (for inter-node). Such communication-communication overlap is not possible with HO and BS+ as they use traditional communication stack and interfaces therein only permit one-at-a-time execution of comm-ops with greedy execution plan.

S-Sched: With S-Sched, we observe that iteration time is further improved and is 21.9% faster than BS+, despite the constraints on execution planning (we also handicap S-Sched with choosing the default execution plan option, which is sub-optimal, for a2a). The primary reason for the improvement is that SYNDICATE's scheduler finds a superior comm-op scheduling order and SYNDICATE's enforcer enables enforcing of this order. SYNDICATE's scheduling order moves a2a-data-prep from the ith iteration and moves it back in time as to overlap it with the fwd-top-mlp and bwd-top-mlp compute blocks in the $(i-1)^{th}$ iteration. The enforcer design enables this ordering due to the presence of busy loop in the `enforceOrder` procedure in Pseudocode 2. The enforcer blocks execution of all the comm-ops in the very first training iteration until a2a-data-prep collective for the next batch is submitted by the application layer. This increases the training iteration time only for the first iteration but significantly improves the training iteration time for all the subsequent iterations by unlocking pipelining.

SYNDICATE*:* With SYNDICATE, we observe that iteration time is faster than all the baselines and is 30.6% faster than BS+. We observe that as compared to S-Sched, SYNDICATE is able to hide the overheads of ar-top-mlp by completely overlapping it with compute. SYNDICATE enables this by leveraging network heterogeneity and enabling communication-communication overlap of ar-top-mlp and a2a-emb-bwd. S-Sched, due to its execution planning constraints is unable to do so and in its scheduling order has to partially push ar-top-mlp to the very end where it cannot be overlapped with compute. In this way, SYNDICATE's joint optimizer maximizes compute-communication overlap by leveraging the benefits of communication-communication overlap.

SYNDICATE's Optimizer Plan for DLRM: Here, we summarize the key highlights of the optimizer plan that SYNDICATE finds for DLRM Model A2. In our study, we find that these observations also hold for Model A1 and Model A3.

Data Prefetch The scheduling order proposed by the optimizer moves a2a-data-prep back in time from the ith iteration to the $(i-1)^{th}$ iteration. As mentioned before, this is made possible by SYNDICATE's enforcer.

a2a-ar Overlap The execution plan proposed by the optimizer overlaps all-to-all collective with all-reduce collective over two separate communication channels as explained before. SYNDICATE binds both the a2a's to the 4-way splined execution plan, the ar-bottom-mlp to the ring all-reduce execution plan and the ar-top-mlp to the tree all-reduce execution plan. This maximizes multipath network utilization and also enables greater communication-compute overlap.

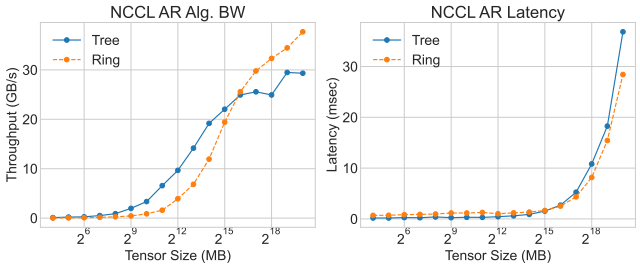


Figure 16: Effect of different execution plans on all-reduce performance

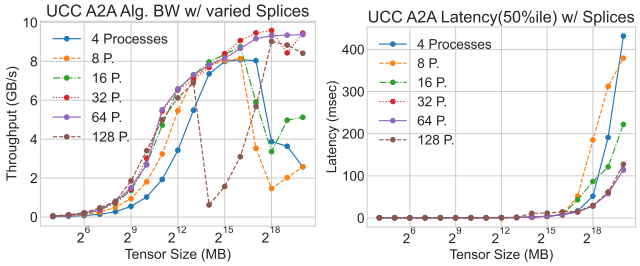


Figure 17: Effect of different execution plans on all-to-all performance

6.6 Microbenchmarks

We use the communication microbenchmark, PARAM [3] to systematically understand the space of execution plans for different collectives to better understand the choices made by SYNDICATE's optimizer plan. SYNDICATE uses these microbenchmarks as a cost model for its joint optimizer. We highlight a subset of these microbenchmarks and explain the various choices made by SYNDICATE for Model A2.

Execution planning options for all-reduce: Figure 16 shows the effect of different all-reduce execution planning options in our testbed. We see that the optimal execution plan depends on the input message size. The optimal plan at small message sizes is tree all-reduce motif whereas the optimal plan at large message sizes is ring all-reduce motif. Bottom MLP is wider and induces larger (O(100's of MB) vs. top MLPs O(MB)) all-reduce collectives and explains choice of ring all-reduce and tree all-reduce for ar-bottom-mlp and ar-top-mlp, respectively.

Execution planning options for all-to-all:

Figure 17 shows the effect of different all-to-all execution planning options. We note that the optimal plan at small mes-

		TicTac [16]	P3 [18]	Blink [34]	ByteScheduler [29]	Syndicate
Execution	Network Throughput	×	×	✓	✓	✓
	Network Heterogeneity	×	×	✓	×	✓
	Network Ops	Push-Pull	Push-Pull	All-Reduce	Push-Pull; All-Reduce	Send-Recv; Collectives
Scheduling	Preemptible	✓	✓	—	✓	✓
	Models	General DAGs	Layer-by-Layer	—	Layer-by-Layer	General DAGs
	Frameworks	PS	PS	—	PS; ∼ P2P	PS; P2P
	Policy	DAG Optimal	LIFO	—	LIFO	DAG Optimal
Joint Optimization		×	×	—	×	✓

Table 3: Comparison of systems optimizing communication operations for training workloads

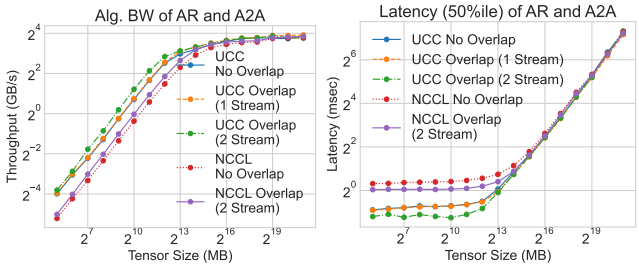


Figure 18: Effect of different execution plans for all-to-all and all-reduce overlap

sage sizes is the 1-way splined motif (simultaneous transfers to 128 destination processes from all source processes), at intermediate message sizes is the 2-way splined motif (64 Processes), and at large message sizes is the 4-way splined motif (32 Processes). The effects of incast are significant as we increase the message sizes and more splining helps reduce incast. DLRM training induces large a2a's (O(GB) message size) and SYNDICATE chooses the 4-way splined execution plan.

Execution planning options for overlap of all-reduce and all-to-all: Figure 18 shows that the optimal execution plan is the one where all-reduce uses NCCL implementation and all-to-all use UCC implementation over non-overlapping communication channels (i.e., 2 streams). With this implementation all-to-all is CPU-driven and uses the PCIe complex and TCP/IP over Ethernet, while all-reduce is GPU-driven and uses NVLink complex and GPUDirect RDMA. This choice is optimal (as opposed to vice versa) as all-reduce also does compute (gradient aggregation) which is faster with GPUs. SYNDICATE uses this execution plan for overlap of all-to-all and all-reduce.

7 Other Related Work

Several works speed-up training by optimizing two main concerns of communication operations: scheduling and execution. Table 3 shows comparison of SYNDICATE against several state-of-the-art systems.

Scheduling concerns looks at reordering communication operations to maximize overlap of compute and communication. The optimal scheduling policy is dependent on factors such as the model architecture, and the parallelization strategy/framework.

Execution concerns look at accelerating individual communication operations through efficient transport over all communication links. These optimizations propose optimal batching to improve link utilization, propose multipath in collectives to make better use of heterogeneous links in the network, and enable preemption to enable scheduling optimizations.

Existing scheduling works fall short in generalizing optimally to all scenarios and they exercise only a subset of optimizations as highlighted in Table 3. Crucially, unlike SYNDICATE, existing works do not jointly optimize both these concerns.

8 Conclusion

We propose SYNDICATE that rethinks communication scheduling granularity and the interfaces in the communication stack for ML training to enable joint optimization of scheduling and execution planning. Using the novel notion of motifs and a split control/data plane architecture SYNDICATE achieves improvements of 21-74% for production scale large-model training as it better utilizes the network multipath opportunities in emerging training clusters.

References

[1] Ai and compute. https://openai.com/blog/ai-and-compute. Accessed: 2021-08-26.

[2] Nvidia collective communication library: Optimized primitives for collective multi-gpu communication. https://github.com/NVIDIA/nccl. Accessed: October 3, 2022.

[3] Parametrized recommendation and ai model benchmark. https://github.com/facebookresearch/param. Accessed: October 3, 2022.

[4] Pytorch kineto. https://github.com/pytorch/kineto. Accessed: 2021-08-26.

[5] Pytorch process group third-party plugin for ucc. https://github.com/facebookresearch/torch_ucc. Accessed: October 3, 2022.

[6] Unified collective communication (ucc). https://ucfconsortium.org/projects/ucc/. Accessed: October 3, 2022.

[7] A. Adcock, V. Reis, M. Singh, Z. Yan, L. van der Maaten, K. Zhang, S. Motwani, J. Guerin, N. Goyal, I. Misra, L. Gustafson, C. Changhan, and P. Goyal. Classy vision. https://github.com/facebookresearch/ClassyVision, 2019.

[8] M. Baines, S. Bhosale, V. Caggiano, N. Goyal, S. Goyal, M. Ott, B. Lefaudeux, V. Liptchinsky, M. Rabbat, S. Sheiffer, A. Sridhar, and M. Xu. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021.

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[10] M. Cho, U. Finkler, D. Kung, and H. Hunter. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *Proceedings of the 2nd SysML Conference*, 2019.

[11] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. Panda. NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, 2020.

[12] A. Desai, L. Chou, and A. Shrivastava. Random Offset Block Embedding Array (ROBE) for CriteoTB Benchmark MLPerf DLRM Model : 1000× Compression and 2.7× Faster Inference, 2021.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[14] N. Goyal, J. Du, M. Ott, G. Anantharaman, and A. Conneau. Larger-scale transformers for multilingual masked language modeling. *arXiv preprint arXiv:2105.00572*, 2021.

[15] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 81–97, 2016.

[16] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[17] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[18] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019.

[19] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019.

[20] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 463–479, 2020.

[21] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[22] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query re-planning using {QOOP}. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 253–267, 2018.

[23] Message Passing Interface Forum. http://www.mpi-forum.org/. Accessed: October 3, 2022.

[24] D. Mudigere, Y. Hao, J. Huang, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. R. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao. High-performance, distributed training of large-scale deep learning recommendation models. *CoRR*, abs/2104.05158, 2021.

[25] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[26] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[27] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2011. Accessed: October 3, 2022.

[28] NVIDIA. DGX A100 System User Guide. https://docs.nvidia.com/dgx/pdf/dgxa100-user-guide.pdf, 2021. Accessed: October 3, 2022.

[29] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[30] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[31] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, et al. Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[32] M. Smelyanskiy. Zion: Facebook next-generation large memory training platform. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–22. IEEE Computer Society, 2019.

[33] Y. Ueno and R. Yokota. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 430–439, May 2019.

[34] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica. Blink: Fast and generic collectives for distributed ml. *arXiv preprint arXiv:1910.04940*, 2019.

[35] J. Yin, S. Gahlot, N. Laanait, K. Maheshwari, J. Morrison, S. Dash, and M. Shankar. Strategies to deploy and scale deep learning on the summit supercomputer. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 84–94, 2019.

A Appendix

A.1 Transformation Operator Algebra

We now present the algebra for the motif transformation operators. We denote the segmentation operator by $\stackrel{s}{=}$ and the splining operator by $\stackrel{p}{=}$. Note that the algebraic rules presented below are not exhaustive and are extensible. Here, we present the algebraic rules that we use in the context of DLRM to transform all-reduce and all-to-all collectives into motifs. We first go over the various symbols used in the algebra.

$\texttt{N}$: Total number of Processes

$\texttt{PG[0:N]}$: Process $\texttt{IDs}$ involved in a Motif

$\texttt{T}_i\texttt{[0:D]}$: Tensor of size $\texttt{D}$ on Process $\texttt{P}_i$

$\texttt{T}_i\texttt{[0:N,0:D]}$: $\texttt{N}$ Tensors of size $\texttt{D}$ on Process $\texttt{P}_i$ with first dimension indicating destination Process $\texttt{ID}$

$\|$: Parallel Execution

$\rightarrow$: Sequential Execution

$\stackrel{s}{=}$: Segmentation Transformation

$\stackrel{p}{=}$: Splining Transformation

$\texttt{AR}$: $\texttt{all-reduce motif}$

$\texttt{AA}$: $\texttt{all-to-all motif}$

$\texttt{RE}_r$: $\texttt{reduce motif}$ with root Process $\texttt{P}_r$

$\texttt{RS}$: $\texttt{reduce-scatter motif}$

$\texttt{BC}_r$: $\texttt{broadcast motif}$ with root Process $\texttt{P}_r$

$\texttt{AG}$: $\texttt{all-gather motif}$

$\texttt{COLL(T}_i\texttt{[:], PG[IDs])}$: Motif $\texttt{COLL}$ with input tensor $\texttt{T}_i$ executing on each Process $\texttt{P}_i$ for all i in $\texttt{IDs}$

We now present the algebraic rules for transforming the all-reduce motif using the segment and spline operators.

Segmented All-Reduce: First, we show application of the segment operator which splits the input tensor at all the processes and converts an all-reduce motif into smaller all-reduce motifs over the splits. Each smaller all-reduce motif are independent and can execute at the same time in parallel.

$$\texttt{AR(T}_i\texttt{[0:D], PG[0:N])} \stackrel{s}{=} \|_{s=0}^{\frac{D}{d}-1} \texttt{AR(T}_i\texttt{[s*d:(s+1)*d], PG[0:N])}$$

Ring All-Reduce: Next, we show an instance of the spline operator that divides the pattern in original all-reduce into two sub-patterns: reduce-scatter motif followed by the all-gather motif. The reduce-scatter motif does aggregation and the all-gather motif broadcasts the aggregated result. The reduce-scatter and all-gather motifs induce a pattern of communication over a ring, where the processes are arranged in a ring and the tensor is divided into $\texttt{N}$ pieces. Each process $\texttt{P}_i$ does a point-to-point transfer of the $\texttt{(i+r)\%N}$ piece to its neighboring process in the ring in the $\texttt{r}^{th}$ round for $\texttt{N}$ rounds.

$$\begin{aligned}
\texttt{AR(T}_i\texttt{[0:D], PG[0:N])} &\stackrel{c}{=} \texttt{RS(T}_i\texttt{[0:D], PG[0:N])} \\
&\quad \rightarrow \texttt{AG(T}_i\texttt{[0:D], PG[0:N])} \\
\texttt{RS(T}_i\texttt{[0:D], PG[0:N])} &= \texttt{ring pattern of communication} \\
\texttt{AG(T}_i\texttt{[0:D], PG[0:N])} &= \texttt{ring pattern of communication}
\end{aligned}$$

Tree All-Reduce: Next, we show an instance of the spline operator that divides the pattern in the original all-reduce into three smaller sub-patterns: reduce motif followed by a smaller all-reduce motif followed by broadcast motif. The same spline operator algebraic can be recursively applied to the smaller all-reduce motif. Recursive application results in a hierarchical tree pattern of communication where several reduce motifs first aggregate results in a tree like fashion at a single root process and several broadcast motifs broadcast the aggregated result from the root process in a tree like fashion until it is updated at all the processes. Each reduce motif results in a convergent pattern of communication where all the processes involved in the reduce send their tensors to the root process where it is aggregated. Each broadcast motif results in a divergent pattern of communication where the root process sends its tensor to all the processes involved in the broadcast motif.

$$\begin{aligned}
\texttt{AR(T}_i\texttt{[0:D], PG[0:N])} &\stackrel{c}{=} \|_{c=0}^{\frac{N}{n}-1} \texttt{RE}_{c*n}\texttt{(T}_i\texttt{[0:D], PG[c*n:(c+1)*n])} \\
&\quad \rightarrow \texttt{AR(T}_i\texttt{[0:D], PG[}\cup_{c=0}^{\frac{N}{n}-1} \texttt{c*n])} \\
&\quad \rightarrow \|_{c=0}^{\frac{N}{n}-1} \texttt{BC}_{c*n}\texttt{(T}_i\texttt{[0:D], PG[c*n:(c+1)*n])} \\
\texttt{RE}_j\texttt{(T}_i\texttt{[0:D], PG[j:j+n])} &= \texttt{convergent pattern of communication} \\
\texttt{BC}_j\texttt{(T}_i\texttt{[0:D], PG[j:j+n])} &= \texttt{divergent pattern of communication} \\
\texttt{AR(T}_i\texttt{[0:D], PG[}\cup_{c=0}^{\frac{N}{n}-1} \texttt{c*n])} &= \text{recursive application of } \stackrel{c}{=} \text{ induces tree pattern of communication}
\end{aligned}$$

Segmented and Splined All-To-All: Next, we show examples of segmenting and splining an all-to-all collective into smaller motifs. With segmentation, the tensor at all the processes is split and the original all-to-all is deconstructed into several smaller all-to-all motifs over the split tensors. With splining, the pattern of communication in the original all-to-all motif with a clique of point-to-point transfers between all the processes is broken down into smaller all-to-all motifs with smaller patterns where each process $\texttt{P}_i$ initiates point-to-point transfers to a subset of destination processes (with ids in the range $\texttt{(i+c*n)\%N:(i+(c+1)*n)\%N}$). Here, $\texttt{n}$ parameterizes the all-to-all splining operator with larger $\texttt{n}$ resulting in breaking the original all-to-all into fewer all-to-all motifs with larger sub-patterns.

$$\texttt{AA(T}_i\texttt{[0:N,0:D], PG[0:N])} \stackrel{s}{=} \|_{s=0}^{\frac{D}{d}-1} \texttt{AA(T}_i\texttt{[0:N, s*d:(s+1)*d], PG[0:N])}$$

$$\texttt{AA(T}_i\texttt{[0:N,0:D], PG[0:N])} \stackrel{c}{=} \|_{c=0}^{\frac{N}{n}-1} \texttt{AA(T}_i\texttt{[(i+c*n)\%N:(i+(c+1)*n)\%N, 0:D], PG[0:N])}$$

Addax: A fast, private, and accountable ad exchange infrastructure

Ke Zhong* Yiping Ma* Yifeng Mao* Sebastian Angel*†
*University of Pennsylvania †Microsoft Research

Abstract. This paper proposes Addax, a fast, verifiable, and private online ad exchange. When a user visits an ad-supported site, Addax runs an auction similar to those of leading exchanges; Addax requests bids, selects the winner, collects payment, and displays the ad to the user. A key distinction is that bids in Addax's auctions are kept private and the outcome of the auction is publicly verifiable. Addax achieves these properties by adding public verifiability to the affine aggregatable encodings in Prio (NSDI'17) and by building an auction protocol out of them. Our implementation of Addax over WAN with hundreds of bidders can run roughly half the auctions per second as a non-private and non-verifiable exchange, while delivering ads to users in under 600 ms with little additional bandwidth requirements. This efficiency makes Addax the first architecture capable of bringing transparency to this otherwise opaque ecosystem.

1 Introduction

Ad exchanges such as DoubleClick and OpenX are key players in online advertising; their role is to auction ad space on a publisher's website in real time to advertisers. When a user visits a publisher's page, the user's browser contacts a server that triggers an auction on an exchange. The exchange gives advertisers information about the publisher (e.g., URL, ad size and type, category of site) and the user (e.g., demographic, metadata for syncing cookies across sites) in real time, and collects bids from interested parties. The exchange then runs an auction (e.g., second-price auction), delivers to the user the ads of the winning advertisers, and credits the publisher. Finally, technologies like header bidding [4] and Google's open bidding platform [30] allow publishers to auction users across many exchanges (essentially an exchange of exchanges), increasing competition and improving publishers' revenue.

While ad networks and exchanges serve as the financial backbone of the free web, their centralized nature means that: (1) they are privy to sensitive information, including user's browsing habits and the preferences and valuations of advertisers; and (2) they are opaque and hard to audit. The former has received considerable attention [39, 45, 56, 71, 77, 89, 91]; the lack of auditing mechanisms and the knowledge of advertisers' valuations is becoming a serious sociotechnical issue. A recent antitrust lawsuit alleges that Google used insider knowledge of past bids submitted by advertisers to gain unfair advantages whenever its subsidiaries participated in auctions [63]. Further, it is alleged that Google convinced Facebook to not participate in header bidding—a technology considered an "existential threat" to Google's business [28, 81, 84]. According to disclosed reports, in return for Facebook choosing to participate instead in Google's open bidding platform, "Google provided Facebook with special information and speed advantages to help [Facebook's exchange] succeed in the auction [over other bidders]—even including a guaranteed win rate" [84].

Regardless of the merits of these cases, the key issue—and the crux of this paper—is that there are no ways for exchanges to *prove* to their customers and to regulators that they are not abusing their position. To address this, we present Addax, an online advertising architecture that achieves 4 goals:

- *Auction integrity.* Auctions should be publicly verifiable to allow the ad exchange to prove that it is not biasing auctions towards particular bidders or lying about their outcome.
- *Auction privacy.* The bids of losing bidders should be hidden from all parties—even the exchange itself! This ensures that the exchange cannot abuse or share this information.
- *High performance.* Addax should handle the stringent performance requirements of the ad ecosystem.
- *Better tracking.* Addax should work with recent tracking efforts such as Google's Topics API [16] and Microsoft's PARAKEET [13] that allow targeted ads but collect less information about individuals.

Overview. In Addax, browsers track users' histories with existing privacy-preserving client-side techniques [57, 58, 80, 89], and kickstart verifiable and private auctions whenever the user navigates to an ad-supported site. Auctions in Addax proceed in three steps. First, the browser invites relevant bidders (e.g., demand-side platforms) by finding their information (e.g., URL of their ad server) on a database. In existing header bidding platforms [2] such databases are currently maintained by publishers; Addax preserves this model, but we additionally experiment with a more decentralized approach where the database is maintained in a public append-only log and discuss how to reduce the cost of lookups in this model (§7). As part of the invitation, the browser supplies to bidders information about the site being visited and a variable amount of user information based on the user's configuration of Addax (ranging from fully targeted to generic ads).

Second, bidders submit encrypted bids to the publisher and one or more *auxiliary servers*. The auxiliary server helps the publisher run a new lightweight secure auction computation over the encrypted bids (§4). The role of an auxiliary server could be taken up by today's exchanges or it can be a separate entity propped up by the industry at large. Under the *anytrust model* [88] (either the publisher or any of the auxiliary servers is honest), the secure auction computation returns the winning bidder's identity and bid, and the auction's sale price, but no other information.

Last, Addax produces an *audit trail* that is uploaded to a public log and that allows any auditor to verify that the auction was conducted with integrity (§5). At the conclusion of the auction, the browser fetches the ad from the winning bidder (or a content distribution network) and the publisher learns which bidder and how much to bill for the ad impression.

Technical contributions. To maintain good performance, Addax cannot use expensive cryptography (e.g., homomorphic encryption, multiparty computation) in order to achieve the integrity and privacy goals. Indeed, our evaluation of such baselines confirms that they are far too inefficient to meet online advertising's low latency and communication requirements (§9). Instead, Addax makes three contributions:

Secure auction protocol. Addax introduces a new auction protocol based on Prio's *affine-aggregatable encodings* (AFE) [51]. Addax's auction is simple and lightweight and allows two or more parties to run the auction over secret-shared bids without revealing anything beyond the auction's outcome.

Verifiable AFEs (V-AFEs). Addax extends Prio's AFEs to provide public verifiability for *outputs* (Prio has mechanisms to verify inputs). Addax then uses V-AFEs to allow anyone (e.g., an auditor) to confirm that an auction was conducted correctly without learning any of the input bids.

Integration with Algorand and Chrome. Addax implements mechanisms to interact quickly with the public log (we use Algorand [3, 55]) and smart contracts to manage the registration of advertisers and the collection of audit materials. Addax also leverages Chrome native messaging to launch auctions.

Our implementation of Addax can complete auctions over WAN with twice the average number of bidders reported in production ad exchanges [92] in 440–580 ms (for first and second-price auctions), and requires only 1.2 MB of communication between the publisher and the auxiliary server (§9.2). This is fast enough for ads to be loaded asynchronously without affecting page load time for the overwhelming majority of websites today [1, 7, 20, 87]. In terms of throughput, Addax can handle around 250–360 auctions per second per core (for second and first-price auctions), which is roughly 40% of what our non-private and unverified baseline can achieve. Creating the audit trail requires additional computation on the part of bidders but adds negligible overhead to users' browsers and publishers. In contrast, the same auction implemented in existing state-of-the-art cryptographic frameworks (MPC and FHE) requires over 4 GB of communication and over 100 sec.

Limitations. Ad exchanges do more than just run auctions and deliver ads. They vet advertisers to ensure users do not receive malware; mitigate fraud; and provide powerful analytics. Addax does not yet address these complementary and critical aspects, but Section 11 discusses concrete directions to incorporate such features into Addax's architecture. Finally, Addax can achieve better performance (optionally) at the expense of revealing the existence of winning ties (§4.4).

2 Background and goals

Ad exchanges are platforms that auction *impressions* (the display of a text, image, or video ad) on a publisher's website or mobile application in real time. Exchanges support highly targeted advertising whereby bidders (advertisers or their representatives, called demand-side platforms) get a chance to evaluate the publisher and the user to whom the ad will be shown to decide how much they would be willing to pay (if at all). This type of programmatic *real-time bidding* (RTB) advertising accounts for over a third of all digital ad spending today [22, 27]. Some of the largest ad exchanges include DoubleClick, PubMatic, OpenX, and Facebook.

To participate in an ad exchange, a publisher inserts a *supply-side platform*'s (SSP) iframe or JavaScript snippet into their page. An SSP is a service that sells the publisher's ads on an exchange (publishers can also run their own SSP). When a user's browser fetches the publisher's site and executes the provided JavaScript, it sends an HTTP GET request to the SSP supplying the user's cookie, and awaiting for an ad to be returned. At this point, the SSP can identify the user and publisher, and start an RTB auction. During this process, the exchange invites dozens of potentially interested bidders to bid on the user [92], supplying them with demographic information, and relevant details about the publisher and the ad space (size, type, location within the page). To facilitate the valuation of the user, exchanges and bidders synchronize cookies [19, 23] to allow bidders to learn the identity of the user in their respective platforms (if applicable). Based on this information, bidders return a bid in CPM (cost per 1000 impressions), which ranges from cents to tens of dollars [32].

Upon receiving all bids, the exchange runs an auction where it selects the winning bidder and charges them the auction's sale price based on the type of auction. Two common types are *first-price* (winner pays what they bid) and *second-price* [83] (winner pays second highest bid) auctions. Finally, the exchange notifies the SSP with the result of the auction, who then responds to the user's GET requests with the information that the browser needs to retrieve the ads (images, videos, etc.) from a storage server.

2.1 Header bidding

Header bidding [4] is a recent advertising paradigm where the publisher (or its SSP) works with multiple exchanges to sell its ad slot in real time. It is called "header bidding" because the publisher supplies JavaScript code that runs in the `<header>` part of the page (which loads as soon as the page starts opening in the user's browser), and this code triggers the process of contacting the exchanges. The exchanges then internally run their own auctions (first or second-price) and send back the winning bids to the browser. The browser then sends the winning bids to the publisher (or its SSP), which runs another auction (typically first-price), selects the highest bid as the winner, and forwards the winner's ad tag to the user's browser. Google's Open Bidding platform is similar [30].

2.2 Concerns with existing exchanges

We highlight three areas of concern with existing ad exchanges. First, there is no visibility into the auction process. VEX [35] argues that this opens the door to a variety of issues—including those that are mentioned by the antitrust lawsuits [28, 63, 84]. Second, exchanges observe all submitted bids in the clear. These bids represent how valuable different users and publishers are to bidders, which reveals information about bidder's trading algorithms, finances, and future plans. Last, users lack agency and have no say over which types of ads they receive or what information is shared with bidders. One might imagine a different world in which users can express an opinion on the types of ads they consume (e.g., no ads for kids toys to avoid children exploitation), and what information about themselves they reveal in order to receive targeted ads.

2.3 Goals

Addax aims to address many of the shortcomings of existing ad exchanges by giving agency to users, privacy to bidders, and transparency to all. Addax is compatible with both traditional exchanges and with the header bidding model (including Google's open bidding platform). We detail these goals next.

Integrity of the auction. All parties should be able to verify that Addax's auctions are conducted correctly, as per the auction type (first-price, second-price, etc.).

Privacy for losing bids. Addax should hide the bids of all of the losing bidders from *everyone*, even the auctioneers. One exception is that in second-price auctions, the second highest bid (which is technically a losing bid) becomes the sale price and cannot be hidden.

Privacy among bidders. Bidders should not need to learn each others' identities or interact with one another in order to participate in an auction. Existing exchanges do not reveal this information, and neither should Addax.

High performance. Addax must ensure that auctions complete quickly, as ads need to be displayed within hundreds of milliseconds in order to preserve a good user experience and follow existing RTB requirements [10, 11].

User agency. Addax's focus is on making the auction process accountable without exposing bidders' information. Addax should also allow users to have a say on which kinds of ads they wish to receive. Ideally, Addax would also improve user privacy, but this is not a goal of this work. Instead, we ask that Addax make things no worse than they are today for users, and that it be compatible with other works that aim to reduce user tracking (such as Topics [16]). Appendix G expands on this compatibility aspect.

2.4 Potential solutions (baselines)

Given Addax's desire for privacy and verifiability, one might ask whether existing tools such as homomorphic encryption or multiparty computation fit the bill. This is not the case.

Homomorphic encryption (HE). HE libraries [24–26, 61] allow the computation of additions and multiplications over encrypted data without access to plaintext values. Computing an auction, however, requires comparisons (such as "less than") which are expensive to express with arithmetic operations as they typically require decomposing values into bits and encrypting bits separately [25, 50]. Even recent optimizations are expensive [41, 48, 64]. As we show in our evaluation (§9.2) an auction with 96 bidders using the state-of-the-art TFHE library [49, 50] takes 181 seconds. Finally, HE lacks integrity: an auctioneer is free to compute an incorrect auction. Recent work on composing verifiable computation with HE can address this, but at orders-of-magnitude cost increase [40, 54].

Secure multi-party computation (MPC). MPC frameworks [31, 65, 85] allow mutually distrusting parties to compute a function over secret inputs without revealing anything beyond the function's outcome. It might seem natural to encode the auction as an MPC among the bidders but this is impractical when there are many bidders. An alternative is to use a *delegated* MPC setting whereby two parties (publisher and auxiliary server in our setting) run the MPC on behalf of others; bidders could send secret shares of their bids to these two parties. However, this delegated setting lacks integrity: either party is free to supply bogus shares to the MPC to cause the auction's output to be *undetectably* incorrect. As we show in Section 9.2, addressing this introduces prohibitive costs.

Trusted execution environments (TEEs). Another possibility is to use trusted hardware. Besides side channel [43, 46, 68, 90] and integrity attacks [72, 82], TEEs alone cannot solve this problem. Appendix F discusses this in depth.

3 Addax Overview

Addax is a platform where the exchange's duties are split among different parties. Figure 1 gives a high-level description. Addax consists of: (i) publishers who run their own SSP and who wish to show ads to fund their services, (ii) the client's browser, (iii) auxiliary servers who help to run auctions, (iv) bidders (demand-side platforms, advertisers, other exchanges, etc.) who bid on ad slots, and (v) an append-only log (e.g., blockchain, BFT consortium) for persisting an audit trail. We discuss what happens when a user visits a page below, and give details in the sections that follow. We defer a discussion of how bidders join Addax and what information they supply to Section 7 and Appendix E.

Steps ①–②: Client visits a publisher. When a client visits a publisher, it receives the page content, along with a unique *auction id* and a list of valid ad categories that the publisher supports. Addax uses the 392 categories from the Internet Advertising Bureau's (IAB) contextual taxonomy [10], which include things like "Humor", "Nutrition", etc. This metadata is embedded within the header of the page, as in header bidding (§2.1). An Addax-enabled browser, hereafter named "browser", parses the web page and extracts this metadata.

Step ③: Advertising filtering. Addax adopts a client-based tracking approach inspired by Privad [58], Adnostic [80], and Google's recent FLoC proposal [89]. Briefly, the browser tracks which sites the user visits over time and generates a profile of the user's interests, which it stores locally in a SQLite database similar to how cookies are stored. After parsing a page's ad spot metadata, the browser combines the user's profile, the ad spot's categories supplied by the publisher, and disallowed categories previously flagged by the user through a local configuration (e.g., to prevent categories that target children). Based on the refined information, the browser fetches bidders' details from a bidder database. Addax supports two types of databases: an embedded database supplied by the publisher during Step ②; and a public database where bidder information is maintained on the public log (blockchain). The former is how header bidding works today while the latter option is more decentralized and gives users more agency over the ads they receive. We defer the details to Section 7.

Steps ④–⑥: Private, decentralized, and verifiable auction. The browser invites the k bidders from step ③ to an auction. To do so, it provides them with an *auction id* (unique identifier supplied by the publisher), information about the user, and information to contact the publisher and the auxiliary server. Bidders decide whether to join the auction; if so, they respond to the auxiliary server and publisher with their required materials. The auxiliary server and the publisher collaboratively run the auction and select the auction's winner, and the auction's sale price. Asynchronously, and off the critical path, all participants upload to the public append-only log materials needed for public auditing (§5).

Step ⑦: Notify winner and display the ad. After the auction concludes, the publisher and auxiliary server learn the outcome (but nothing else). The publisher notifies the winner and asks it for an ad tag and payment (e.g., a signed IOU). The publisher then forwards the ad tag to the browser so that it can fetch and display the ad on the designated ad spot.

Verification. Auditors can use the information on the public log to verify the auction's outcome. By default, they only learn whether the auction was correct and the number of bidders that participated. In case that verification fails, Addax helps narrow down which parties were faulty (§5.3).

3.1 Assumptions and threat model

Addax assumes an append-only log (blockchain, BFT, etc.) and an *anytrust* model [88] where either the publisher or the auxiliary server is honest. The parties may act as follows.

Bidders. Bidders who are invited to the auction can submit bogus bids and cryptographic material. We model bidders as *covert adversaries* [36] who can deviate from the protocol arbitrarily as long as their malicious actions cannot be detected. If detected, bidders can incur financial or legal penalties, and can be banned by publishers. Addax assumes at least 2 non-colluding losing bidders (otherwise information about losing bids can be inferred from the outcome).

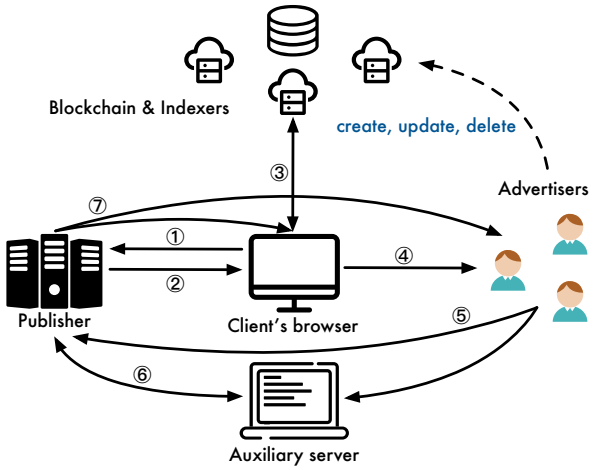


FIGURE 1—In Addax, the exchange's functionality is divided among the publisher, browser, an auxiliary server and a blockchain.

Publishers and auxiliary servers. Publishers may wish to increase their revenue by lying about the auction's outcome (e.g., forcing the winner to pay a fee higher than the second highest bid), learn the bids of losing bidders, or force users to view certain ads. Auxiliary servers may wish to bias the auction's result to help particular bidders. We model both parties as covert adversaries since detectable misbehavior can tarnish their reputation or incur legal penalties.

3.2 Security properties

Addax's auction protocol provides the following properties.

Completeness. If all parties are honest and the auction's outcome is correct (e.g., the winner is the highest bidder and the sale price is the second highest bid), then Addax's verification protocol passes with high probability.

Soundness. If a bidder, the publisher, or an auxiliary server misbehaves, Addax's verification fails with high probability.

Privacy. Addax's auction and verification hides all bids except the highest bid and the sale price.

4 Private ad auction

This section describes Addax's private ad auction. We begin by describing our building blocks.

4.1 Affine-aggregatable encodings (AFE)

Prio [51] shows how one can take two or more data values and encode each of them as a vector of λ bits such that adding up the vectors and running a decoding function on the sum is equivalent to computing some boolean function `f` (e.g., OR, AND, XOR) on the original data values; λ is a parameter that controls the probability of the result being correct. Prio calls this and other similar transformations an *Affine-Aggregatable Encoding* (AFE). Addax uses the "OR" boolean function to compute auctions, so it could use Prio's AFE. However, we

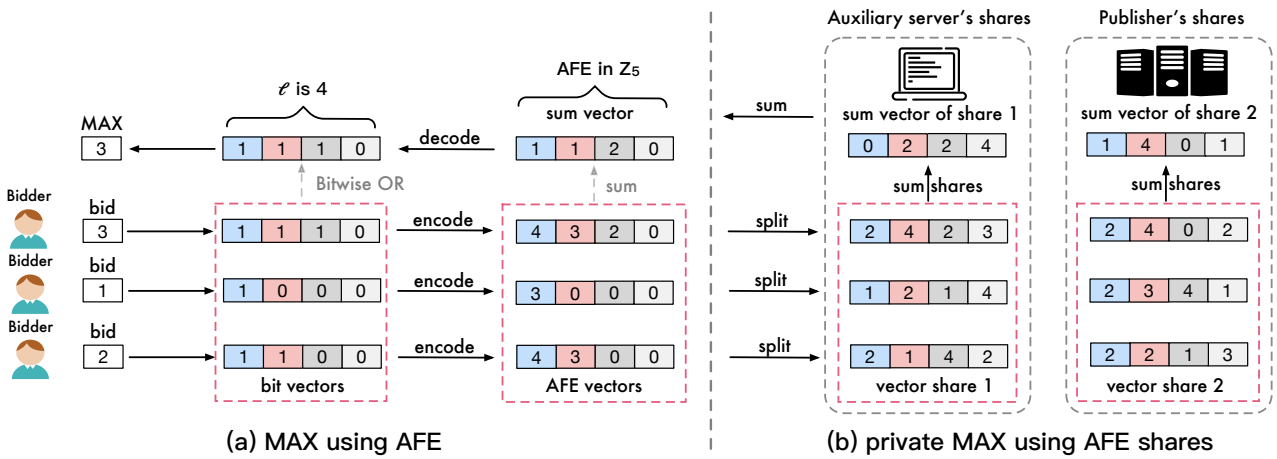


FIGURE 2—Example of (a) the MAX algorithm using AFE, and (b) the private MAX algorithm using AFE shares. In this example, $\ell = 4$ (which affects the range of bids) and we use AFE in $\mathbb{Z}_5$ (which affects the probability of obtaining the correct result).

depart slightly from Prio by encoding data values into a single element in $\mathbb{Z}_p$ (the set of integers modulo a large prime p) rather than λ-sized bit vectors. This encoding is more expensive than Prio's (since $\lambda < \log p$), but it allows Addax to add public verifiability, as we discuss in Section 5.1. Below we give our AFE for the "OR" function over bit values.

Encode OR. Given a bit $x \in \{0,1\}$, its AFE is:

$$\text{Encode-OR}(x) = \begin{cases} 0 \in \mathbb{Z}_p & \text{if } x = 0 \\ \text{a random element} \in \mathbb{Z}_p & \text{if } x = 1 \end{cases}$$

Compute OR. Given a set of n AFE values $\{v_1, \ldots, v_n\}$, which encode n bits $\{x_1, \ldots, x_n\}$ with the above Encode-OR procedure, one can compute the OR of the n bits as:

$$\mathbf{v} = v_1 + \cdots + v_n \in \mathbb{Z}_p$$

Decode OR. Given the sum AFE value $\mathbf{v}$, one can correctly recover the result of the OR operation over the underlying n bits with probability of at least $1 - 1/p$ as follows.

$$\text{Decode-OR}(\mathbf{v}) = \begin{cases} 0 & \text{if } \mathbf{v} = 0 \\ 1 & \text{otherwise} \end{cases}$$

To see why Decode-OR returns the correct value with probability $1 - 1/p$, we consider two cases. First, when all n input bits are 0. In this case, all AFE values are zeros so $\mathbf{v}$ is guaranteed to be zero; Decode-OR always outputs the correct value of 0. Second, when at least one of the n input bits is 1. In this case, since the value is independent and uniformly random, the probability that the sum in $\mathbb{Z}_p$ is zero is $1/p$.

4.2 Computing the MAX function with AFE

Following the approach in Prio, we show how to extend the above AFE to support *MAX*, which Addax uses to find the highest bid in an auction. This construction provides neither privacy nor verifiability; we add these later.

Suppose all input values are integers in the range $[0, \ell]$. Each input x is first represented in unary. That is, as a bit vector β of length ℓ $(\beta_1, \beta_2, \ldots, \beta_\ell)$ where $\beta_i = 1$ if and only if $i \leq x$. Observe that if we perform a bitwise OR on the unary bit vectors of all inputs, the result will be a unary bit vector where the index of the last "1" represents the maximum value across all inputs. This is the idea behind the AFE of MAX.

Encode MAX. Given a value $x \in [0, \ell]$, its AFE is a vector of ℓ values, where each value is an element in $\mathbb{Z}_p$. The encoding happens in two steps: (1) represent x as a bit vector β of length ℓ in unary format; and (2) for each bit β_i, encode β_i with the Encode-OR of Section 4.1. The result is a vector M with ℓ values, where $M[i]$ is the AFE value of bit β_i.

Compute MAX. Given n AFE vectors $\{M_1, \ldots, M_n\}$ that encode the values $\{x_1, \ldots, x_n\}$ as above, MAX is computed by adding the n vectors: $\mathbf{M} = M_1 + \cdots + M_n$.

Decode MAX. Given the sum vector $\mathbf{M}$, one can recover the MAX of the underlying n values in two steps. First, use Decode-OR (§4.1) on each of the ℓ entries of $\mathbf{M}$. The result is a bit vector β of length ℓ in unary format. Second, output the highest index j for which β_j is 1. This value is the correct MAX among the n inputs if Decode-OR outputs the correct OR for all ℓ bits. This event occurs with probability $\geq (1 - 1/p)^\ell$.

Figure 2(a) gives an example of these procedures with three inputs. Below we describe how to add privacy by secret sharing the AFE vectors among multiple parties.

4.3 Private and decentralized MAX

Observe that computing the MAX of n values $(x_1, \ldots, x_n)$ using AFE vectors $(M_1, \ldots, M_n)$ requires only additions. We can split each vector M_i into two shares (M_i^1 and M_i^2) that add up to the original ($M_i = M_i^1 + M_i^2$) as depicted in Figure 2(b). Each share is made up of uniform random elements in $\mathbb{Z}_p$, and reveals no information about M_i without the other share.

Suppose that two non-colluding parties, Alice and Bob, are

tasked with computing the MAX of n values given n AFE vector shares. Alice receives $\{M_1^1, \ldots, M_n^1\}$ and Bob receives $\{M_1^2, \ldots, M_n^2\}$. Each party sums up their n shares to get two sum vector shares: M^1 for Alice and M^2 for Bob. Finally, both parties exchange their sum vector shares. Observe that by adding M^1 and M^2, the parties can recover the sum vector $\mathbf{M} = M^1 + M^2 = M_1 + \cdots + M_n$ as shown in Figure 2(a), and then use Decode-MAX to recover the max value.

4.4 Private and decentralized auction

We use the private MAX of Section 4.3 to compute an auction where the auctioneer's duty is split between the auxiliary server and the publisher. This protocol provides privacy but not integrity (i.e., malicious actions can lead to an incorrect outcome); we add verifiability in Section 5. A bid is given by the position of the last "1" in a unary vector (e.g., [1,1,1,0] and [0,0,1,0] both represent 3, though the latter is ill-formed). We assume a maximum bid ℓ, and everyone bids within $[0, \ell]$.

Step 1: Set up shared secret. Before the auction starts, the publisher and the auxiliary server commit to a random secret to be used later as an unbiased source of randomness. Concretely, when the client visits the publisher, the publisher contacts the auxiliary server, notifies it of an incoming auction, and supplies to it a commitment to a uniform random secret, $secret_p$. The auxiliary server replies with its own commitment to a uniform random secret, $secret_a$. They keep these secrets hidden until Step 4.

Step 2: Encode and split bids. The browser sends an invitation to selected bidders with the auction's id. If a bidder wishes to participate, they encode their bid using the Encode-MAX procedure (§4.2), split the resulting vector into two additive shares as discussed in Section 4.3, and generate a fresh signing and verification key pair. The verification key acts as the bidder's *bidder id* in the auction. The bidder then sends share M^1 to the auxiliary server and share M^2 to the publisher, supplying both with the bidder id. Bidders who fail to submit their shares before a timeout are kept out of the auction.

Step 3: Find the highest bid. Before computing the auction, the auxiliary server sends to the publisher all the bidder ids that it received, the publisher matches them with the ids that it received, and responds to the auxiliary server with the intersection. The publisher and the auxiliary server then use the vector shares of bidders in the intersection in the MAX protocol of Section 4.3. If Decode-MAX produces an invalid unary vector such as [1,0,1,0], the auction is aborted; when parties are honest, abort happens with negligible probability (§4.2). At the end, both parties learn the highest bid, b^*, but nothing else. To avoid parties adapting their sum vector share in response to the other's sum vector share, parties first exchange commitments of their sum vector share; the honest party aborts if misbehavior is detected.

Lemma 1. Let $b_1, \ldots, b_j$ be the bids from j honest bidders, and let $M_1, \ldots, M_j$ be the AFE vectors resulting from running Encode-MAX on the bids. Similarly, let $M_{j+1}, \ldots, M_{j+k}$ be AFE vectors that MAX-encode bids from k malicious bidders (these AFE vectors can represent invalid unary vectors like [1,0,1,0]). Decode-MAX on the sum of these $j+k$ AFE vectors outputs, with high probability, either an invalid value (invalid unary representation) or a value $\geq max(b_1, \ldots, b_j)$.

Lemma 2. Let M^1 and M^2 be sum vector shares held by the auxiliary server and publisher, respectively. During Decode-MAX, if the auxiliary server uses a different sum vector share M'^1 without having seen M^2 first or the publisher uses a different sum vector share M'^2 without having seen M^1 first, then the output of Decode-MAX is, with high probability, either an invalid value (invalid unary representation) or ℓ.

Appendix A gives proofs for both lemmas. Together they imply that malicious actions by participants lead to the resulting highest bid being invalid or at worst larger than the real highest bid. Either outcome leaks no information about honest losing bidders' bids to the attacker (our privacy goal). Furthermore, malicious actions are detected by Addax's verification.

Step 4: Find the winner. The publisher and the auxiliary server find the *winner* (the bidder id of the party who submitted b^*) interactively. First, both parties decommit to the secrets they generated in Step 1, check the decommitment, and XOR the secrets together to obtain $secret = secret_a \oplus secret_p$. Since at least one party is honest, *secret* is uniformly random and independent of the bidder ids generated by the bidders; the parties use *secret* as the seed to a pseudorandom generator (PRG). Both parties locally use the PRG to pick the same random bidder w from the set of participating bidder ids, which avoids biasing the auction towards a particular bidder in the case of ties (the PRG is for fairness not for privacy). The auxiliary server sends the b^*-th value of its share of bidder w's vector, $M_w^1[b^*]$, to the publisher and the publisher sends $M_w^2[b^*]$. Both parties then locally sum the two shares to obtain $M_w[b^*] = M_w^1[b^*] + M_w^2[b^*]$. Applying Decode-OR to $M_w[b^*]$ yields β_{b^*}, which is the bit of bidder w at position b^* in the unary vector (§4.2). If β_{b^*} is 0, bidder w is not the winner (since its bid must be lower than b^*). Note that learning β_{b^*} reveals no additional information. The publisher and the auxiliary server continue to pick a random bidder id w until the bit β_{b^*} of w is 1 ($n/2$ tries in expectation). In such a case, w is the winner. Finally, the auxiliary server and publisher ask w if its bid is b^*. Bidder w replies only if it receives the same query from both parties. If w's bid is not b^*, it sends *abort* to both parties and the auction is aborted. If there are ties (i.e., multiple bidders submitted b^*), this procedure returns a uniformly chosen one. The ids of other tied bidders remain hidden.

Lemma 3. In Step 4, if the auxiliary server sends to the publisher an AFE share that is different than what it received from the candidate winner w (i.e., different from $M_w^1[b^*]$) or the publisher sends to the auxiliary server an AFE share that is different than what it received from w, then the auction aborts

or w is declared the winner with high probability. In the latter case, w is either the real winner or a malicious bidder.

Appendix A proves this lemma. It basically means that a malicious publisher or auxiliary server can only ever make a colluding bidder the winner; they cannot cause a winner (if chosen by the PRG) to lose, nor can they make an honest losing bidder the winner (and hence learn its bid).

Step 5: Compute the sale price. The above four steps are sufficient to compute first-price auctions (the most common type) where the winner is the highest bidder and the sale price is its bid. To support second-price auctions (sometimes used by exchanges), the auxiliary server and publisher subtract the winning bidder's vector share from the sum vector share (e.g., the auxiliary server subtracts M_w^1 from M^1). They then rerun Step 3 to obtain the second highest bid.

Lemma 4. If either the auxiliary server or the publisher misbehaves in Step 5, or a malicious bidder is declared the winner in Step 4, then the computed sale price is, with high probability, either: (1) the highest bid among all bidders; (2) the second highest bid among all bidders; or (3) ℓ.

Appendix A has the proof of this lemma, which again hides the losing bids (besides the second-highest). Furthermore, any misbehavior is eventually detected during an audit.

5 Adding public verifiability

For an auditor to verify the outcome of an auction, the auditor needs to check that (1) the highest bid b^* selected in Step 3 of the auction is correct; (2) that the bit β_{b^*} of the winning bidder is 1 in Step 4; and (3) that the value computed in Step 5 was set as the auction's sale price. We start by making the output of AFEs publicly verifiable, and then discuss how an auditor can perform the above checks.

5.1 Verifiable and private AFEs

We make AFEs verifiable with a procedure that takes the result of the AFE computation—the sum vector $\mathbf{v}$—and commitments to the inputs, and outputs whether $\mathbf{v}$ is correct.

The key idea of our verification procedure is to observe that by their very nature, AFEs encode inputs in such a way that the desired functions (OR, MAX, etc.) can be computed with only additions. Hence, if one uses an *additively homomorphic commitment* scheme on the input AFE values, it is possible to check the result of the AFE computation without learning the inputs by adding the commitments and confirming whether the result is also a valid commitment of the output. We explain this process for the "OR" AFE of Section 4.1.

Encode V-OR. Given a bit $x \in \{0,1\}$, its verifiable AFE is a tuple v consisting of 2 elements in $\mathbb{Z}_p$ defined as follows. The first element in v is given by Encode-OR (§4.1). The second element in v is a non-zero uniform random element in $\mathbb{Z}_p$.

Commit V-AFE. Given a V-AFE tuple $v \in \mathbb{Z}_p^2$ encoding bit x with Encode V-OR, we use the Pedersen commitment [74] defined over a multiplicative group $\mathbb{G}$ of prime order p with generators $\{g,h\}$.[1] The commitment is $c = g^{v[0]} \cdot h^{v[1]}$.

This commitment *perfectly hides* the V-AFE tuple (an adversary cannot learn the tuple from the commitment); it *binds* the tuple (a committer cannot claim to have committed to a different tuple) if the discrete log problem is hard in $\mathbb{G}$. It is also additively homomorphic: given a commitment $c_1 \in \mathbb{G}$ to a tuple $v_1 \in \mathbb{Z}_p^2$ and a commitment $c_2 \in \mathbb{G}$ to a tuple $v_2 \in \mathbb{Z}_p^2$, $c_3 = c_1 \cdot c_2$ is a valid commitment to the tuple $v_1 + v_2$.

Compute and Decode. Given a set of n V-AFE tuples $\{v_1, \ldots, v_n\}$, which encode n bits $\{x_1, \ldots, x_n\}$ with the above Encode V-OR procedure, compute the OR of the n bits by adding the V-AFE tuples component-wise: $\mathbf{v} = v_1 + \cdots + v_n$. Decode V-OR calls Decode-OR on the first element in $\mathbf{v}$.

Verify V-OR. Given the V-AFE sum tuple $\mathbf{v}$ which encodes the result of the Compute V-OR procedure over n V-AFE tuples $\{v_1, \ldots, v_n\}$, and given a set of commitments $\{c_1, \ldots, c_n\}$ to these tuples generated with the Commit V-AFE procedure, one can verify $\mathbf{v}$ by checking if $g^{\mathbf{v}[0]} \cdot h^{\mathbf{v}[1]} \stackrel{?}{=} \prod_{j=1}^{n} c_j$. Verify V-OR outputs "ok" if the check passes, and "fail" otherwise.

The above approach generalizes to other functions (e.g., MAX) that require more complex encodings (e.g., vectors) since those encodings are just sets of AFE values. For example, a V-AFE vector is simply a vector of V-AFE tuples, and the commitment is a vector of Pedersen commitments—one for each tuple in the V-AFE vector. The approach can also be combined with secret sharing (§4.3) to hide the inputs from non-colluding parties. Specifically, the input providers (e.g., bidders in our case) generate their V-AFE vectors $\{M_1, \ldots, M_n\}$ and compute the corresponding commitments $\{c_1, \ldots, c_n\}$, which are made available on a public log. Then, the input providers generate secret shares for their V-AFE vectors and give these shares to the computing parties as described in Section 4.3. Finally, the computing parties combine their sum vector shares into the V-AFE vector $\mathbf{M}$ and verify each entry with Verify V-OR and the commitments.

5.2 Verifiable, private, and decentralized auction

We now discuss how to extend the protocol of Section 4.4 with the V-AFE construction of Section 5.1 to obtain verifiability of the auction's outcome in addition to privacy.

Recall that in Step 2 of the auction protocol (§4.4), a bidder i encodes its bid using Encode-MAX (§4.2) which produces an AFE vector M_i, where each entry in M_i is an Encode-OR (§4.1) of each bit of bidder i's unary-formatted bid. In our verifiable auction, the bidder instead uses the Encode V-OR procedure (§5.1), so M_i is made up of ℓ V-AFE tuples. Bidder i also creates, for each entry of M_i, a commitment using Commit

[1] As an (insecure) example, the set $\{1,3,4,5,9\}$ in $\mathbb{Z}_{11}$ forms a multiplicative group with 5 elements (its order is $p = 5$). A generator for this group is 3 since repeated multiplications of 3 with itself generates every element.

V-AFE (§5.1). Let C_i denote the corresponding vector of ℓ commitments for M_i. Bidder i then splits M_i (§4.3), and sends to the auctioneers a collision-resistant hash of C_i and the AFE vector shares (M_i^1 or M_i^2, depending on the party).

Asynchronously, bidder i uploads to the public log (§7) its bidder id, C_i, and a signature of C_i that validates with the bidder id (recall that bidder ids are verification keys). The other steps of the auction proceed as before. At the end of the auction, the publisher and the auxiliary server upload an *audit trail* to the public log containing: (1) the auction's outcome, consisting of the bidder id of the winner w, the highest bid b^*, and the auction's sale price; (2) their share of the sum vector computed in Step 3 and 5 of the auction protocol; (3) the b^*-th entry of the V-AFE vector share of each candidate winner chosen in Step 4 and the seed for the PRG used; and (4) the hashes (to commitments) they received from bidders.

Deferred public verification. After the auction completes, an *auditor* can choose to verify that the auction was done correctly as follows. The auditor accesses the auction's audit trail from the public log, and verifies that the uploaded hashes match the commitments, and all signatures on the commitments are valid. To verify the highest bid in Step 3, the auditor aggregates the sum vector shares in the audit trail to obtain $\mathbf{M}$. Then, the auditor computes the highest bid b^* by calling Decode-MAX on $\mathbf{M}$ (§4.2). Finally, the auditor runs, for all $j \in [1, \ell]$, Verify V-OR (§5.1) using as input the j-th entry of $\mathbf{M}$ (acting as the V-AFE sum value), and the j-th entry of every commitment vector submitted by the n bidders (i.e., for all $i \in [1, n]$, $C_i[j]$), as the commitment set. If all checks pass, then Step 3 was correct. The auditor performs the same actions for Step 5 to verify the second highest bid.

To verify Step 4, the auditor checks, for each of the candidate winners x, whether $g^{M_x[b^*][0]} \cdot h^{M_x[b^*][1]} \stackrel{?}{=} C_x[b^*]$. The auditor also checks that the Decode-OR of $M_w[b^*]$ is 1 (i.e., the actual winner's bit at position b^* is indeed a 1), and the Decode-OR of $M_x[b^*]$ for all other candidate winners x is 0. Then, the auditor uses the PRG and the seed in the audit trail to check that the bidder ids of the set of candidate winners are correct and that w was the last bidder id sampled.

Theorem 1. Addax's auction protocol with deferred verification achieves completeness, soundness, and privacy.

We give the full definitions and proofs in Appendix B. Note that detection is different from finding the party at fault.

5.3 Assigning blame

An auction may be aborted during the online phase, or deferred verification may fail. In these cases, Addax can narrow down the set of faulty parties. As parties participate in many auctions (recall that exchanges process billions of auctions per day), one could develop detection algorithms that flag those who are present in an unusually high number of aborted or failed auctions. We discuss this in more detail in Appendix D.

6 Optimizations

This section discusses two optimizations. The first adds interaction between the bidders and the auctioneers to dramatically cut costs. The second reduces interaction between the auctioneers, which lowers latency, but leaks the existence of ties.

6.1 Less communication with an interactive MAX

A major drawback of the proposed private auction protocol is that the computation and communication complexity of computing MAX using AFE vectors and their corresponding shares is $O(\ell)$, where ℓ is the highest possible bid (§4.2). Meanwhile, bids range from cents to tens of dollars; a realistic deployment would need $\ell \geq$ 1,000, which is too costly. In this section we show how to modify the auction protocol to add r rounds of interaction between bidders and the *auctioneers* (publisher and auxiliary server) in exchange for reducing computation and communication complexity to $O(r \cdot \ell^{1/r})$.

High-level idea. In Figure 2, bidders first represent their bids as a unary bit vector, and then use Encode-OR on each bit to create vector M. This vector is then split into shares M^1 and M^2. The auctioneers aggregate their shares locally and then exchange their sum vector shares to construct the sum vector $\mathbf{M}$. This vector is then decoded into a unary bit vector that contains the result of max. Observe that if the bidders were to use Encode-OR only on the last two bits of their bit vectors (the gray and light gray cells), they would obtain the last 2 entries of M, which would then be split into the last two entries of M^1 and M^2, and would become the last 2 entries of the sum vector shares, and finally of $\mathbf{M}$. Decoding these two entries of $\mathbf{M}$ results in the last two bits of the final unary bit vector (in the example these bits are 1 and 0). The fact that the last bit is 0 means that the max value must be $< \ell$. The fact that the penultimate bit is 1 means that the max value must be $\geq \ell - 1$. Hence, encoding and sharing only a subset of bidders' unary bit vectors is enough to compute the max value. Of course, in this example we knew ahead of time which two elements to pick to get a tight upper and lower bound on the max. In our protocol, the auctioneers do r rounds of k-ary search ($k = \ell^{1/r}$) to find the consecutive positions at which the final unary bit vector changes from a 1 to a 0, which yields the max.

Protocol. Using the notation of Section 4.3, each bidder i sends $\lceil \ell^{1/r} \rceil$ entries of the AFE vector shares M^1 and M^2 to the auctioneers in each round. The entries sent in each round are evenly distributed between the current lower and upper bounds on the maximum bid (initially set to 1 and ℓ, respectively). For each of the chosen entries j, the auxiliary server runs the Compute-OR procedure (§4.1) by aggregating the shares it receives from each bidder i: $M^1[j] = \sum_i M_i^1[j]$. Likewise, the publisher computes $M^2[j] = \sum_i M_i^2[j]$. The publisher and the auxiliary server then exchange their sum shares for each entry j, allowing the reconstruction of $\mathbf{M}[j] = M^1[j] + M^2[j]$. Calling Decode-OR (§4.1) on $\mathbf{M}[j]$ returns whether bit β_j in the unary vector is 1 or 0. If β_j is 1, the highest bid $b^* \geq j$. Else, $b^* < j$.

This establishes a new lower and upper bound on b^* with respect to the exchanged entries. After r rounds, the number of entries sent by each bidder to each auctioneer is $\leq r \cdot \lceil \ell^{1/r} \rceil$.

In this protocol, bidders transmit a subset of the entries that they send to the auctioneers in the non-interactive variant, and hence they reveal less information. But there is one downside: bidders or an auctioneer can adaptively send inconsistent shares in response to partial information (e.g., knowledge that the max is in a given range). This could affect the auction's integrity. Addressing this issue requires extending the protocol with two extra safeguards: (1) an asynchronous step to find the sale price bidder which is similar to Step 4 in Section 4.4; and (2) generating a zero-knowledge proof that the sale price bidder's AFE vectors are valid without leaking the original AFE vector. Appendix C describes these steps in detail and proves the following two lemmas.

Lemma 5. If either the auxiliary server or the publisher misbehaves, or malicious bidders issue inconsistent AFE shares, the above interactive protocol leaks no more information about losing bidders' bids than the non-interactive variant.

Lemma 6. If either the auxiliary server or the publisher misbehaves, or malicious bidders issue inconsistent AFE shares, the above protocol (with the extra safeguards) ensures that malicious actions are detected during an audit.

6.2 Lower latency by leaking the existence of ties

Of all the steps in the auction protocol, finding the winner (§4.4, Step 4) is the most expensive since each interaction between the publisher and the auxiliary server occurs over WAN. This step consists of two parts: (1) pick a random candidate winner w, and (2) exchange the b^*-th entry of w's AFE vector shares to determine whether w indeed had the highest bid—trying again otherwise. The iterated nature of this algorithm aims to find one of the highest bidders at random (as soon as a highest bidder w is found, the auctioneers halt). One can eliminate this cost if one is willing to leak the number of ties. The protocol is simple: the publisher and auxiliary server exchange the b^*-th entry of the vector shares of *all* bidders. In the absence of ties, after decoding, only one bidder will have a 1 and all others will have a 0. If there are ties, multiple bidders will have a 1 at position b^*, and the auctioneers use the PRG to break the tie. Addax adopts this tradeoff.

We note that the added leakage is actually minor given that in the interactive protocol (Step 4), one learns that it took k tries to find the winner. In an auction with no ties, k would be $n/2$ in expectation, so the value of k already leaks some information about the number of potential ties that may exist.

7 Search and filtering

In Addax, bidders register to participate in auctions by storing their information (e.g., ad categories, domain of their bidding service) on a public tamper-proof log, and auction participants also use this log to create an audit trail (§5.2). Our implementation uses the Algorand blockchain [3] to maintain the log, though we could have used a BFT consortium or a trusted party (if one exists). Addax also needs a way to *search* the blockchain. This is typically done by downloading the entire blockchain and locally searching for the desired objects. Of course, this is onerous for browsers, as no user would ever maintain a copy of the blockchain just to receive ads. Instead, our implementation uses the Purestake indexer [14]. The downside is that one must trust this indexer. One way to remove this assumption is to use a verifiable search engine for blockchains [69].

Even with the Purestake indexer, querying data is slow: it takes seconds to get a response. Therefore, Addax keeps a copy of the log in *untrusted* cache servers; Addax then queries Purestake asynchronously to verify the cache servers' results. Querying cache servers takes only a few milliseconds.

In the rest of this section we describe how browsers do local filtering and fetch advertisers' data. We discuss how browsers interact with the Algorand blockchain in Appendix E.

7.1 Filtering and inviting advertisers

Upon visiting a page with ads and obtaining a list of allowed categories from the publisher, the browser queries the cache server to get bidders who match these categories. The browser caches bidder information and only sends "if-modified-since" requests to the cache server to reduce communication. Borrowing ideas from Privad [58] and Adnostic [80], the browser assigns a preference score for each of the returned bidders. The browser then picks the top k bidders and invites them to join the auction, supplying them with information about the publisher and the user. Depending on the configuration of Addax, the user information can be empty (for generic ads), include a group or topic id (as in FLoC [89] and Topics [16]), or include cookies and demographic information. Since the publisher's revenue depends on bids, and bidder valuations are based on user information, different publishers can require different levels of information disclosure to access their content. This is similar to how publishers detect ad blocking software and request that users disable it.

8 Implementation

Addax consists of 2.2K lines of C++ and 400 lines of Python and PyTeal [15] for Algorand smart contracts. Addax's client-side tracking is done outside the browser and interacts with Chrome via native messaging [9]. We use OpenSSL 3.0.0 [12] for basic cryptographic operations (e.g., `BN_rand` as the PRG). Addax's Pedersen commitment (§5.1) is defined over elliptic curve `secp192r1`, as is the Schnorr signature scheme [78] that bidders use to sign their log entries. Elements in V-AFE vectors are defined over the 192-bit field used in `secp192r1`.

Baselines. To contextualize our contributions, we implement baselines using state-of-the-art homomorphic encryption (HE) and secure two-party computation (2PC) frameworks:

- *CKKS* on *SEAL* [25, 47]: HE for arithmetic operations.
- *TFHE* [49, 50]: HE for boolean operations.
- *MASCOT* on *MP-SPDZ* [65, 66]: Arithmetic 2PC.
- *ag2pc* on *EMP toolkit* [85, 86]: Boolean 2PC.

Homomorphic encryption. The publisher generates cryptographic keys and sends the public key to bidders. Bidders send bids encrypted with the public key to the auxiliary server, who runs the auction over ciphertexts and supplies the result to the publisher for decryption with the secret key. For SEAL we implement and measure the `maxId` algorithm by Cheon et al. [48] which is the best known way to find the ciphertext with the max value. While this is a subset of running an auction, this one step is already more expensive than Addax's full auction protocol. For TFHE we implement the whole auction. Neither baseline provides integrity.

Multiparty computation. Advertisers commit to their bids and send them to the publisher and auxiliary server alongside additive shares of their bids and the commitment randomness. Inside the MPC, the auxiliary server and publisher reconstruct the bids and the commitment randomness from their shares, check that the commitments match, and the bids are the committed values, and then run the auction using the bids. For commitments we use $H(rand||bid)$ and assume H is a random oracle. We use hash functions already implemented and optimized for these frameworks (e.g., SHA3, SHA256, MiMC).

9 Evaluation

This section studies the following questions:

1. What are the costs of Addax's auction for each party?
2. How does Addax's auction compare with alternatives?
3. What is the resource overhead of deploying Addax over a non-private and unverifiable exchange?
4. How expensive is the verification procedure?

Appendix E.3 discusses the cost of interacting with the log.

Evaluation environment. We run our experiments across AWS data centers to account for Addax's decentralized nature. The publisher is in US East (Ohio) on a c5.2xlarge instance, the auxiliary server in US West (OR) on a c5.2xlarge instance, and bidders in US West (CA) on c5.12xlarge instances. We use standard Ubuntu 20.04 for all of them. PureStake exposes a REST API and runs on servers in Ontario, CA, and OR.

Method and metrics. Our key metrics are the end-to-end latency, total network communication, and throughput of the auction procedure. This includes the events after the browser fetches the page from the publisher and initiates the auction, but before the browser fetches and displays the ad on the user's screen. In short, we measure the overhead of Addax over the status quo of using a centralized non-private ad exchange. We report the mean over 20 trials and one standard deviation. *We focus on second-price auctions in this evaluation*, as they are the more complex type of auction. If Addax is used for first-price auctions, the costs are 30% lower: auctions with 96 bidders complete within 440 ms, and Addax can sustain a throughput of 360 auctions per second per core.

	Size (MB)		Generation (ms)
AFE vector shares	0.48		87.55
Materials (non-interactive)	0.25		537.9
Materials (interactive)	1.705		1,802.0
	Non-interactive	**2-round**	**4-round**
Communication (MB)	0.48	0.0144	0.0034

FIGURE 3—Size of AFE vector shares and other materials (e.g., commitments), their generation time, and the total communication between a bidder and one auctioneer under different Addax variants.

Parameters. Prior reports [92] suggest that the typical number of bidders (usually demand-side platforms) in an auction is under 30. We experiment with up to 96 bidders, but Addax could handle more with little extra latency since most of the latency comes from round trips between the two servers and is not impacted by the number of bidders. We set $\ell = 10{,}000$, which supports bid ranges consistent with those observed in practice [93]. This results in a probability of computing the wrong MAX of $\approx 1 - (1 - \frac{1}{2^{192}})^{10{,}000}$, which is negligible.

Our baseline implementations are generous: we use 13-bits for bids (4/5 of our bid range) and do not measure the time to receive shares or ciphertexts from bidders for any of them.

9.1 Microbenchmarks: Addax's auction protocol

To answer our first research question we microbenchmark the operations of each of the auction participants.

Bidder's cost. Before the auction starts, bidders encode their bids, commit to the encodings, and send their shares to the auctioneers. Figure 3 depicts the time required to generate an AFE vector, and the verification materials in both the non-interactive protocol (§5) and the interactive variant (§6.1) using 8 CPU threads. For the latter we include the cost of the safeguards detailed in Appendix C.2. As shown in the figure, generating these materials is more expensive than the time budgeted for an auction. However, AFE vectors are made up of random elements; the only dependence on bids is whether to use a uniform element or a zero (§4.1). As a result, all materials can be precomputed and kept aside. Furthermore, their generation is parallelizable: we get a 5.83× speedup with 6× more cores. We expect bidders to be able to maintain their desired throughput, albeit at a higher cost ($) than they incur today.

When the auction starts and the bidder decides on its bid, it can draw from the set of pre-generated materials to construct bid-specific AFE vector shares, commitments, and proofs. With pre-generated materials, bidders respond in 10 ms.

Local auction computation. To determine the costs to the auxiliary server and the publisher we run a microbenchmark where both auctioneers run on the same machine, are given all materials (e.g., AFE shares), and compute the auction without the effects of network latency. Figure 4a shows the time for

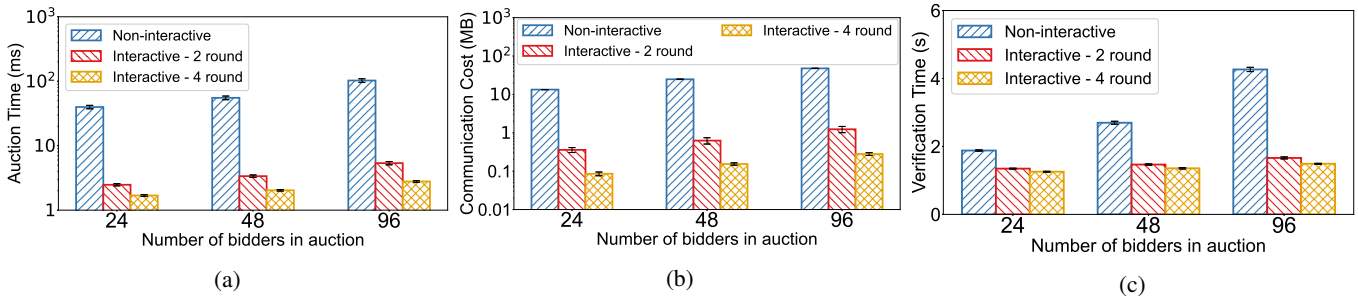


FIGURE 4—Cost of running and verifying an auction across Addax's variants. Figure (a) depicts the processing time incurred by each auctioneer; (b) depicts the communication costs for each auctioneer; and (c) depicts the costs to an auditor.

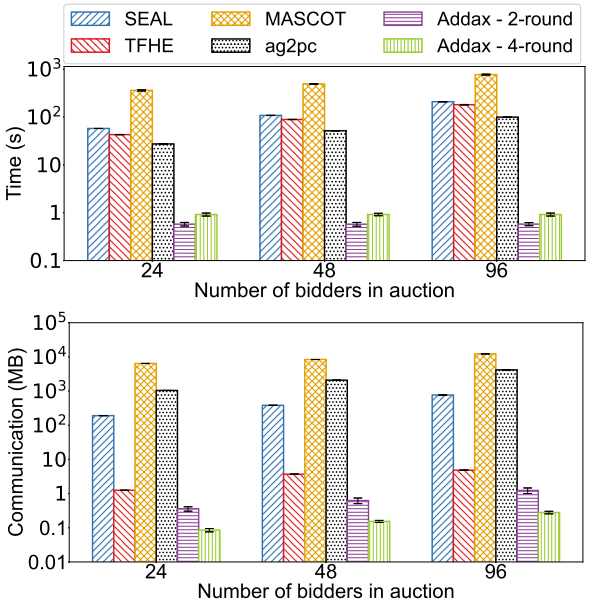


FIGURE 5—End-to-end latency and communication costs for an auction in Addax and several baselines over WAN.

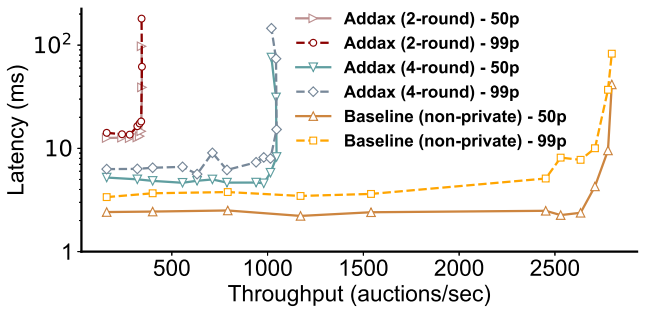


FIGURE 6—Median and 99-percentile response time and server throughput for Addax and a non-private baseline for an auction with 96 bidders. Each data point represents the latency and the throughput achieved at a given load (low and to the right is better).

different Addax variants. Compared to the non-interactive protocol, Addax with 4 rounds (§6.1) requires fewer operations since it acts on a tiny subset of the entries of the AFE vector, and reduces computation time for a 96 bidder auction from 102 ms to 2.8 ms. Interactivity also reduces communication costs for effectively the same reason (acts on fewer entries). As we show in Figure 4b, the size of the AFE shares exchanged between bidders and each auctioneer in the non-interactive variant of Addax with 96 bidders is 47.68 MB, whereas it is 0.28 MB with 4-rounds and 1.23 MB with 2-rounds.

9.2 End-to-end performance

The above microbenchmarks give an idea of the computation and communication costs that are expected when running auctions with Addax. However, the metric that actually matters is end-to-end latency over WAN. Figure 5 shows the computation and communication costs of Addax's end-to-end protocol over a WAN deployment, from the time that the publisher starts the auction, to the time the winner is notified. This figure also shows the baselines described in Section 8.

In terms of auction latency, Addax's 2-round variant is by far the most efficient option, often by orders of magnitude compared to the baselines. Addax's 2-round variant beats the 4-round variant due to fewer WAN RTTs at a slight increase in the amount of communication. At 96 bidders, the browser receives an ad tag from Addax in 579 ms; behind the scenes, the auctioneers exchange 1.23 MB of data to compute the auction. Of this time, the servers only spend 5 ms computing; the rest is network latency. Thus, having more bidders will not meaningfully increase the end-to-end latency of Addax.

For comparison, studies [1, 7, 20, 87] show that page loading times today take several *seconds*, so we expect Addax to run auctions asynchronously as the page loads without significantly impacting the user experience.

9.3 Costs over a non-private unverifiable baseline

To understand the additional computational resources required to deploy Addax, we compare its throughput on a c5.2xlarge instance (4-core VM) to a baseline that simply finds the highest and second highest bids (the only non-trivial computation is establishing a TLS session between the browser and the publisher). We run an open-loop workload with varying load and with all inputs already in-memory, so we do not measure network latency. Figure 6 gives the results.

Addax's 2-round and 4-round variant achieve 8.1× and

	Non-interactive	2-round	4-round
Auctioneers	1.932	0.056	0.025
Bidders	0.250	0.250	0.250
Winner	0.250	0.730	0.730
Sale price bidder	0.250	1.705	1.705

FIGURE 7—Total size (MB) of the audit information that parties must upload to the public log in an auction of 96 bidders.

2.7× lower throughput than the baseline. As Addax requires two servers, this translates to 16.2× and 5.4× more computation resources to maintain the same throughput as the baseline. This suggests that Addax could process the high volume of auctions that exchanges process today while providing integrity and privacy guarantees—albeit at a premium cost.

9.4 Cost of verification

In this section we evaluate the cost for auction participants to supply the necessary materials to leave an audit trail, and for an auditor to validate the correctness of an auction.

Leaving an audit trail. After the auction finishes, participants upload their audit materials (§5.2–§5.3) to Algorand. This takes around 0.8 sec. Figure 7 gives the size of the materials that each party uploads for a 96-bidder auction. In the interactive variants, the winner uploads its full AFE vector, and the sale price bidder uploads its full AFE vector with a random mask and proofs as described in Appendix C.5.

Verification time. Verification requires downloading the materials from Algorand, checking the hash of commitments, and checking the recovered AFE sum vectors and bit encodings (§5.2). Auditors also need to validate that AFE vectors from the winner and sale price bidder are valid (§C.2). Figure 4c depicts the time of verification. In the non-interactive variant, deserializing commitments and verifying the two sum vectors takes most of the time. In contrast, in the interactive variants the expensive step is validating the winner and sale price bidders' AFE vectors. To verify a 96-bidder auction, the non-interactive variant requires 4.27 sec, while the 2-round and 4-round variants take 1.66 sec and 1.49 sec, respectively.

10 Related work

This section describes other efforts that relate to Addax.

Advertising. There is a rich literature in privacy preserving ads [35, 44, 52, 57–59, 75, 76, 80], but none focuses on private and verifiable auctions. VEX [35] provides verifiability but the auctioneer learns all bids. Privad [58, 59], Adnostic [80], FloC [89], Topics [16], and others [37, 57, 76] reduce the collection of user information, but auctions are still conducted by a party that learns all bids and cannot be audited.

Private and verifiable auctions. In other domains, there is work on private or verifiable auctions. Parkes et al. [73] provide auction integrity but the auctioneer learns all bids, unlike Addax. Other works [62, 67] provide privacy but not integrity. Finally, there are several multiparty protocols [38, 42] where the bidders jointly compute the auction. This is worse than our MPC baseline in Section 8 in that here bidders actively participate in the protocol rather than merely generating shares. This does not scale to more than a handful of bidders.

11 Discussion

Addax departs from the status quo by introducing accountability to an opaque ecosystem. While this is a disruptive change, there are two things on Addax's favor. First, the ad-tech industry already uses browsers to kickstart auctions and invite bidders [4] and newer proposals like Google's FLEDGE [33] push even more functionality to browsers include client-side tracking. Second, Addax is incrementally deployable: an Addax-enabled browser can send an HTTP X-header indicating its support of the protocol, and interested publishers can respond with Addax-based ad spots while continuing to offer traditional ads to other users. Furthermore, we think many missing features can be implemented in Addax.

Content curation. A key role of exchanges is to prevent *malvertising* (the use of ads to spread malware) or ads that can damage the publisher's brand. On the one hand, content curation is hard even in centralized environments: reports of malicious actors leveraging ad networks to distribute malware are common [8]. On the other hand, since advertisers publish their information on Addax's public log, one could imagine requiring advertisers to upload their ads as well. Then, just like existing services scan blockchains for anomalous transactions, they can scan Addax's log to detect and flag malicious ads.

Fraud prevention. Many existing mechanisms to prevent *publisher fraud* (e.g., using clickbots to increase revenue) [79] still work in our setting. For example, bidders can still observe anomalous changes in ad traffic from a publisher, and can perform randomized auditing with *bluff ads* [60] (uninviting ads unlikely to be clicked by real users). Other techniques that collect hard-to-fake signals from a device with the aim of detecting bots [18, 21] could also be used, but more work is needed to port them to our context.

Conversions. Analytics are also critical to the ad ecosystem. Currently, advertisers and publishers rely on third-party cookies to track when a user performs an action after viewing an ad (a "conversion"). A recent proposal [94] shows how this can be done without cookies and without learning the user's identity; this approach is compatible with Addax's architecture.

Trust-performance tradeoff. Our description of Addax uses 2 parties but the protocols naturally generalize to k auxiliary servers; if either the publisher or any of the k auxiliary servers is honest, Addax provides its guarantees. Of course, as the number of parties increases the costs also increase. This tradeoff can be taken into account at deployment time.

Acknowledgments

We thank the OSDI and NSDI reviewers, and our shepherd, Bryan Ford, for their thoughtful comments that improved this paper. This work was funded in part by NSF grant CNS-2045861 and DARPA contract HR0011-17-C0047.

References

[1] About pagespeed insights. https://developers.google.com/speed/docs/insights/v5/about.

[2] Adtelligent's header bidding platform. https://adtelligent.com/header-bidding-platform/.

[3] Algorand. https://www.algorand.com/.

[4] The beginner's guide to header bidding. https://adprofs.co/beginners-guide-to-header-bidding/.

[5] Google Has a New Plan to Kill Cookies. People Are Still Mad. https://www.wired.co.uk/article/google-floc-cookies-chrome-topics.

[6] Google's Topics API: Rebranding FLoC Without Addressing Key Privacy Issues. https://brave.com/web-standards-at-brave/7-googles-topics-api/.

[7] Here's what we learned about page speed. https://backlinko.com/page-speed-stats.

[8] Malvertising: What is it and how to avoid it. https://us.norton.com/internetsecurity-malware-malvertising.html.

[9] Native messaging. https://developer.chrome.com/docs/apps/nativeMessaging/.

[10] Openrtb protocol buffer 2.5.0. https://developers.google.com/authorized-buyers/rtb/downloads/openrtb-proto.

[11] OpenRTB (real time bidding). https://www.iab.com/guidelines/real-time-bidding-rtb-project/.

[12] OpenSSL. https://www.openssl.org.

[13] Parakeet. https://github.com/WICG/privacy-preserving-ads/blob/main/Parakeet.md.

[14] Purestake. https://www.purestake.com/.

[15] Pyteal: Algorand smart contracts in python. https://github.com/algorand/pyteal.

[16] The Topics API. https://developer.chrome.com/docs/privacy-sandbox/topics/.

[17] This is how Google plans to track you now. https://www.slashgear.com/this-is-how-google-plans-to-track-you-now-25708910/.

[18] What is recaptcha? https://www.google.com/recaptcha/about/.

[19] Cookie synching. https://www.admonsters.com/cookie-synching/, 2010.

[20] Find out how you stack up to new industry benchmarks for mobile page speed. https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf, 2017.

[21] Fighting fraud using partially blind signatures. https://engineering.fb.com/2019/10/16/security/partially-blind-signatures/, 2019.

[22] Iab internet advertising revenue report. https://www.iab.com/wp-content/uploads/2019/05/Full-Year-2018-IAB-Internet-Advertising-Revenue-Report.pdf, 2019.

[23] Cookie matching. https://developers.google.com/authorized-buyers/rtb/cookie-guide, 2020.

[24] Lattigo v2.1.1. Online: http://github.com/ldsec/lattigo, Dec. 2020.

[25] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, Nov. 2020.

[26] PALISADE Lattice Cryptography Library (release 1.10.6). https://palisade-crypto.org/, Dec. 2020.

[27] Private marketplace ad spending to surpass open exchange in 2020. https://www.emarketer.com/content/private-marketplace-ad-spending-to-surpass-open-exchange-in-2020, 2020.

[28] Antitrust battle latest: Google, facebook 'colluded' to smash apple's privacy protections. https://www.theregister.com/2021/10/22/google_facebook_antitrust_complaint/, 2021.

[29] Azure attestation client library for .net - version 1.0.0. https://docs.microsoft.com/en-us/dotnet/api/overview/azure/security.attestation-readme, 2021.

[30] Bring more bids to the auction with open bidding. https://admanager.google.com/home/resources/feature-brief-open-bidding/, 2021.

[31] SCALE and MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA, 2021.

[32] The comprehensive guide to online advertising costs. https://www.wordstream.com/blog/ws/2017/07/05/online-advertising-costs, 2022.

[33] Fledge api. https://developer.chrome.com/docs/privacy-sandbox/fledge/, 2022.

[34] E. Anderson, M. Chase, F. B. Durak, E. Ghosh, K. Laine, and C. Weng. Aggregate measurement via oblivious shuffling, 2021. https://ia.cr/2021/1490.

[35] S. Angel and M. Walfish. Verifiable auctions for online ad exchanges. In *Proceedings of the ACM SIGCOMM Conference*, 2013.

[36] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2), 2010.

[37] M. Backes, A. Kate, M. Maffei, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[38] S. Bag, F. Hao, S. F. Shahandashti, and I. G. Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15, 2020.

[39] M. A. Bashir and C. Wilson. Diffusion of User Tracking Data in the Online Advertising Ecosystem. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2018.

[40] A. Bois, I. Cascudo, D. Fiore, and D. Kim. Flexible and efficient verifiable computation on encrypted data. Cryptology

ePrint Archive, Report 2020/1526, 2020.

[41] F. Bourse, O. Sanders, and J. Traoré. Improved secure integer comparison via homomorphic encryption. In *Topics in Cryptology – CT-RSA 2020*, 2020.

[42] F. Brandt. A verifiable, bidder-resolved auction protocol. In *Proceedings of the 5th International Workshop on Deception, Fraud and Trust in Agent Societies*, 2002.

[43] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the USENIX Security Symposium*, 2018.

[44] J. Cartlidge, N. P. Smart, and Y. Talibi Alaoui. Mpc joins the dark side. In *International Symposium on Information, Computer, and Communications Security*, 2019.

[45] F. Chanchary and S. Chiasson. User perceptions of sharing, advertising, and tracking. In *Symposium On Usable Privacy and Security (SOUPS)*, 2015.

[46] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[47] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2017.

[48] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee. Numerical method for comparison on homomorphically encrypted numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2019.

[49] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[50] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33, 2020.

[51] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[52] E. Deng, H. Zhang, P. Wu, F. Guo, Z. Liu, H. Zhu, and Z. Cao. Pri-rtb: Privacy-preserving real-time bidding for securing mobile advertisement in ubiquitous computing. In *Information Sciences*, 2019.

[53] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — CRYPTO' 86*, 1987.

[54] D. Fiore, A. Nitulescu, and D. Pointcheval. Boosting verifiable computation on encrypted data. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2020.

[55] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[56] A. Goldfarb and C. E. Tucker. Privacy regulation and online advertising. *Management Science*, 2010.

[57] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[58] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[59] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving ads from localhost for performance, privacy, and profit. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2009.

[60] H. Haddadi. Fighting online click-fraud using bluff ads. *ACM SIGCOMM Computer Communication Review*, 40(2), 2010.

[61] S. Halevi and V. Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020.

[62] M. Harkavy, J. D. Tygar, and H. Kikuchi. Electronic auctions with private bids. In *3rd USENIX Workshop on Electronic Commerce (EC 98)*, 1998.

[63] J. Horwitz and K. Hagey. Google's secret 'project bernanke' revealed in texas antitrust case. `https://www.wsj.com/articles/googles-secret-project-bernanke-revealed-in-texas-antitrust-case-11618097760`, Apr. 2021.

[64] I. Iliashenko and V. Zucca. Faster homomorphic comparison operations for BGV and BFV. Cryptology ePrint Archive, Report 2021/315, 2021.

[65] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020.

[66] M. Keller, E. Orsini, and P. Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[67] H. Kikuchi, S. Hotta, K. Abe, and S. Nakanishi. Distributed auction servers resolving winner and winning bid without revealing privacy of bids. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, 2000.

[68] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the USENIX Security Symposium*, 2017.

[69] M. Li, T. Zhang, J. Zhu, C. Tan, Y. Xia, S. Angel, and H. Chen. Bringing decentralized search to decentralized services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[70] Y. Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. `https://ia.cr/2016/046`.

[71] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[72] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[73] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy

auctions. *Electronic Commerce Research and Applications*, 2008.

[74] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1991.

[75] G. Pestana, I. Querejeta-Azurmendi, P. Papadopoulos, and B. Livshits. Themis: Decentralized and trustless ad platform with reporting integrity. `https://arxiv.org/abs/2007.05556v2`, 2020.

[76] A. Reznichenko, S. Guha, and P. Francis. Auctions in do-not-track compliant internet advertising. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[77] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[78] C. P. Schnorr. Efficient identification and signatures for smart cards. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1989.

[79] B. Stone-Gross, R. Stevens, R. Kemmerer, C. Kruegel, G. Vigna, and A. Zarras. Understanding fraudulent activities in online ad exchanges. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2011.

[80] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[81] R. Tracy and J. Horwitz. Inside the Google-Facebook ad deal at the heart of a price-fixing lawsuit. `https://www.wsj.com/articles/inside-the-google-facebook-ad-deal-at-the-heart-of-a-price-fixing-lawsuit-11609254758`, Dec. 2020.

[82] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[83] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 1961.

[84] D. Wakabayashi and T. Hsu. Behind a secret deal between Google and Facebook. `https://www.nytimes.com/2021/01/17/technology/google-facebook-ad-deal-antitrust.html`, Jan. 2021.

[85] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[86] X. Wang, S. Ranellucci, and J. Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[87] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[88] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Scalable anonymous group communication in the anytrust model. In *Proceedings of the European Workshop on System Security (EUROSEC)*, Apr. 2012.

[89] Y. Xiao and J. Karlin. Federated learning of cohorts. `https://wicg.github.io/floc/`, 2021.

[90] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[91] J. Yan, N. Liu, G. Wang, W. Zhang, Y. Jiang, and Z. Chen. How much can behavioral targeting help online advertising? In *International World Wide Web Conference (WWW)*, 2009.

[92] S. Yuan, J. Wang, B. Chen, P. Mason, and S. Seljan. An empirical study of reserve price optimisation in real-time bidding. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014.

[93] W. Zhang, S. Yuan, J. Wang, and X. Shen. Real-time bidding benchmarking with iPinYou dataset. `https://arxiv.org/abs/1407.7073`, 2015.

[94] K. Zhong, Y. Ma, and S. Angel. Ibex: Privacy-preserving ad conversion tracking and bidding. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

A Proofs for lemmas

Proof of Lemma 1. Let h be the maximum bid among all honest bidders. The event claimed in Lemma 1 is equivalent to one of two cases: (1) Decode-MAX outputs a valid unary bit vector whose value is larger than or equal to h; or (2) Decode-MAX outputs an invalid unary bit vector. Consider the opposite event where Decode-MAX outputs a valid unary bit vector whose value is smaller than h. We denote the probability of this event as $\Pr(\text{opposite})$. $\Pr(\text{opposite}) \leq \Pr(\text{Decode-OR outputs 0 at position } h) \leq 1/p$. Therefore, the probability that the output of Step 3 is greater than or equal to h or is an invalid unary bit vector is $1-\Pr(\text{opposite}) \geq 1-1/p$. In our construction p is a large prime, and hence $1 - 1/p \approx 1$.

Proof of Lemma 2. Let two parties hold additive shares for a given AFE value that was produced by Encode-OR. Let one of the parties be honest and the other malicious. Without seeing the share held by the honest party, the probability of the malicious party generating a share that results in Decode-OR outputting 0 is $1/p$: the malicious party would have to correctly guess the exact value needed to make the two shares add up to 0, and shares are uniformly random values in $\mathbb{Z}_p$. Thus, the probability of a malicious party generating AFE shares which lead to Decode-OR outputting 1 is $1 - 1/p$.

We use $\Pr(b)$ to denote the probability that Decode-MAX outputs b, and $\Pr(\text{invalid})$ to denote the probability that Decode-MAX outputs an invalid bit vector. Decode-MAX outputs b means that the decoded bit vector is $[\underbrace{1,\ldots,1}_{b},\underbrace{0,\ldots,0}_{\ell-b}]$. Thus, $\Pr(b) = (1/p)^{\ell-b} \cdot (1-1/p)^b$.

$$\begin{aligned}\Pr(0)+\ldots+\Pr(\ell-1) &< (1/p)^{\ell}+\ldots+(1/p)^{1}\\ &= \frac{\frac{1}{p}-(\frac{1}{p})^{\ell+1}}{1-\frac{1}{p}}\\ &< \frac{1}{p-1}\\ \Pr(\ell)+\Pr(\text{Invalid}) &= 1-(\Pr(0)+\ldots+\Pr(\ell-1))\\ &\geq 1-\frac{1}{p-1}\end{aligned}$$

In our construction p is a large prime, so $1-\frac{1}{p-1}\approx 1$.

Proof of Lemma 3. From Lemma 2 (and its proof), if a malicious party uses a bogus share for one of the bits, Decode-OR outputs 1 with probability $\geq 1-1/p$. Thus, if a malicious publisher or auxiliary server ever sends a bogus AFE share, Decode-OR would output 1 with high probability, leading to that candidate w becoming the winner. The auxiliary server and publisher then need to get acknowledgment from bidder w on whether its bid b_w is b^*. If w is honest and $b_w \neq b^*$, the auction aborts. If w is honest and $b_w = b^*$, w is the real winner, as b^* is greater than or equal to the real highest bid among all honest bidders with high probability. If w is malicious, it could abort or can choose to be the winner at its own discretion. If the latter, it must pay the second-highest bid (sale price).

Proof of Lemma 4. We consider two cases: (1) bidder w is honest and the real winner (highest bidder); (2) bidder w is a malicious bidder and not the highest bidder. We denote b as the computed sale price in Step 5.

For case (1), if the two auctioneers do not misbehave and use the correct inputs from bidders to compute Step 5, then b is the real sale price (i.e., the second highest bid) among all bidders. If a malicious auctioneer (auxiliary server or publisher) sends a sum vector share that is not computed correctly from bidders' inputs (i.e., the malicious auctioneer sends a sum vector share that is not $\sum_{i=1}^{N} M_i^1 - M_w^1$) in Step 5 when computing the sale price, b would be ℓ with high probability (Lemma 2).

For case (2), if the two auctioneers do not misbehave and use the correct inputs from bidders to compute Step 5, Step 5 finds the highest bid among all bidders excluding bidder w. Thus, b equals the highest bid among all bidders. If a malicious auctioneer sends an incorrect sum vector share (not computed correctly from the inputs of all bidders) in Step 5, then b is ℓ with high probability (Lemma 2).

B Proof for Addax's security properties

This section proves that Addax meets its security properties, which include auction completeness, soundness, and privacy.

Completeness. When all parties are honest, Addax's completeness relies on the probability of Decode-MAX being successful when it is used to find the highest bid and the sale price. Further, it relies on the probability of Decode-OR being successful when finding the id of the winner and the second highest bidder (recall this happens interactively by calling Decode-OR on a particular entry in the AFE sum vector of a candidate winner). In the worst case, Decode-OR might be run on up to n candidate winners ($2n-1$ times for candidate winners and sale price bidders in the interactive variant). The probability of success is $\geq (1-1/p)^{2\ell}\cdot(1-1/p)^{n}$, and is $\geq (1-1/p)^{2\ell}\cdot(1-1/p)^{2n-1}$ in the interactive variant.

Soundness. There are three scenarios which result in an incorrect outcome: (1) publisher and auxiliary server are honest and some bidders are malicious; (2) either the publisher or the auxiliary server is malicious and all bidders are honest; (3) either the publisher or the auxiliary server is malicious and some bidders are malicious and colluding with the malicious auctioneer.

When bidders are malicious, they can: (A1) encode bids into an invalid unary bit vector (e.g., [1,0,0,1]), then generate AFE shares for such invalid unary bit vector and submit them to the publisher and auxiliary server; (A2) provide inconsistent commitments which are not commitments to the AFE it generates or provide inconsistent hash values of their commitments; (A3) claim to be the winner even when they are not. When one of two computing servers is malicious, it could: (B1) send incorrect sum vectors for the highest bid or sale price; (B2) send incorrect AFE shares when finding the winner or the bidder of the sale price.

All malicious behaviors above except (A1) would lead to failure of verification using commitments with high probability due to Pedersen commitments being computationally binding and the hash function being collision resistant. Specifically, after all bidders send a hash of their commitments, they are bound to their AFE vectors. When the random seed used to find the winner and sale price bidder is fixed, the winner and the sale price bidder (in the interactive variant) are also bound. This effectively fixes the outcome of the auction.

(A1) may still pass verification but the outcome of the auction will still be correct since we do not explicitly check whether inputs from all bidders are valid or not. In the non-interactive auction protocol, Addax can treat the highest index with bit one of the decoded bit vector as the bid of the bidder (e.g., [1,0,1,0] corresponds to bid 3). Thus, an invalid AFE vector does not affect the outcome of the auction. And we only need to check the case of (A1) in the interactive variant. If a bidder who submitted an invalid AFE vector does not become the winner or the bidder of the sale price, then they do not affect the auction's outcome. Thus, auditors need only check whether the winner and bidder of the sale price provided valid AFE vectors. We discuss how to do this in Appendix C.

Privacy. We will prove Addax's privacy guarantees using a simulation proof [70]. A simulation proof is done by first defining an ideal functionality $\mathcal{F}$. One can think of it as the function that one would run if one had access to a trusted third party. This ideal functionality will provide some output that is avail-

able to everyone, but it will keep all inputs and internal values secret. We want to show that a protocol is as good as the ideal functionality in terms of what information it leaks: anything that an adversary can learn from interacting and observing the output of Addax, the adversary can learn from interacting and observing the output of the ideal functionality. To prove this, we build a simulator *Sim* that interacts with the ideal functionality $\mathcal{F}$ and obtains only the outputs that $\mathcal{F}$ provides without having access to the inputs of the honest parties. If the simulator can produce a *view* (a transcript of all messages sent and received by all parties) that is computationally indistinguishable from the view produced by the execution of the original protocol, we say that the protocol is as secure as the ideal functionality.

To show the security of Addax, we first define a variant that we call Addax-V. This variant differs from the original Addax in that instead of deferring *all* verification to after the protocol finishes, Addax-V verifies the outcome of each computation step (i.e., the highest bid, the winner, and the sale price) immediately after the step completes. If at any step verification fails, the protocol stops without moving forward. Note that we could deploy Addax-V itself, but it would be inefficient; Addax instead moves the verification to the asynchronous step so that it is not part of the critical path of real-time ad auctions.

We will show that Addax-V is as secure as the ideal functionality $\mathcal{F}$; then we will prove that the original Addax protocol with deferred verification is as secure as Addax-V.

Below we give Addax-V's protocol. For simplicity, we omit the exchange of hash values between P_1 and P_2 before sending messages, and whenever Decode-MAX outputs an invalid bit vector, it is assumed that P_1 or P_2 aborts. We also assume that the two auctioneers find the winner (or bidder of sale price) sequentially starting from the first bidder and stopping when the winner (or bidder of sale price) is found.

Addax-V's auction protocol

Step 1 (Bidders encode and send AFE shares):

- Each bidder i among the n bidders encodes its bid as a V-AFE vector M_i, splits it into additive shares M_i^1 and M_i^2, and generates commitments C_i (§5.1).
- Bidder i sends M_i^1 to P_1 and M_i^2 to P_2, and C_i to both P_1 and P_2.

Step 2 (Compute highest bid):

- P_1 sets $s_1 = \sum_{i=1}^{n} M_i^1$; P_2 sets $s_2 = \sum_{i=1}^{n} M_i^2$.
- P_1 and P_2 exchange s_1 and s_2, compute $S = s_1 + s_2$, and run Decode-MAX on S to get b^*.
- P_1 and P_2 use commitments to verify whether b^* is correct (§5.2) and abort if it fails.

Step 3 (Find winner):

For $i = 1$ to n, P_1 and P_2 repeat the following:

- P_1 sends $M_i^1[b^*]$ to P_2, and P_2 sends $M_i^2[b^*]$ to P_1.
- P_1 and P_2 set $\beta_i = \text{Decode-OR}(M_i^1[b^*] + M_i^2[b^*])$.
- If β_i is 1, then i is the winner (set $w = i$); else continue. If $i = n$ and $\beta_i = 0$, then P_1 and P_2 abort.
- P_1 and P_2 ask bidder w if its bid is b^*. If w says no, P_1 and P_2 abort. Else, P_1 and P_2 use commitments to validate w is correct (§5.2) and abort if it fails.

Step 4 (Compute sale price):

- P_1 sends $m_1 = \sum_{i=1}^{n} M_i^1 - M_w^1$ to P_2, and P_2 sends $m_2 = \sum_{i=1}^{n} M_i^2 - M_w^2$ to P_1.
- P_1 and P_2 compute $sp = \text{Decode-MAX}(m_1 + m_2)$.
- P_1 and P_2 use commitments to verify whether sp is correct (§5.2) and abort if it fails.

Note that whenever there is a party that sends *abort*, the protocol terminates and all parties are notified. The detailed process of termination works as follows.

If an auctioneer (either P_1 or P_2) wants to send *abort*, it directly sends *abort* to all bidders and another auctioneer. If a bidder wants to send *abort* when asked whether b^* is its bid, it replies "no" and sends *abort* to the two auctioneers. The two auctioneers then forward the *abort* message to all the bidders.

Now we define an *ideal functionality* that captures the privacy properties of Addax-V's protocol. Let $n = h + k$ be the total number of bidders, where the first h bidders are honest (non-adversarial) and the last k bidders are malicious (actual position is irrelevant). The two auctioneers are denoted as P_1 and P_2. Without loss of generality, we assume an adversary that corrupts P_1 and k bidders. The ideal functionality is:

Ideal functionality $\mathcal{F}$ of Addax-V's auction protocol

Inputs: h bids $b_1, \ldots, b_h$ from h honest bidders and a *cheat* message (an integer from 1 to 4) from P_1.
Outputs: (b^*, w, sp) are computed as:

- $b^* = \max(b_1, \ldots, b_h)$.
- w, such that $b_w = b^*$ while $b_1 \neq b^*, \ldots, b_{w-1} \neq b^*$ (i.e., w is the first bidder whose bid is b^*).
- $sp = \max(b_1, \ldots, b_{w-1}, b_{w+1}, \ldots, b_h)$ (i.e., the maximum bid excluding b_w).

$\mathcal{F}$ conditionally outputs the above computed values depending on the value of *cheat*.

- When *cheat* is 1: $\mathcal{F}$ outputs b^* to P_1 and nothing to P_2.
- When *cheat* is 2: $\mathcal{F}$ outputs (b^*, w) to P_1 and b^* to P_2.
- When *cheat* is 3: $\mathcal{F}$ outputs (b^*, w, sp) to P_1 and (b^*, w) to P_2.
- When *cheat* is 4: $\mathcal{F}$ outputs (b^*, w, sp) to both P_1 and P_2.

Note that the ideal functionality $\mathcal{F}$ also provides an interface

to take an *abort* message as input which allows terminating the execution of a protocol by the simulator.

Now we see how the ideal functionality $\mathcal{F}$ captures the privacy properties of Addax-V's protocol. In the real world execution of Addax-V's protocol, there are five different cases: (1) verification in Step 2 fails; (2) verification in Step 2 passes, verification in Step 3 fails, and P_1 does not learn the real winner; (3) verification in Step 2 passes, verification in Step 3 fails, and P_1 learns the real winner; (4) verification in Step 2 and 3 passes but verification in Step 4 fails; and (5) all verification passes.

When *cheat* message to $\mathcal{F}$ is set to 1, the outputs of $\mathcal{F}$ capture cases (1) and (2) above. And when *cheat* is set to other values, the outputs of $\mathcal{F}$ capture the remaining three cases, respectively. Thus, the outputs of $\mathcal{F}$ capture all different cases of real world execution of Addax-V.

Lemma 7. The Addax-V auction protocol securely implements ideal functionality $\mathcal{F}$ under the assumption that the commitment scheme is binding and hiding and that p is large enough to ensure that Decode-OR and Decode-MAX produce incorrect outputs with negligible probability.

Proof. We now build a simulator *Sim* that interacts with the ideal functionality $\mathcal{F}$ which at most leaks the highest bid b^*, the winning bidder's index w, and the sale price sp. Note that the simulator below simulates all the five different cases (when the protocol aborts in different steps) of real world execution in Addax-V. We use $\mathcal{A}$ to denote the adversary who can corrupt P_1 and k malicious bidders. Note that P_1 and P_2 are symmetric and do the same computation in the protocol. Thus, the proof below also applies to an adversary who corrupts P_2.

Simulator *Sim*

Step 1 (Generate random V-AFE vectors):

- $\mathcal{F}$ gives its output to *Sim*. This output depends on which of the five cases of the real world execution we are simulating (based on the *cheat* message).
- For the h honest bidders, *Sim* assigns a bid to each of them in such a way that one of the bids is b^* and all other bidders' bids are set as smaller than or equal b^*.
- If w is included in the output of $\mathcal{F}$, the honest bidder w's bid is the one that is set to b^*.
- If sp is included in the output of $\mathcal{F}$, a random honest bidder's bid (excluding w) is set to sp and all other bids are set as smaller than or equal to sp.
- *Sim* encodes the h honest bidders' bids into h V-AFE vectors and generates commitments (§5). It then splits the V-AFE vectors into additive shares, and sends one of the V-AFE shares and the commitment of each honest bidder to $\mathcal{A}$.
- $\mathcal{A}$ generates V-AFE shares and commitments for the k malicious bidders. It sends one of the V-AFE shares and the commitment of each of the k malicious bidders to *Sim*.
- At this point, $\mathcal{A}$ has shares $M'^1_1, \ldots, M'^1_n$ and *Sim* has shares $M'^2_1, \ldots, M'^2_n$. They both get commitments $C'_1, \ldots, C'_n$.

Step 2 (Compute highest bid):

- *Sim* computes $s'_2 = \sum_{i=1}^{n} M'^2_i$, and sends it to $\mathcal{A}$.
- $\mathcal{A}$ sends a V-AFE vector s'_1 (s'_1 should be $\sum_{i=1}^{n} M'^1_i$ if it follows the protocol, or some vector generated based on its cheating strategy) to *Sim*.
- *Sim* computes $b'^* = \text{Decode-MAX}(s'_1 + s'_2)$.
- If $b'^* \neq b^*$, *Sim* sends *abort*.

Step 3 (Find winner):

For $i = 1$ to n, *Sim* and $\mathcal{A}$ repeat:

- *Sim* sends $M'^2_i[b'^*]$ to $\mathcal{A}$, and $\mathcal{A}$ sends v_i (v_i should be $M'^1_i[b'^*]$ if it follows the protocol, or a vector generated based on its cheating strategy) to *Sim*.
- *Sim* computes $\beta'_i = \text{Decode-OR}(M'^2_i[b'^*] + v_i)$.
- If $\beta'_i = 1$, then $w' = i$; else continue to the next round. If $i = n$ and $\beta'_i = 0$, *Sim* sends *abort* to $\mathcal{A}$.
- After finding w', if bidder w' is malicious, *Sim* asks $\mathcal{A}$ whether the bid of w' is b'^*. If $\mathcal{A}$ replies with no, *Sim* sends *abort*.
- If bidder w' is an honest bidder and $w' \neq w$, *Sim* sends *abort* to $\mathcal{A}$.
- If $w' \neq w$, *Sim* sends *abort* to $\mathcal{A}$.

Step 4 (Compute sale price):

- *Sim* sends $m'_2 = \sum_{i=1}^{n} M'^2_i - M'^2_{w'}$ to $\mathcal{A}$, and $\mathcal{A}$ sends m'_1 (m'_1 should be $\sum_{i=1}^{n} M'^1_i - M'^1_{w'}$ if it follows the protocol, or a V-AFE vector generated based on its cheating strategy) to *Sim*.
- *Sim* computes $sp' = \text{Decode-Max}(m'_1 + m'_2)$.
- If $sp' \neq sp$, *Sim* sends *abort*.

Note that whenever Decode-MAX outputs an invalid bit vector, we assume that *Sim* sends *abort*. When *Sim* wants to send *abort*, it notifies $\mathcal{A}$ and the ideal functionality $\mathcal{F}$, and $\mathcal{F}$ forwards *abort* and outputs to all honest parties. When $\mathcal{A}$ wants to abort, it sends *abort* to *Sim*, *Sim* forwards *abort* to the ideal functionality $\mathcal{F}$, and $\mathcal{F}$ forwards *abort* and outputs to all honest parties. In the simulation, as long as each party receives one *abort* message, it terminates. Finally, for simplicity whenever $\mathcal{A}$ and *Sim* exchange messages (e.g., V-AFE shares), *Sim* asks $\mathcal{A}$ to send first.

Analyzing the views. When *cheat* is set to 3 or 4 in the ideal world (ideal functionality), which corresponds to the real world (Addax-V) protocol proceeds to the step of computing the sale price, $\mathcal{A}$ learns the entire view in both worlds. The view of $\mathcal{A}$ in the ideal world is: $\{\sum_{i=1}^{n} M'^2_i, w', b'^*, sp', M'^2_{w'}, M'^2_1[b'^*], \ldots M'^2_{w'}[b'^*], M'^1_1, \ldots, M'^1_n, C'_1, \ldots, C'_n\}$. In

the real world (Addax-V), $\mathcal{A}$'s view includes: $\{\sum_{i=1}^{n} M_i^2, w, b^*, sp, M_w^2, M_1^2[b^*], \ldots M_w^2[b^*], M_1^1, \ldots, M_n^1, C_1, \ldots, C_n\}$.

When *cheat* is set to 1, $\mathcal{A}$'s view in ideal world only includes: $\{\sum_{i=1}^{n} M'^2_i, b'^*, M'^1_1, \ldots, M'^1_n, C'_1, \ldots, C'_n\}$. In the real world, the corresponding view of $\mathcal{A}$ includes: $\{\sum_{i=1}^{n} M_i^2, b^*, M_1^1, \ldots, M_n^1, C_1, \ldots, C_n\}$.

When *cheat* is set to 2, $\mathcal{A}$'s view in ideal world only includes: $\{\sum_{i=1}^{n} M'^2_i, w', b'^*, M'^2_1[b'^*], \ldots M'^2_{w'}[b'^*], M'^1_1, \ldots, M'^1_n, C'_1, \ldots, C'_n\}$. In the real world, the corresponding view of $\mathcal{A}$ includes: $\{\sum_{i=1}^{n} M_i^2, w, b^*, M_1^2[b^*], \ldots M_w^2[b^*], M_1^1, \ldots, M_n^1, C_1, \ldots, C_n\}$.

V-AFE shares are uniformly random elements in $\mathbb{Z}_p$, and commitments to V-AFE vectors are hiding, thus they have the same distribution. For w, b^*, sp and w', b'^*, sp', their distributions are also identical. M_w and $M'_{w'}$ are both generated by encoding bid b^* into V-AFE vectors, thus having the same distribution. $\sum_{i=1}^{n} M_i - M_w$ and $\sum_{i=1}^{n} M'_i - M'_{w'}$ are both generated following the requirement that the highest bid among all remaining bidders (excluding bidder w and bidder w') is sp, thus having the same distribution. $M'_1[b'^*], \ldots, M'_{w'-1}[b'^*]$ and $M_1[b^*], \ldots, M_{w-1}[b^*]$ are zeros. These say that in the simulation, bidder 1 to bidder $w' - 1$ are not the winner, and in the real world protocol, bidder 1 to bidder $w - 1$ are not the winner. $M'_{w'}[b'^*]$ and $M_w[b^*]$ are encodings of bit 1.

As a result of the above exhaustive case analysis, both the real world view and the ideal world view are identically distributed. Consequently, an adversary for Addax-V learns nothing beyond what is revealed by the ideal functionality. □

Lemma 8. Addax's protocol does not leak more information to the adversary than the variant Addax-V.

Proof. The only difference between the variant and the original protocol is that in the original protocol, the adversary learns the entire view of $\{\sum_{i=1}^{n} M_i^2, w, M_w^2, M_1^2[b^*], \ldots M_w^2[b^*], M_1^1, \ldots, M_n^1, C_1, \ldots, C_n\}$, while in Addax-V, it stops after aborts in Step 2 or 3, and only learns a partial view.

In the original protocol, the malicious auctioneer always learns the correct highest bid regardless of how it behaves, as the honest auctioneer always sends the correct sum of AFE shares. From Lemma 3, if an incorrect bidder is claimed as the winner and the protocol does not abort after finding winner, the incorrect winner w' must be malicious. And in the original protocol, the auctioneers would proceed to compute the sale price with the *incorrect* winner w'. In this case, the AFE vector of the malicious bidder, $M'_{w'}$, is revealed to the auctioneers and auditors.

In Step 4, when computing the sale price, the adversary receives $\sum_{i=1}^{n} M'^2_i - M'^2_{w'}$ from the honest party P_2. Adversary knows $M'^2_{w'}$ since it's from a malicious bidder, and $\sum_{i=1}^{n} M'^2_i$ is already learned in Step 2 to compute the highest bid. Thus, in the original protocol, when the protocol proceeds to Step 4 with an incorrect winner w', it can only learn the same amount of information as what it learned in Step 2 (Lemma 4).

Now we can conclude that the original protocol does not leak more information about *honest* bidders' bids compared to the Addax-V variant, where verification is not deferred. □

C Safeguarding interactive Addax

In the non-interactive protocol, the underlying value of a bit encoding is defined as the rightmost position among all the non-zero values. For instance, both [1,1,1,0] and [1,0,1,0] are the encodings of value 3. The above encoding neither brings issues for privacy nor integrity, but under the interactive variant, this encoding does not work. See an example below.

Suppose a malicious bidder submits an invalid AFE vector which yields an invalid bit vector [1,1,0,1], and all other bidders submit [1,0,0,0]. When finding the winner interactively, two auctioneers will first check the first and the third positions. Since the corresponding results are 1 and 0, the next position to be checked should lie in between the first and the third entry, which, in this example, is the second position. As a result, the highest bid found is 2, which means the bidder wins the auction with bid 2.

Therefore, in the interactive variant, Addax checks whether the AFE vectors of the winner and the bidder of the sale price correspond to valid unary bit encodings. To this end, we add the following two extra steps: (1) an asynchronous procedure for finding sale price bidder (we need this to find *whose* bit encoding to validate); and (2) proving that a bidder's AFE vector is valid without leaking its original AFE vector (we need this to avoid leaking the *third-highest bid*). Note that in the interactive variant of the first-price auction, we also need to prove the winner's AFE vector is valid while hiding its original value so the *second-highest bid* is not leaked.

C.1 Asynchronous: find sale price bidder

The invalid encoding as above impacts the outcome of an auction (if it is not aborted) if the malicious bidder is the winner or the bidder of sale price (if the malicious bidder is neither of these, it has no effect). Therefore, Addax requires validating the AFE vector of the sale price bidder. To this end, Addax has to find its bidder id, though not the real identity. We make the tradeoff of leaking such bidder id in order to keep the completion of the auction within hundreds of milliseconds with this extra asynchronous step.

Specifically, the auxiliary server and publisher first find its bidder id by running Step 4 of Section 4.4 on all AFE vectors except the winning one. This is done *after* the auction is complete and off the latency-critical path (hence why we say this is an *asynchronous* step). Verifiers can check, ex post facto, whether the AFE encodings of the second-highest bidder were valid or not. Similarly to Lemma 3, the bidder found in this step is either the real bidder of the sale price or a malicious bidder.

C.2 Checking the validity of a V-AFE vector

Insecure strawman. To check the validity of V-AFE vectors we can let the vectors be revealed in the clear and check the validity by ensuring they are in the right unary bit form, and are consistent with the Pedersen commitments. However, if we do this for the second-highest bidder's V-AFE vector, this would result in leaking the third-highest bid—substracting the V-AFE vectors of the winner and sale price bidder from the overall sum vector (**M**) reveals the sum of V-AFE vectors of the remaining bidders, thus leaking their maximum bid. We provide the following construction to check validity of the V-AFE vector of the sale price bidder without leakage.

Secure check for AFE validity. In a nutshell, the idea is that in Step 1 of the auction protocol (§ 4.4), bidders additionally generate a V-AFE vector **vr** with a random mask **r** for their original V-AFE vector **v**, such that we can still check the validity of **v** by utilizing **vr**.

Specifically, the bidder first generates the random mask **r**, which is a vector of ℓ random non-zero elements in $\mathbb{Z}_p$ and **vr** is the element-wise product between the two $\{r_1 \cdot v_1, \ldots, r_\ell \cdot v_\ell\}$. Since $r_i \cdot v_i = 0$ if and only if $v_i = 0$ (for $1 \leq i \leq \ell$), therefore, as long as **vr** is decoded to be a valid bit vector, so is **v**.

When generating commitments for V-AFE vector **v**, a bidder also generates commitments to **vr** and a proof that the underlying messages in the above two commitments indeed differ by a multiplicative factor **r**. Concretely, the bidder uses a *Sigma protocol* [78], which is type of very simple and efficient zero-knowledge proof (with the *Fiat-Shamir heuristic* [53] it is made into a non-interactive proof) to prove that, for each element cr_i in the commitments of **vr** and c_i in commitments of **v**, there exists a non-zero r_i which satisfies $cr_i = {c_i}^{r_i}$. We give details about the above zero-knowledge proof in Appendix C.4. Appendix C.3 proves the binding and hiding properties of the commitment to **vr** (which is slightly different than standard Perdersen commitments).

During verification, the sale price bidder reveals only its **vr**, and an auditors: (1) verify whether **vr** is consistent with its commitment; (2) check that the decoded result of **vr** is a valid AFE encoding; (3) verify the zero-knowledge proof with respect to the commitments of **v** and **vr**.

A key optimization to this process is as follows. Observe that in the interactive variant the two auctioneers only compute the sum vector of partial entries (e.g., they may only use the first 100 entries to compute based on the lower/upper bounds derived). Thus, the sale price bidder need only hide its original AFE vector for those entries (by using the above **vr** and zero-knowledge proofs for those entries); the sale price bidder can actually reveal its original AFE encoding for the other entries without need for proofs. The result is that auditors need only check the zero-knowledge proofs for the partial entries used to compute the sale price.

Remark. First, the above approach only hides non-zero values in **v**, (i.e., it leaks whether a value in **v** is zero or not). This is because for entries with zero values in the V-AFE vector of sale price bidder, those entries in the sum vector are also zeros, as the sum vector computed in Step 5 (§4.4) decodes to the bid of the sale price bidder. Thus, learning those entries with zero values does not leak the bids of any other bidders.

Second, an adversary may use zeros in the random mask **r** to flip non-zero values into zero, which might turn an invalid AFE vector into a valid one. To check that a bidder did not use zero as random mask, for each tuple vr_i in **vr**, we check that the second element of vr_i is not zero.

C.3 Commitment to a V-AFE tuple with random mask

Given a V-AFE tuple $v = (a, b)$, its tuple with a random mask r is $vr = (r \cdot a, r \cdot b)$. The commitment is as follows. Let $\mathbb{G}$ be a group of prime order p and let $\{g, h\}$ be two random generators $\{g, h\}$ of $\mathbb{G}$. The commitment is $g^{r \cdot a} \cdot h^{r \cdot b}$.

The reason why the above is slightly different than standard Pedersen commitments is that the randomness (the exponent of h in Pedersen) is sampled uniformly at random and independent of the exponent of g, whereas here there is a relationship between the exponent of g (which is $r \cdot a$) and the exponent of h (which is $r \cdot b$).

Lemma 9. Let $c = g^{r \cdot a} \cdot h^{r \cdot b}$ be a commitment to $vr = (r \cdot a, r \cdot b)$. Then c perfectly hides vr, and computationally binds vr if the discrete logarithm problem is hard in $\mathbb{G}$.

Perfect hiding. The commitment perfectly hides both r and a. Let $x \in Z_p$ be an element such that $g = h^x$ (this is well defined since h is a generator and hence there exists an x such that $h^x = g$). Given r, a, b, for any a' there exist r' and b' such that $g^{r \cdot a} \cdot h^{r \cdot b} = g^{r' \cdot a'} \cdot h^{r' \cdot b'}$. And r', b' satisfy that $r' = \frac{x \cdot r \cdot a + r \cdot b}{x \cdot a' + b'}$. Thus, the commitment hides a. Using a similar proof, we can show that the commitment also hides r.

Binding. We now prove that for a message $m = r \cdot a$, the commitment $g^{r \cdot a} \cdot h^{r \cdot b}$ binds m. Specifically, our goal is to show that, if an adversary can find a different message $m' = r' \cdot a'$ and some randomness b' such that $g^{r \cdot a} \cdot h^{r \cdot b} = g^{r' \cdot a'} \cdot h^{r' \cdot b'}$, then it can break discrete log for the instance $(g, h = g^x)$. Note here we only assume $m' \neq m$, and it does not mean $r' \neq r$ or $a' \neq a$.

Suppose the adversary finds such r', a', b' as above, which implies $r \cdot a + r \cdot b \cdot x = r' \cdot a' + r' \cdot b' \cdot x$ where $h = g^x$ for some x (i.e., $r \cdot a - r' \cdot a' = (r' \cdot b' - r \cdot b) \cdot x$). If $r' \cdot b' = r \cdot b$, then the above equation means $r \cdot a = r' \cdot a'$, which contradicts with our assumption that $m' \neq m$. If $r' \cdot b' \neq r \cdot b$, then the adversary can compute $x = (r' \cdot b' - r \cdot b)^{-1}(r \cdot a - r' \cdot a')$, which means now the adversary solves discrete log for instance $(g, h = g^x)$.

C.4 Zero-knowledge proof of commitment relation

The prover, which is the bidder in our case, wants to prove that the there exists a secret element $r \in \mathbb{Z}_p$ such that commitments cr and c satisfy $cr = c^r$. Figure 8 gives the pseudocode for how the prover generates the non-interactive proof π. The prover first samples a random element r' from $\mathbb{Z}_p$, and computes $u =$

Prover$(c, r, cr = c^r)$	**Verifier**$(c, cr = c^r, \pi = (u, v, z))$
$r' \xleftarrow{R} \mathbb{Z}_p$	$v \stackrel{?}{=} H(c, cr, u)$
$u \leftarrow c^{r'}$	$c^z \stackrel{?}{=} u \cdot cr^v$
$v \leftarrow H(c, cr, u)$	
$z \leftarrow r' + v \cdot r$	
output $\pi = (u, v, z)$	

FIGURE 8—Non-interactive zero-knowledge proof of knowledge of secret r using the Fiat-Shamir Heuristic. H is a random oracle and its range is $\mathbb{Z}_p$ (heuristically instantiated with a secure hash function).

$c^{r'}$. The prover then computes $H(c, cr, u)$ as the challenge v. H is modeled as a random oracle but heuristically instantiated with a collision-resistant hash function. Finally, the prover computes $z = r' + v \cdot r$. The prover sends $\pi = (u, v, z)$ to the verifier, and the verifier checks π as described in Figure 8.

C.5 Proofs of lemmas 5 and 6

Below we prove the Lemmas for the interactive variant of Addax which leverages the safeguards described in this section.

Proof of Lemma 5. The non-interactive protocol and interactive variant differ in the following two places: (1) in interactive variant, computing MAX only uses partial entries; (2) in interactive variant, Addax checks validity of winner and sale price bidder's AFE vectors with additional verification materials (Appendix C.2).

For (1), in the non-interactive protocol, adversary learns the entire sum vector $\mathbf{M}$, while in the interactive variant, it only learns $r \cdot \lceil \ell^{1/r} \rceil$ entries of $\mathbf{M}$ (§6.1). It therefore leaks no more information about bids than the non-interactive protocol.

For (2), in the interactive protocol, the winner w reveals its full AFE vector, and the sale price bidder needs to provide materials as in Appendix C.2. In the non-interactive protocol, w's full AFE vector can already be inferred from Step 3 (which computes $\mathbf{M}$) and Step 5 (which computes $\mathbf{M} - M_w$), so this leaks no additional information. And additional materials provided in the interactive variant leak no more information of bids than the sale price as detailed in Appendix C.2.

Proof of Lemma 6. Integrity is ensured in the non-interactive protocol as per Theorem 1. The only difference in the interactive variant is that if either the winner or the sale price bidder submits an invalid AFE vector, it can lead to the k-ary search in the interactive protocol to converge to an incorrect value. As we discuss in Appendix C, Addax adds checks to ensure that the AFE vectors of the winner and sale price bidder are both correct, and hence the k-ary search converges to the same value as in the non-interactive protocol.

D Subsets of faulty parties

An auction may be aborted during the online phase, or deferred verification may fail due to a lack of enough materials on the public log (e.g., commitments, sum shares), or due to inconsistent materials such as the bidder sending invalid shares to auctioneers, or an auctioneer claiming it received one value when a bidder submitted another. Figure 9 gives pseudocode for how Addax narrows down the parties at fault. Below is a text explanation for the pseudocode.

Auction aborts. An auction may abort for two reasons: (1) Decode-MAX outputs an invalid bit vector; or (2) the chosen winner or sale price bidder claims that their bids do not equal the found highest bid or sale price. For case (1), if verification on the decoded sum vectors passes, then it implies that some bidders provided invalid AFE vectors. Addax assigns blame to all participating bidders as potentially malicious. If verification of the sum vectors fails, Addax assigns blame to all bidders, the publisher, and the auxiliary server. For case (2), auditors check the shares revealed by the publisher and auxiliary server, and check whether they are consistent with the corresponding commitments. If they are not consistent, Addax assigns blame to the specific bidder (winner or second highest bidder), publisher, and auxiliary server. If it is consistent, and that entry decoded to 0, Addax assigns blame to the auxiliary server and the publisher; if the entry decoded to 1, Addax assigns blame only to the corresponding bidder.

Lack of materials. Participants may not upload all required materials to the public log, which prevents auditors from verifying the auction. Bidders are bound to their bidder ids which should be uploaded by the publisher and auxiliary server. An auditor can easily tell who did not upload the required materials and assign blame to that set of participants.

Inconsistent views between publisher and auxiliary server. The publisher and auxiliary server may provide an inconsistent view for messages they send and receive. For example, the publisher may claim that it receives sum vector M from the auxiliary server, while the auxiliary server claims that it sends M' to the publisher. In this case, Addax assigns blame to both publisher and auxiliary server. If publisher and auxiliary server upload different views of hash values of certain bidder, Addax assigns blame to the publisher, auxiliary server, and the specific bidder, as the bidder may send inconsistent hash values on purpose.

Inconsistency between hash values and commitments. Auditors may find that hash values of commitments uploaded by publisher and auxiliary server are inconsistent with that uploaded by the bidder. Addax assigns the blame to that particular bidder (since publisher and auxiliary server are assumed to not collude).

Inconsistent AFE sum vectors or bit encodings. Verification on sum vectors or revealed bit encodings to find the winner may fail. If verification on sum vectors fails, Addax assigns blame to publisher, auxiliary server and all bidders. If verification on specific bidder's bit encoding fails, Addax assigns blame to publisher, auxiliary server and the specific bidder.

Invalid AFE vector. The winner needs to upload its full AFE vector and the bidder of sale price needs to upload its full AFE

```
1:  function AssignBlame(materials, abort, auctioneers, bidders)
2:    # Check if abort happens
3:    if abort ≠ null then
4:      if abort.decodeMax == true then
5:        if verifySumvec(materials) == true then
6:          blame(bidders)
7:        else
8:          blame(auctioneers, bidders)
9:      else if abort.findBidder == true then
10:       if verifyBit(materials.bidders[abortId]) == false then
11:         blame(auctioneers, bidders[abortId])
12:       else
13:         if decode(materials.bidders[abortId].bitEncoding) == 0 then
14:           blame(auctioneers)
15:         else
16:           blame(bidders[abortId])
17:   # Check all materials are not missing
18:   for auc in auctioneers do
19:     if materials.auc == null then
20:       blame(auc)
21:   for b in bidders do
22:     if materials.b == null then
23:       blame(b)
24:   # Check inconsistency between auctioneers
25:   if inconsistent(materials.auctioneers.sumvec) then
26:     blame(auctioneers)
27:   for b in bidders do
28:     if inconsistent(materials.auctioneers.hash[b]) then
29:       blame(auctioneers, b)
30:   # Check inconsistency between hash and commitments
31:   for b in bidders do
32:     if inconsistent(materials.b.hash, materials.b.commitment) then
33:       blame(b)
34:   # Verify sum vectors and bit encodings
35:   if verifySumvec(materials) == false then
36:     blame(auctioneers, bidders)
37:   for b in bidders do
38:     if verifyBitEncoding(b.materials) == false then
39:       blame(auctioneers, b)
40:   # Validate AFE vector of winner and sale price bidder
41:   if validate(materials.bidder.AFE) == false then
42:     blame(bidder)
```

FIGURE 9—Pseudocode of how to assign blames in Addax, see texts for more details.

vector with random mask (§C.2), the commitments and non-interactive zero-knowledge proofs. An auditor needs to check the winner's AFE vector decodes to be a valid bit vector. Also, an auditor must check whether the AFE vector with random mask is consistent with the supplied commitments, whether it decodes to be a valid bit vector, and verify the proofs. If the check fails, Addax assigns blame to the specific bidder. If the check passes, even if verification on the sum vectors fails, Addax explicitly knows that these two bidders are honest, and can avoid assigning blame to them. Addax will then protect their identities.

D.1 Narrow down faulty bidders when both auctioneers are honest

There are some cases (e.g., line 8 in Figure 9) where Addax can only assign blame to all of the bidders due to the fact that a malicious auctioneer could collude with bidders. However, if both auctioneers were honest (how one would establish this is orthogonal, though likely hard), Addax can identify the specific bidders who provided inconsistent AFE vectors and commitments. To detect such faulty bidders, each of the two auctioneers computes the commitment over its local V-AFE vector share of a bidder and reveals the commitment. They then verify whether multiplying these two commitments from both auctioneers yields the commitment submitted by the bidder. If not, then Addax can blame the bidder.

E Interacting with the public log

E.1 A brief primer on Algorand

Accounts and Transactions. An account in Algorand consists of a key pair. Transactions include payment, key registration, and asset transferring. Each transaction is created by one account and must be signed with its corresponding secret key.

Smart contracts and application calls. Smart contracts are programs that run on the blockchain with user-defined functionality. Each smart contract is specified with a unique ID. Application calls are transactions used to invoke functions in smart contracts like RPC calls. To interact with a smart contract, an account needs to join the smart contract first. The `opt-in` call allows one account to join one smart contract instance while a `close-out` call allows one account to leave. Addax maintains one smart contract per ad category which includes all advertisers' information in that category. When one advertiser belongs to multiple categories, it joins all related smart contracts. There is no upper bound on the number of accounts that can join one smart contract in Algorand. Bidders invoke application calls defined in the running smart contract to insert, update, or delete their own information.

Indexer. Indexers are special nodes which provide RESTful interfaces to search for transactions or states of certain apps by answering SQL-like queries. For example, it can answer queries to search for all transactions that happened during a certain period in one smart contract instance with a query like: `SELECT * from transactions WHERE appID = {ID of App} AND after-time = {start} AND before-time = {end};`

E.2 Workflow of a deployed smart contract

There are two kinds of smart contracts in Algorand, stateful ones which have their own storage and stateless ones which do not. Storage in smart contracts consists of several key-value pairs which can be read and written. There is *global storage* which maintains the state of the smart contract instance and *local storage*. All opted-in accounts have their own local storage. All storage can be read by anyone and updated via application calls. Application calls for writing local storage can only be invoked by its owner account.

Addax maintains one stateful smart contract per ad category which includes all advertisers in that category. When one advertiser belongs to multiple categories, it joins all related

```
1: function INIT(N)
2:     gs.counter ← 0
3: function CREATE(info)
4:     ls.id ← gs.counter
5:     gs.counter ← gs.counter + 1
6:     ls.info ← info
7: function UPDATE(newInfo)
8:     ls.info ← newInfo
9: function DELETE
10:    delete(ls.info, ls.id)
```

FIGURE 10—Pseudocode of smart contract, *gs* is global state, *ls* is local storage of each advertiser. Init function is called when smart contract is initialized. Create, Update, Delete functions are called by advertisers.

smart contracts. Figure 10 shows the functionalities provided by the deployed smart contracts in Addax. Each smart contract is created by an `INIT` function to initialize an incremental counter starting from zero in global storage. Advertisers can invoke `CREATE`, `UPDATE` and `DELETE` functions. `CREATE` is the `opt-in` application call in Addax that opts an advertiser into the smart contract of its category and write information into the invoker's local storage. Local information of advertisers can include brand name, all categories the advertiser belongs to, domain and port of ad server, protocols and serialization formats supported, etc. The `UPDATE` call is invoked by opted-in advertisers to update their local information. Advertiser invokes `DELETE` to clear all information stored in local storage and leave this smart contract instance.

E.3 Costs of interacting with public log

In this section, we answer the question of costs for interacting with the public log and for querying the indexer. We use the PureStake indexer which provides a REST API that one can use to upload and retrieve information from the Algorand blockchain. PureStake has servers all over the world as they contract with Cloudfront. In our experiments, requests to PureStake that originate from AWS US East (Ohio) contact servers in Ontario. Requests that originate from the AWS US West (California) contact servers in California. Requests that originate from AWS US West (Oregon) contact servers in Oregon. PureStake then takes care of broadcasting the transaction to the Algorand peer-to-peer network.

The size of advertisers' information (§E.1) uploaded to Algorand to participate in Addax is 960 bytes. We experiment with a modest number of advertisers. The reason that we do not have tens of thousands of advertisers is that creating a new advertiser requires creating an Algorand account (new email address, password, account verification, etc.) which is time-consuming. Nevertheless, we semi-automate this painstaking process and generate 1,000 accounts.

Time for advertisers' operations. In Figure 11, we evaluate the time of advertisers' operations (invoking CREATE, UPDATE or DELETE application call) on the Algorand blockchain. We vary whether advertisers belong to one or multiple categories, and also vary the distribution of the number of opted-in advertisers for a given category. Figure 11a shows the results of an advertiser interacting with Algorand where the advertiser belongs to a single category while varying the number of advertisers registered for that category. Figure 11b shows the results when the advertiser belongs to multiple categories, all of which have 500 advertisers registered. Finally, Figure 11c shows the results when the advertiser belongs to six different categories, while each category contains a different number of advertisers. In Scenario 1, each category contains 100 advertisers. In Scenario 2–4, the numbers of advertisers in each category are [50, 100, 100, 100, 100, 150], [50, 50, 50, 100, 150, 200], and [50, 50, 50, 100, 100, 250].

As can be seen, the time for these three operations remains nearly constant when the number of opted-in advertisers in one category grows. It takes about 8–8.5 seconds for invoking one application call to one category (i.e., one smart contract instance). Most of the overhead (about 7 seconds) comes from advertisers waiting for confirmation from the blockchain that this operation has finished successfully. Advertisers can directly invoke application calls and leave without having to wait for confirmation. Indeed, this is what the browser does when uploading its audit materials as described in the Leaving an audit trail paragraph of Section 9.4.

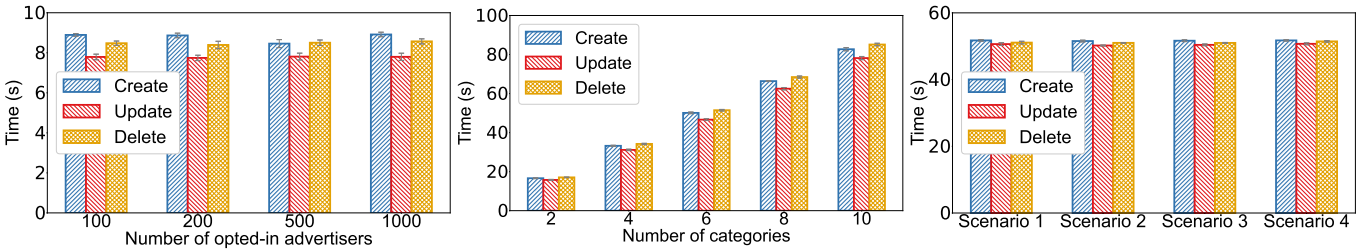
The costs of these operations grow linearly with the number of categories to which an advertiser belongs, regardless of the number of opted-in advertisers in each category.

Time for querying the indexer. We also evaluate the costs for querying the indexer for updates during certain period of time under different scenarios. The time for querying the indexer is also the time to verify the query results from cache servers. Figure 12a shows the time to query indexer for updates of multiple categories. Each category contains 500 advertisers. We simulate 20% updates in each category, namely 100 transactions happened during the period we search for. Figure 12b shows the time to query the indexer for updates of one category with 1,000 advertisers but with different percentages of updates during the period of search.

We find that querying one category generally takes about 0.8 seconds, and this number would grow slightly if the number of total updates (i.e., transactions) grows. This is due to the fact that as the total number of updates grows, the data fetched from the indexer grows as well. Also, the time to query the indexer for updates of multiple categories grows linearly with the number of categories queried. This is because to query N categories, the querier needs to send N requests to the indexer.

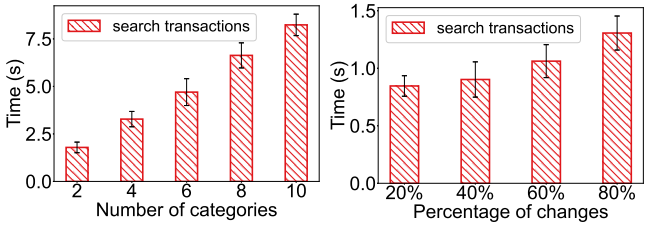
F What about TEE-based solutions

In principle, one might be able to design a solution that leverages TEEs to provide privacy and public verifiability for online ad auctions. However, this is not a trivial task, since TEEs:



(a) (b) (c)

FIGURE 11—Time for advertisers' operations on the Algorand blockchain under different settings (see text).



(a) (b)

FIGURE 12—Time to query indexer for updates in certain period under different scenarios.

- Require the release of code that runs inside the enclave. This includes the auction protocol, the encryption/decryption code to recover plaintext bids, and signing code for creating a proof that can be publicly audited.
- Require careful auditing or formal verification of all the code running inside the enclave to ensure the exchange operator (who is running the TEE) did not inject backdoors or other vulnerabilities that can obviate the TEE.
- Intel SGX in particular requires trusting an Intel cloud server during remote attestation (cloud services like Azure's Attestation Service [29] can also be used). In either case, trusting such servers might not be that different of an assumption than the anytrust model in Addax.
- Require additional mechanisms to prevent replay attacks. For example, suppose an operator runs an auction, invites 10 bidders, and passes as input their 10 encrypted bids to TEE (TEE internally has a key to decrypt bids). The TEE then outputs the winner and sale price in the clear. The operator could then run the same auction again but passing only a subset of the bids (these are all valid encrypted bids under a key known to the TEE). The TEE then outputs a winner and sale price in the clear, so the attacker could quickly discover all bids. In contrast, replay is not possible in Addax since the auction is either completed so both auctioneers forward the result to the publisher, or aborted so at least one honest auctioneer forwards an abort result to the publisher (and the publisher displays a generic ad).

G Compatible user privacy features

One of Addax's goals is to have a flexibile enough design to be compatible with various efforts that aim to improve user-privacy (these efforts are orthogonal but they are also complementary to Addax).

Bidding on groups rather than individuals. User activities are tracked as advertisers need to gather enough information about different users' browsing and purchasing preferences. The information is used by advertisers to decide how to bid for a user. Addax's design is compatible with Google's recent Topics proposal [16] which locally groups users into groups. In particular, Topics enhances the browser to keep track of the user's interest and assigns to the user a Topic identifier. Once this identifier exist, Addax can send this identifier to the selected bidders instead of sending them more demographic information (§7). An advertiser would then decide how to bid for each group of users without learning information about the individual user for which it is bidding. The obvious caveat here is that the current Topics proposal is not perfect and there have been various privacy concerns voiced [5, 6, 17].

Measuring conversions without learning individual's data. Measuring the effectiveness of an ad after the auction is essential for advertisers. However, the current way of measuring conversions leaks users' sensitive information about which websites are visited. There is a recent effort [34] that provides a mechanism to measure the return on investment (ROI) and conversions without requiring the advertiser to learn information about a specific user. At the end of the measurement, advertisers see a differentially private histogram of all users' conversions, which is sufficient for them to determine the effectiveness of their campaigns. In Addax, after the auction finishes, advertisers could apply this approach to privately gather data about conversions for analysis while being more sensible to users' privacy concerns. This mechanism is compatible with Addax as it occurs *after* the auction completes.

SPEEDEX:
A Scalable, Parallelizable, and Economically Efficient Decentralized EXchange

Geoffrey Ramseyer
Stanford University

Ashish Goel
Stanford University

David Mazières
Stanford University

Abstract

SPEEDEX is a decentralized exchange (DEX) that lets participants securely trade assets without giving any single party undue control over the market. SPEEDEX offers several advantages over prior DEXes. It achieves high throughput—over 200,000 transactions per second on 48-core servers, even with tens of millions of open offers. SPEEDEX runs entirely within a Layer-1 blockchain, and thus achieves its scalability without fragmenting market liquidity between multiple blockchains or rollups. It eliminates internal arbitrage opportunities, so that a direct trade from asset $\mathcal{A}$ to asset $\mathcal{B}$ always receives as good a price as trading through some third asset such as USD. Finally, it prevents certain front-running attacks that would otherwise increase the effective bid-ask spread for small traders. SPEEDEX's key design insight is its use of an Arrow-Debreu exchange market structure that fixes the valuation of assets for all trades in a given block of transactions. We construct an algorithm, which is both asymptotically efficient and empirically practical, that computes these valuations while exactly preserving a DEX's financial correctness constraints. Not only does this market structure provide fairness across trades, but it also makes trade operations commutative and hence efficiently parallelizable. SPEEDEX is prototyped but not yet merged within the Stellar blockchain, one of the largest Layer-1 blockchains.

1 Introduction

Digital currencies are moving closer to mainstream adoption. Examples include central bank digital currencies (CBDCs) such as China's DC/EP [90], commercial efforts [65, 75], and many decentralized-blockchain-based stablecoins such as Tether [104], Dai [9], and USDC [17]. These currencies vary wildly in terms of privacy, openness, smart contract support, performance, regulatory risk, solvency guarantees, compliance features, retail vs. wholesale suitability, and centralization of the underlying ledger. Because of these differences, and because financial stability demands different monetary policy in different countries, we cannot hope for a one-size-fits-all global digital currency. Instead, to realize the full potential of digital currencies (and digital assets in general), we need an ecosystem where many digital currencies can efficiently interoperate.

Effective interoperability requires an *exchange*: an efficient system for exchanging one digital asset for another. Users post offers to trade one asset for another on the exchange, and then the exchange matches mutually compatible offers together and transfers assets according to the offered terms. For example, one user might offer to trade 110 USD for 100 EUR, and might be matched against another user who previously offered to trade 100 EUR for 110 USD. A typical exchange maintains *orderbooks* of all of the open trade offers.

The ideal digital currency exchange should, at minimum,

- not give any central authority undue power over the global flow of money,
- operate transparently and auditably,
- give every user an equal level of access,
- enable efficient trading between every pair of currencies (make effective use of all available liquidity), and
- support arbitrarily high throughput, without charging significant fees to users.

Scalability is crucial for a piece of financial infrastructure that must last far into the future, as the number of individuals transacting internationally continues to grow. Furthermore, the above feature list is by no means complete; a deployment may want any number of additional features, such as persistent logging, simplified payment verification [89], or integrations with legacy systems, each of which slows down the system's performance. Scalability, viewed from another angle, enables the system to add features without decreasing overall transaction throughput (at the cost of additional compute hardware).

The gold standard for avoiding centralized control is a *decentralized exchange*, or DEX: a transparent exchange implemented as a deterministic replicated state machine maintained by many different parties. To prevent theft, a DEX requires all transactions to be digitally signed by the relevant asset holders. To prevent cheating, replicas organize history into an append-only blockchain. Replicas agree on blockchain state through a Byzantine-fault tolerant consensus protocol, typically some variant of asynchronous or eventually synchronous Byzantine agreement [46] for private blockchains

or synchronous mining [89] for public ones.

Unfortunately, existing DEX designs cannot meet the last three desiderata.

Equality of Access In existing exchange designs, users with low-latency connections to an exchange server (centralized or not) can spy on trades incoming from other users and *front-run* these trades. For example, a front-runner might spy an incoming sell offer, and in response, send a trade that buys and immediately resells an asset at a higher price [38,82]. In a blockchain, where a block of trades is either finalized entirely or not at all, this front-running can be made risk-free. More generally, some users form special connections with blockchain operators to gain preferential treatment for their transactions [55]. This special treatment typically takes the form of ordering transactions in a block in a favorable manner. The result is hundreds of millions of dollars siphoned away from users [95].

Effective Use of Liquidity Existing exchange designs are filled with arbitrage opportunities. A user trading from one currency A to another B might receive a better overall exchange rate by trading through an intermediate *reserve* currency C, such as USD. Users must typically choose a single (sequence of) intermediate asset(s), leaving behind arbitrage opportunities with other intermediate assets. This challenge is especially problematic in the blockchain space, where market liquidity is typically fragmented between multiple fiat-pegged tokens.

Computational Scalability DEX infrastructure must also be scalable. The ideal DEX needs to handle as many transactions per second as users around the globe want to send, without limiting transaction rates through high fees. Trading activity growth may outpace Moore's law, and should not be limited by the rate of increase of single-CPU-core performance. An ideal DEX should handle higher transaction rates simply by using more compute hardware.

Unfortunately, folk wisdom holds that DEXes cannot scale beyond a few thousand transactions per second. Naïve parallel execution would not be replicable across different blockchain nodes. This wisdom has led to many alternative blockchain scaling techniques, such as off-chain trade matching [108], automated market-makers [24], transaction rollup systems [15, 19], and sharded blockchains [6] or side-chains [92]. These approaches either trust a third party to ensure that orders are matched with the best available price, or sacrifice the ability to set traditional limit orders that only sell at or above a certain price (reducing market liquidity). Offchain rollup systems, sharded chains, and side-chains further fragment market liquidity, leading to cross-shard arbitrage and worse exchange rates for traders.

A challenge for on-chain limit-order DEXes is that the order of operations affects their results. Typically, a DEX matches each offer to the reciprocal offer with the best price: e.g., the first offer to buy 1 EUR might consume the only offer priced at 1.09 USD, leaving the second to pay 1.10 USD. Each trade is a read-modify-write operation on a shared orderbook data structure, so trades must be serialized. This serialization order must be deterministic in a replicated state machine, but naïve parallel execution would make the order of transactions dependent on non-deterministic thread scheduling.

1.1 SPEEDEX: Towards an Ideal DEX

This paper disproves the conventional wisdom about on-chain DEX performance. We present SPEEDEX, a fully on-chain decentralized exchange that meets all of the desiderata outlined above. SPEEDEX gives every user an equal level of access (thereby eliminating a widespread class of risk-free front-running), eliminates internal arbitrage opportunities (thereby making optimal use of liquidity available on the DEX), and is capable of processing over 200,000 transactions per second when deployed on 48-core machines (Figure 3). SPEEDEX is designed to scale further when given more hardware.

Like most blockchains, SPEEDEX processes transactions in blocks—in our case, a block of 500,000 transactions every few seconds. Its fundamental principle is that transactions in a block commute: a block's result is identical regardless of transaction ordering, which enables efficient parallelization [51].

SPEEDEX's core innovation is to execute every order at the same exchange rate as every other order in the same block. SPEEDEX processes a block of limit orders as one unified batch, in which, for example, every 1 EUR sold to buy USD receives exactly 1.10 USD in payment. Furthermore, SPEEDEX's exchange rates present no arbitrage opportunities within the exchange; that is, the exchange rate for trading USD to EUR directly is exactly the exchange rate for USD to YEN multiplied by the rate for YEN to EUR. These exchange rates are unique for any (nonempty) batch of trades. Users interact with SPEEDEX via traditional limit orders, and SPEEDEX executes a limit order if and only if the batch's exchange rate exceeds the order's limit price.

This design provides two additional economic advantages. First, the exchange offers liquid trading between every asset pair. Users can directly trade any asset for any other asset, and the market between these assets will be at least as liquid as the most liquid market path through intermediate reserve currencies. Second, SPEEDEX eliminates a class of front-running that is widespread in modern DEXes. No exchange operator or user with a low-latency network connection can buy an asset and resell it at a higher price, within the same block. (Note that this is not every type of front-running; §8 and §10 contrast SPEEDEX's guarantees with those of other mitigations, and how they can be combined.)

Furthermore, this economic design enables a scalable systems design that is not possible using traditional order-matching semantics. Unlike every other DEX, the operation of SPEEDEX is efficiently parallelized, allowing SPEEDEX to scale to transaction rates far beyond those seen today. Transactions within a block commute with each other precisely because trades all happen at the same shared set of exchange rates. This means that the transaction processing engine has no

need for the sequential read-modify-update loop of traditional orderbook matching engines. Account balances are adjusted using only hardware-level atomics, rather than locking.

1.2 SPEEDEX Overview

SPEEDEX is not a blockchain itself; rather, it is a DEX component that can be integrated into any blockchain. A copy of the SPEEDEX module should run inside every replica of a blockchain using the system. SPEEDEX does not depend on any specific property of a consensus protocol, but automatically benefits from throughput advances in consensus and transaction dissemination (such as [56]). SPEEDEX heavily uses concurrency and benefits from uninterrupted access to CPU caches, and as such is best implemented directly within blockchain node software (instead of as a smart contract).

We implemented SPEEDEX within a custom blockchain using the HotStuff consensus protocol [115]; this implementation provides the measurements in this paper. We created a second implementation as a component of the Stellar blockchain [84], which is considering a Layer-1 SPEEDEX deployment.

Implementing SPEEDEX introduces both theoretical algorithmic challenges and systems design challenges. The core algorithmic challenge is the computation of the batch prices. This problem maps to a well-studied problem in the theoretical literature (equilibrium computation of *Arrow-Debreu Exchange Markets*, §A.1); however, the algorithms in the theoretical literature scale extremely poorly, both asymptotically and empirically, as the number of open limit orders increases.

We show that the market instances which arise in SPEEDEX have additional structure not discussed in the theoretical literature, and use this structure to build a novel algorithm (based on the Tâtonnement process of [53]) that, in practice, efficiently approximates batch clearing prices. We then explicitly correct approximation error with a follow-up linear program.

Our algorithm's runtime is largely independent of the number of limit orders—each Tâtonnement query has a runtime of $O(\#\text{assets}^2 \cdot \lg(\#\text{offers}))$ and the linear program has size $O(\#\text{assets}^2)$. This gives a crucial algorithmic speedup because in the real world, the number of currencies is much smaller than the number of market participants. (The experiments of §6 and §7 use 50 assets and tens of millions of open offers.)

On the systems design side, to implement this exchange, we design natural commutative transaction semantics and implement data structures designed for concurrent, batched manipulation and for efficiently answering queries about the exchange state from the price computation algorithm.

In recent years, the economics literature has begun discussing the use of batched trading systems in traditional markets to combat front-running and externalities associated with high-frequency trading [30, 41, 43]. This literature focuses only on the case of trading between two assets (where price computation is simple) or where all trades use a single *numeraire* currency [44]. Our contribution to this line of work is to demonstrate the feasibility of a batch trading system that exchanges many assets and many numeraire currencies simultaneously, thereby expanding the design space of implementable market structures.

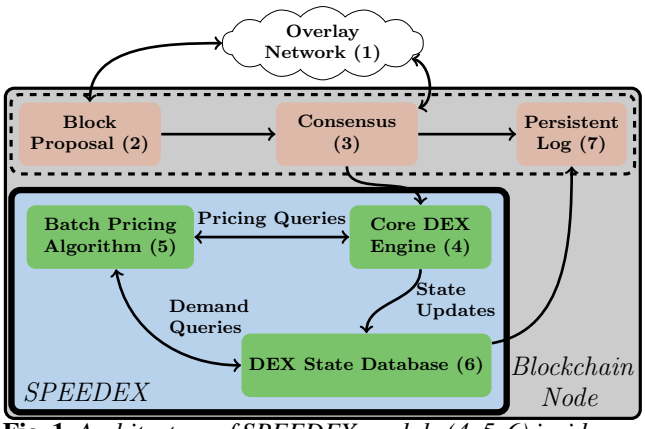


Fig. 1. *Architecture of SPEEDEX module (4, 5, 6) inside one blockchain node.*

2 System Architecture

SPEEDEX is an asset exchange implemented as a replicated state machine in a blockchain architecture (Fig. 1). Assets are issued and traded by *accounts*. Accounts have public signature keys authorized to spend their assets. Signed transactions are multicast on an overlay network (Fig. 1, 1) among block *producers*. At each round, one or more producers propose candidate blocks extending the blockchain history (Fig. 1, 2). A set of *validator* nodes (generally the same set or a superset of the producers) validates and selects one of the blocks through a consensus mechanism (Fig. 1, 3). SPEEDEX is suitable for integration into a variety of blockchains, but benefits from a consensus layer with relatively low latency (on the order of seconds), such as BA⋆ [71], SCP [84], or HotStuff [115].

The implementation evaluated here uses HotStuff [115], while the the Stellar blockchain implementation relies on Stellar's existing consensus protocol, SCP [88].

Most central banks and digital currency issuers maintain a ledger tracking their currency holdings. SPEEDEX is not intended to replace these primary ledgers. Rather, we expect banks and other regulated financial institutions to issue 1:1 backed token deposits onto a blockchain that runs SPEEDEX and provide interfaces for moving money on and off the exchange. These assets could be digital-native tokens as well; any divisible and fungible asset can integrate with SPEEDEX.

SPEEDEX supports four operations: account creation, offer creation, offer cancellation, and send payment. Offers on SPEEDEX are traditional limit orders. For example, one offer might offer to sell 100 EUR to buy USD, at a price no lower than 0.91 USD/EUR. Offers can trade between any pair of assets, in either direction. Another offer, for example, might offer to sell 100 USD in exchange for EUR, at a price no lower than 1.10 EUR/USD.

What makes SPEEDEX different from existing DEXes is the manner in which it processes new orders. Traditional

exchanges process trades sequentially, implicitly computing a matching between limit orders. SPEEDEX, by contrast, processes trades in batches (typically, one batch would consist of all of the limit orders in one block of the blockchain).

In a blockchain, all of the transactions in a block are appended at the same clock time, so there is no reason *a priori* why a DEX should pick one ordering over another. SPEEDEX, by design, imposes no ordering whatsoever between transactions in a block. Side effects of a transaction are only visible to other transactions in future blocks.

Logically, when the SPEEDEX core engine (Fig. 1, 4) receives a finalized block of trades, it applies all of the trades at exactly the same time and computes an unordered set of state changes, which it passes to its exchange state database (Fig. 1, 6). This database records orderbooks and account balances, and is periodically written to the persistent log (Fig 1, 7).

2.1 SPEEDEX Module Architecture

To implement an exchange that operates replicably where trades in a block are not ordered relative to each other, SPEEDEX requires a set of trading semantics such that operations *commute*.

Traditional exchange semantics are far from commutative: one offer to buy an asset is matched with the lowest priced seller, and the next offer to buy is matched against the second-lowest priced seller, and so on. Hence, every trade can occur at a slightly different exchange rate.

Instead, to make trades commutative, SPEEDEX computes in every block a *valuation* $p_{\mathcal{A}}$ for every asset $\mathcal{A}$. The units of $p_{\mathcal{A}}$ are meaningless, and can be thought of as a fictional valuation asset that exists only for the duration of a single block. However, valuations imply exchange rates between different assets—every sale of asset $\mathcal{A}$ for asset $\mathcal{B}$ occurs at a price of $p_{\mathcal{A}}/p_{\mathcal{B}}$. Unlike traditional exchanges, SPEEDEX does not explicitly compute a matching between trade offers. Instead, offers trade with a conceptual *"auctioneer"* entity at these exchange rates. Trading becomes commutative because all trades in one asset pair occur at the same price.

The main algorithmic challenge is to compute valuations where the exchange *clears*—i.e., the amount of each asset sold to the auctioneer equals the amount bought from the auctioneer.

When the auctioneer sets exact clearing valuations, an offer trades fully with the auctioneer if its limit price is strictly below the auctioneer's exchange rate, and not at all if its limit price exceeds the auctioneers rate. When the limit price equals the exchange rate, SPEEDEX may execute the offer partially. Note that an exchange is a zero-sum system; as compared to sequential execution, some users may see better prices and some may see worse, but SPEEDEX guarantees that no user's price is worse than their minimum limit price.

Theorem 1. *Exact clearing valuations always exist. These valuations are unique up to rescaling.*[1]

[1] And technical conditions (§A.3), e.g. everything clears an empty market.

Theorem 1 is a restatement of a general theorem of Arrow-Debreu exchange market theory [57] (§A.3).

Concretely, whenever the core SPEEDEX engine (Fig 1, 4) receives a newly finalized block, one of its first actions is to query an algorithm that computes clearing valuations (Fig 1, 5). It then uses the output of this algorithm to compute the modifications to the exchange state (Fig 1, 6).

As valuations that clear the market always exist for any set of limit orders, there is no adversarial input that SPEEDEX cannot process. And because these valuations are unique, SPEEDEX operators do not have a strategic choice between different sets of valuations. SPEEDEX's algorithmic task is to surface information about a fundamental mathematical property of a batch.

Unfortunately, we are not aware of a practical method to compute clearing prices exactly. (The number of bits required to represent exact clearing prices may be extremely large [57], and in a natural extension of the SPEEDEX model [96] the clearing prices are not even rational.) SPEEDEX therefore uses *approximate* clearing prices.

At nonexact clearing prices, the conceptual auctioneer will not have enough of some asset(s) to pay out all offers willing to accept the market price. SPEEDEX addresses this deficit in two ways. First, the auctioneer proportionally reduces the amount it pays out to offers by a small fraction—in other words, it charges a commission. Commissions are common for exchanges, whether decentralized or not, though SPEEDEX does it for market clearing rather than profit reasons. To avoid incentivizing high trading costs, the implementation returns commissions to the asset issuers, and one goal of our price computation algorithm's design is to make this commission as low as computationally practical. Second, the auctioneer can refrain from filling some marketable offers. Whereas in a perfect Arrow-Debreu exchange market, offers at the market price may be partially filled or not filled, in SPEEDEX the same applies to offers very close to the market price, even if they still beat the market price by a small percentage.

SPEEDEX always rounds trades in favor of the auctioneer. Our implementation burns collected transaction fees and accumulated rounding error (effectively returning them to the issuer by reducing the issuer's liabilities). The Stellar implementation eliminates the fee and returns the accumulated rounding error to asset issuers.

2.2 Design Properties

Computational Scalability SPEEDEX's commutative semantics allow effective parallelization of DEX operation. Because transactions within a block are not semantically ordered, DEX state is identical regardless of the order in which transactions are applied. This exact replicability is, of course, required for a *replicated* state machine. The order-independence also means SPEEDEX transactions can be executed in parallel by all available CPU cores despite the fact that thread interleaving is nondeterministic in multicore machines. Almost all coordination occurs via hardware-level

atomics (e.g., atomic add on 64-bit integers) without spinlocks.

SPEEDEX stores balances in accounts, rather than in discrete, unspent coins (often called "UTXOs"). It also supports single-currency payment operations, which are simpler than DEX trading. Hence, SPEEDEX disproves the popular belief [85, 97] that account-based ledgers are not compatible with horizontal scalability.

No risk-free front running Well-placed agents in real-world financial markets can spy on submitted offers, notice a new transaction T, and then submit a transaction T' (that executes before T) that buys an asset and re-sells it to T at a slightly higher price. In some blockchain settings, T' can be done as a single atomic action [55]. However, since every transaction sees the same clearing prices in SPEEDEX, back-to-back buy and sell offers would simply cancel each other out. Relatedly, because every offer sees the same prices, a user who wishes to trade immediately can set a very low minimum price and be all but guaranteed to have their trade executed, but still at the current market price.

Risk-free front-running is one instance of the widely discussed "Miner Extractable Value" (MEV) [55] phenomenon, in which block producers reorder transactions within a block for their own profit (or in exchange for kickbacks). By eliminating the ordering of transactions within a block, SPEEDEX eliminates a large source of MEV. However, this does not eliminate every type of front-running manipulation, such as delaying victim transactions to a future block (see §8).

No (internal) arbitrage and no central reserve currency An agent selling asset $\mathcal{A}$ in exchange for asset $\mathcal{B}$ will see a price of $p_\mathcal{A}/p_\mathcal{B}$. An agent trading $\mathcal{A}$ for $\mathcal{B}$ via some intermediary asset $\mathcal{C}$ will see exactly the same price, as $\frac{p_\mathcal{A}}{p_\mathcal{C}} \cdot \frac{p_\mathcal{C}}{p_\mathcal{B}} = \frac{p_\mathcal{A}}{p_\mathcal{B}}$. Hence, one can efficiently trade between assets without much pairwise liquidity with no need to search for an optimal path. By contrast, many international payments today go through USD because of a lack of pairwise liquidity. The multitude of USD-pegged stablecoins in modern blockchains further fragments liquidity. Of course, there can still be arbitrage between SPEEDEX and external markets.

3 Commutative DEX Semantics

To propose or execute a block of transactions, the SPEEDEX core engine performs the following three actions.

1 For each transaction in the block (in parallel), check signature validity, collect new limit offers, and compute available account balances after funds are committed to offers or transferred between accounts. When proposing a block of transactions, SPEEDEX discards potentially invalid transations.

2 When proposing a block, compute approximate clearing prices and approximation correction metadata.

3 Iterate over each offer, making a trade or adding it to the resting orderbooks (based on the prices and metadata).

For transaction processing in step 1 to be commutative, it must be the case that the step 1 output effects (specifically: create a new account, create a new offer, cancel an existing offer, and send a payment) of one transaction have no influence on the output effects of another transaction. This means that one transaction cannot read some value that was output by another transaction (in the same block), and that whether one transaction succeeds cannot depend on the success of another transaction.

To meet the first requirement, traders include all parameters to their transactions within the transaction itself. The second requirement necessitates precise management of transaction side effects. At most one transaction per block may alter an account's metadata (such as the account's public key or existence), and metadata changes take effect only at the end of block execution. Similarly, an offer cannot be created and cancelled in the same block. As payments and trading are the common case, we do not consider these restrictions a serious limitation.

SPEEDEX must also ensure that no account is overdrafted. That is to say, after processing all transactions in a block, the unlocked balance of every account must be nonnegative (where an open offer locks the offered amount of an asset for the duration of its lifetime). Unlike most distributed ledgers, SPEEDEX cannot simply deem the second of two conflicting transactions to fail—after all, transactions have no ordering. Instead, our implementation requires a block proposer to ensure that a block cannot cause overdrafts; every node rejects blocks that violate this property. To generate valid blocks, proposers use a conservative process outlined in §K.6. The design requires passing information from the SPEEDEX database (Fig 1, 6) to the proposal module (Fig 1, 2).

The core remaining technical challenge is the batch price computation (Fig 1, 5).

4 Price Computation

4.1 Requirements

As discussed earlier, in every block, SPEEDEX computes batch clearing prices and executes trades in response to these prices. Every DEX is subject to two fundamental constraints:

- **Asset Conservation** No assets should be created out of nothing. As discussed in §2, offers in SPEEDEX trade with a virtual auctioneer. After a batch of trades, this auctioneer cannot be left with any debt. We do allow the auctioneer to burn some surplus assets as a fee.
- **Respect Offer Parameters** No offer trades at a worse price than its limit price.

Additionally, SPEEDEX should facilitate as much trade volume as possible. (Otherwise, the constraints could be vacuously met by never trading.) Furthermore, price computation must be efficient, as it occurs for each block of trades, every few seconds. Finally, SPEEDEX should minimize the number of offers that trade partially; asset quantities are stored as integer multiples of a minimum unit, so each partial trade risks accumulating a rounding error of up to one unit.

4.2 From Theory To Practice

The problem of computing batch clearing prices is equivalent to the problem of computing equilibria in linear Arrow-Debreu Exchange Markets (§A). Our algorithm is based on the iterative Tâtonnement process from this literature [53].

However, the runtimes of the theoretical algorithms scale very poorly, both asymptotically and empirically. They also output approximate equilibria for notions of approximation that violate the two fundamental constraints above (for example, Definition 1 of [53] permits equilibria to mint new assets and to steal from a user).

We develop a novel algorithm for computing equilibria that runs efficiently in practice (§6) and explicitly ensures that (1) asset amounts are conserved and (2) every offer trades at exactly the market prices, and only if the offer's limit price is at or below the batch exchange rate. First, Tâtonnement approximates clearing prices (§5). We show that the structure of the types of trades in SPEEDEX lets each iteration run in time logarithmic in the number of open limit offers (via a series of binary searches), giving an algorithm asymptotically faster than that within the theoretical literature.

We then explicitly correct for the approximation error with a linear program (§D). Crucially, the size of this linear program is linear in the number of asset pairs, and has no dependence on the number of open trade offers. The linear program ensures that, no matter what prices Tâtonnement outputs, (1) asset amounts are conserved, and (2) no offer trades if the batch price is less than its limit price.

To be precise, our algorithm outputs the following:

- **Prices:** For each asset $\mathcal{A}$, SPEEDEX computes an asset valuation $p_{\mathcal{A}}$. One unit of $\mathcal{A}$ trades for $p_{\mathcal{A}}/p_{\mathcal{B}}$ units of $\mathcal{B}$.
- **Trade Amounts:** For each asset pair $(\mathcal{A},\mathcal{B})$, SPEEDEX computes an amount $x_{\mathcal{A},\mathcal{B}}$ of asset $\mathcal{A}$ that is sold for asset $\mathcal{B}$ (again, at exchange rate $p_{\mathcal{A}}/p_{\mathcal{B}}$).

For every asset pair $(\mathcal{A},\mathcal{B})$, SPEEDEX sorts all of the offers selling $\mathcal{A}$ for $\mathcal{B}$ by their limit prices, and then executes the offers with the lowest limit prices, until it reaches a total amount of $\mathcal{A}$ sold of $x_{\mathcal{A},\mathcal{B}}$ (tiebreaking by account ID and offer ID).

As a bonus, this method ensures that at most one offer per trading pair executes partially, minimizing rounding error.

5 Price Computation: Tâtonnement

Tâtonnement is an iterative process; starting from an (arbitrary) initial set of prices, it iteratively refines them until the prices reach a stopping criterion.

Each iteration of Tâtonnement starts with a *demand query*. The *demand* of an offer is the net trading of the offer (with the auctioneer) in response to a set of prices, and the demand of a set of offers is the sum of the demands of each offer. Tâtonnement's goal is to find prices such that the amount of each asset sold to the auctioneer matches the amount bought from it (in other words, the net demand is 0).

Example 1. *Suppose that a limit order offers to sell 100 USD for EUR with a minimum price of* 0.8 *EUR per USD. If the candidate prices are such that* $\alpha = \frac{p_{USD}}{p_{EUR}} > 0.8$*, then the limit order would like to trade, and its demand is* $(-100\ USD, 100\alpha\ EUR)$*. Otherwise, its demand is* $(0\ USD, 0\ EUR)$*.*

Iterative Price Adjustment. If more units of an asset are demanded from the auctioneer than are supplied to it (a positive net demand, meaning a deficit for the auctioneer), then the auctioneer raises the price of the asset. Otherwise, the auctioneer has a surplus, so it lowers the price of the asset. Implementing this process effectively requires careful numerical normalization in response to differences in prices and trade volumes, which we describe in detail in §C.1.

Tâtonnement repeats this process until the current set of prices is sufficiently close to the market clearing prices (or it hits a timeout). Specifically, Tâtonnement iterates until it has a set of prices such that, if the auctioneer charges a commission of ε, then there is a way to execute offers such that:

1. The auctioneer has no deficits (assets are conserved)
2. No offer executes outside its limit price bound
3. Every offer with a limit price more than a $(1-\mu)$ factor below the auctioneer's exchange rate executes in full.

The last condition is a formalization of the notion that SPEEDEX should satisfy as many trade requests as possible. Informally, an offer with a limit price equal to the auctioneer's exchange rate is indifferent between trading and not trading, while one with a limit price far below the auctioneer's exchange rate strongly prefers trading to not trading.

5.1 Efficient Demand Queries

Implemented naïvely, Tâtonnement's demand queries would consist of a loop over every open exchange offer. This is impossibly expensive, even if the loop is massively parallelized. Concretely, one invocation of Tâtonnement can require many thousands of demand queries. Every demand query therefore must return results in at most a few hundred microseconds.

This naïve loop appears to be required for the (more general) problem instances studied in the theoretical literature. However, all of the offers in SPEEDEX are traditional limit orders that sell one asset in exchange for one other asset at some limit price. An offer with a lower limit price always trades if an offer with a higher limit price trades. Therefore, SPEEDEX groups offers by asset pair and sorts offers by their limit prices. We drive the marginal cost of this sorting to near zero by using an offer's limit price as the leading bits (in big-endian) of the keys in our Merkle tries (§K.5).

SPEEDEX can therefore compute a demand query with a sequence of binary searches (§G). Individual binary searches can run on separate CPU cores. The number of open offers (say, M) on an exchange is vastly higher than the number of assets traded (say, N). Our experiments in §7 trade $N = 50$ assets with $M =$ tens of millions of open offers; the complexity reduction from $O(M)$ to $O(N^2 \lg(M))$ is crucial.

5.2 Multiple Tâtonnement Instances

§C describes several other Tâtonnement adjustments that help it respond well to a wide variety of market conditions. Some of these adjustments are parametrized (such as how quickly one should adjust the candidate prices); rather than pick one set of control parameters, we run several instances of Tâtonnement in parallel and take whichever finishes first as the result. (In the case of a timeout, we choose the set of prices that minimizes the *unrealized utility* [§6.2].) SPEEDEX includes the output of Tâtonnement and the subsequent linear program in the headers of proposed blocks (§K.3).

6 Evaluation: Price Computation

Tâtonnement's runtime depends primarily on the target approximation accuracy, the number of open trade offers, and the distribution of the open trade offers. The runtime increases as the desired accuracy increases. Surprisingly, the runtime actually *decreases* as the number of open offers increases. And like many optimization problems, Tâtonnement performs best when the input is normalized, meaning in this case that the (normalized, §C.1) volume traded of each asset is roughly the same.

Tâtonnement runs once per block. To produce a block every few seconds, Tâtonnement must run in under one second most of the time. Our implementation runs Tâtonnement with a timeout of 2 seconds, but it typically converges much faster.

6.1 Accuracy and Orderbook Size

We find that Tâtonnement converges more quickly as the number of open offers *increases*. Tâtonnement converges fastest when small price changes do not cause comparatively large changes in overall net demand. However, an offer's behavior is a discontinuous function (of prices); it does not trade below its limit price and trades fully above it.

There are two factors that mitigate these "jump discontinuities." First, Tâtonnement approximates optimal offer behavior by a continuous function (§B). Smaller μ means a closer approximation. Second, the more offers there are in a batch, the smaller any one offer's relative contribution to overall demand. This last factor explains why Tâtonnement converges more quickly when there are more offers on the exchange. A real-world deployment might raise accuracy as trading increases.

Fig. 2 plots the minimum number of trade offers that Tâtonnement needs to consistently find clearing prices for 50 distinct assets in under 0.25 seconds (for the same trade distribution used in §7). To put these fee rates in context, BinanceDex [1] charged a fee of either $0.1\% \approx 2^{-10}$ or $0.04\% \approx 2^{-11.3}$. Uniswap [24, 25] charges 1%, 0.3%, or 0.05% ($\sim 2^{-6.6}$, $\sim 2^{-8.4}$, and $\sim 2^{-11}$, respectively), and Coinbase charges 0.5% to 4% [4] ($\sim 2^{-7.6}$ to $\sim 2^{-4.6}$).

Though our experiments rarely experienced Tâtonnement timeouts, Tâtonnement timeouts caused by sparse orderbooks may be self-correcting: If SPEEDEX proposes suboptimal prices, fewer offers will find a counterparty and trade. When fewer offers clear in one block, more are left to facilitate

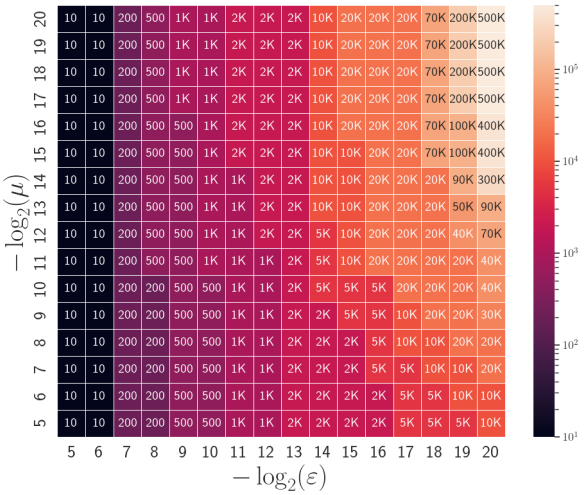


Fig. 2. *Minimum number of offers needed for Tâtonnement to run in under* 0.25 *seconds (Smaller is better. Times averaged over 5 runs). The x axis denotes offer behavior approximation quality (μ), and the y axis denotes the commission (ε).*

Tâtonnement in the next block. §F describes an alternative algorithm that is effective on small batches.

6.2 Robustness Checks

As a robustness check, we run Tâtonnement against a trade distribution derived from volatile cryptocurrency market data. In an ideal world, we could replay trades from another DEX through SPEEDEX. Unfortunately, doing so poses several problems. First, in practice, almost all DEX trades go through four de facto reserve currencies (ETH, USD, USDC, and USDT), three of which are always worth close to $1. The decomposition between a few core "pricing" assets and a larger number of other assets makes price discovery too simple. Second, transaction rates on existing DEXes are too low to provide enough data. Finally, we suspect users would submit different orders to SPEEDEX than they might on a traditional exchange, due to the distinct economic properties of batch trading systems.

Experiment Setup. As a next-best alternative, we generate a dataset based on historical price and market volume data. We took the 50 crypto assets that had the largest market volume on December 8, 2021 (as reported by coingecko.com) and for each asset, gathered 500 days of price and trade volume history. We then generated 500 batches of 50,000 transactions. A new offer in batch i sells asset $\mathcal{A}$ (and buys asset $\mathcal{B}$) with probability proportional to the relative volume of asset $\mathcal{A}$ (and asset $\mathcal{B}$, conditioned on $\mathcal{A} \neq \mathcal{B}$) on day i, and demands a minimum price close to the real-world exchange rate on day i. The extreme volatility of cryptocurrency markets and variation between these 50 assets make this dataset particularly difficult for Tâtonnement. To further challenge Tâtonnement, we use a smaller block size of $\sim$30,000 (compared to 500,000 in §7).

The experiment charged a commission of $\varepsilon = 2^{-15} \approx$

0.003%, and attempted to clear offers with limit prices more than $1-\mu$ below the market prices, for $\mu = 2^{-10} \approx 0.1\%$ (§B).

Experiment Results. The experiment ran for 500 blocks. Each block created about 25,000 new offers and a few thousand cancellations and payments.

Tâtonnement computed an equilibrium quickly in 350 blocks, and in the remainder, computed prices sufficiently close to equilibrium that the follow-up linear program facilitated the vast majority of possible trading activity.

We measure the quality of an approximate set of prices by the ratio of the "unrealized utility" to the "realized utility." The utility gained by a trader from selling one unit of an asset is the difference between the market exchange rate and the trader's limit price, weighted by the valuation of the asset being sold. Note that the units do not matter when comparing relative amounts of "utility."

In the blocks where Tâtonnement computed an equilibrium quickly, the mean ratio of unrealized to realized utility was 0.71% (max: 4.7%), and in the other blocks, the mean ratio was 0.42% (max: 3.8%).

Recall that Tâtonnement terminates as soon as a stopping criteria is met; roughly, "does the supply of every asset exceed demand," so one mispriced asset will cause Tâtonnement to keep running. However, every Tâtonnement iteration continues to refine the price of every asset. This is why Tâtonnement actually gives more accurate results in the batches it found challenging. A deployment might enforce a minimum number of Tâtonnement rounds.

Qualitatively, Tâtonnement correctly prices assets with high trading volume and struggles on sparsely traded assets (as might be expected from Fig. 2). Tâtonnement also adjusts its price adjustment rule in response to recent market conditions (§C.1), a tactic which is less effective on volatile assets.

Should this pose a problem in practice, a deployment could choose to vary the approximation parameters by trading pair.

7 Evaluation: Scalability

We ran SPEEDEX on four r6id.24xlarge instances in an Amazon Web Services datacenter. Each instance has 48 physical CPU cores divided over two Intel Xeon Platinum 8375CL chips (32 total cores per socket, 24 of which are allocated to our instances), running at 2.90Ghz with hyperthreading enabled, 768GB of memory, 4 1425GB NVMe drives connected in a RAID0 configuration. We use the XFS filesystem [102]. These experiments use the HotStuff consensus protocol [115], and do not include Byzantine replicas or a rotating leader.

Experiment Setup. These experiments simulate trading of 50 assets. Transactions are charged a fee of $\varepsilon = 2^{-15}(0.003\%)$. We set $\mu = 2^{-10}$, guaranteeing full execution of all orders priced below 0.999 times the auctioneer's price. The initial database contains 10 million accounts. Tâtonnement never timed out, and typically required fewer than 1,000 iterations.

Transactions are generated according to a synthetic data model—every set of 100,000 transactions is generated as though the assets have some underlying valuations, and users trade a random asset pair using a minimum price close to the underlying valuation ratio. The valuations are modified (via a geometric Brownian motion) after every set. Accounts are drawn from a power-law distribution.

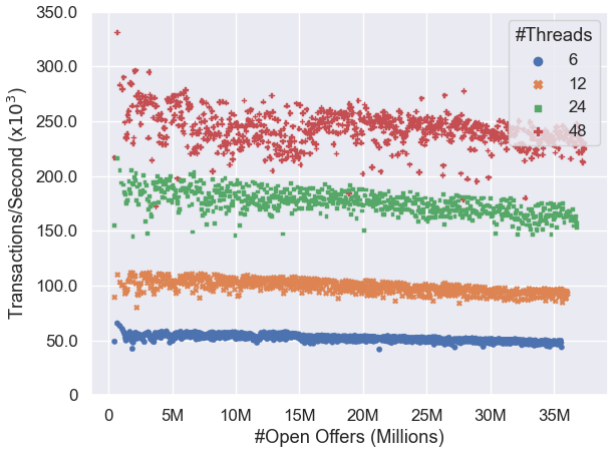


Fig. 3. *Transactions per second on SPEEDEX, plotted over the number of open offers.*

Each set is split into four pieces, with one piece given to each replica. Replicas load these sets sequentially and broadcast each set to every other replica. Each replica adds received transactions to its pool of unconfirmed transactions.

Replicas propose blocks of roughly 500,000 transactions. In these experiments, each block consists of roughly 350,000–400,000 new offers, 100,000–150,000 cancellations, 10,000–20,000 payments, and a small number of new accounts. We generate 5,000 sets of input transactions. Some of these transactions conflict with each other and are discarded by SPEEDEX replicas. Each experiment runs for 700–750 blocks.

Every five blocks, the exchange commits its state to persistent storage in the background (via LMDB [50], §K.2).

Performance Measurements. Fig. 3 plots the end-to-end transaction throughput rate of SPEEDEX as the number of worker threads inside SPEEDEX increases. The x-axis plots the number of open offers on the exchange.

Most importantly, Fig. 3 demonstrates that SPEEDEX can efficiently use its available CPU hardware. The speedup is near-linear, until the number of threads approaches the number of CPU cores—from 6 to 12, $\sim 1.9x$, from 12 to 24, $\sim 1.8x$, and from 24 to 48, $\sim 1.4x$. The thread counts are only for the number of threads directly for SPEEDEX's critical path, and not for many of the tasks that the implementation must perform in the background, such as logging data to persistent storage (logging the account database uses 16 threads), consensus, and garbage collection, and these threads begin to contend with SPEEDEX as the number of SPEEDEX worker threads increases.

Secondly, Fig. 3 demonstrates the scalability of SPEEDEX with respect to the number of open offers. The number of open

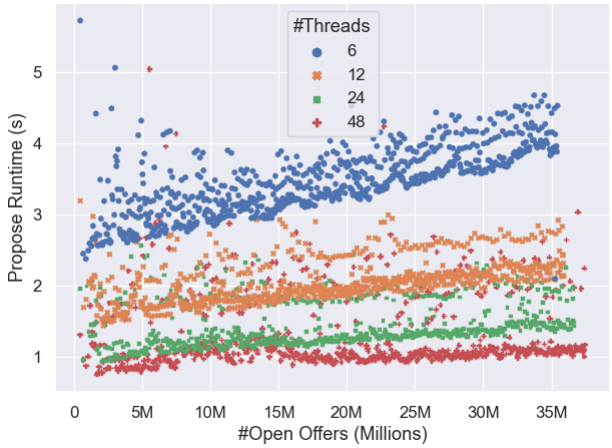


Fig. 4. *Time to propose and execute a block, plotted over the number of open offers.*

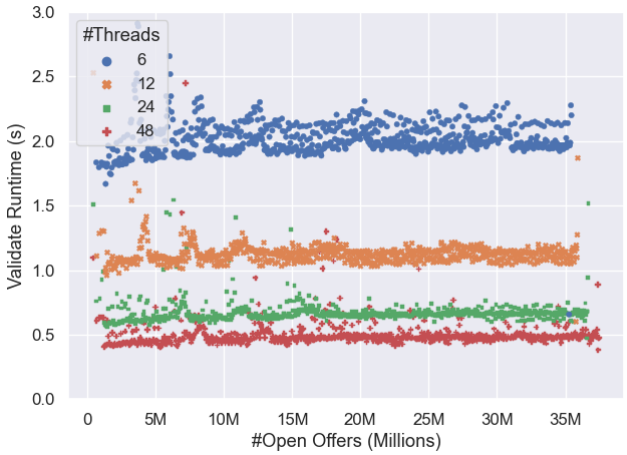


Fig. 5. *Time to validate and execute a proposal, plotted over the number of open offers (measurements from one replica).*

offers SPEEDEX works with in these experiments is already quite large, but most importantly, as the number of open offers goes from 0 to the 10s of millions, SPEEDEX's transaction throughput falls by only $\sim 10\%$. This slowdown is primarily derived from a Tâtonnement optimization (the precomputation outlined in §9.2). Tâtonnement is the one part of SPEEDEX that cannot be arbitrarily parallelized, so we design our implementation towards making it as fast as possible. An implementation might skip this work in some parameter regimes.

To focus on the performance of SPEEDEX, Figs. 4 and 5 plot the time to propose and execute blocks, and to validate and execute proposals, respectively, when we disable signature verification (which is trivial to parallelize). First, note that both proposal and validation scale with the number of threads; validation scales better than proposal due to the aforementioned Tâtonnement optimization. Second, note that validating and executing a proposal from another replica is substantially faster than proposing a block; this lets a replica that is somehow delayed catch up.

The runtime variation in Fig. 4 results from the fact that SPEEDEX without signature verification runs too quickly for our persistent logging implementation.

SPEEDEX is not a consensus protocol, and these experiments (one consensus invocation every few seconds) do not come close to stressing the consensus throughput of Hotstuff. However, network bandwidth requirements necessarily scale (at least) linearly with transaction rate. Recent work, such as [56, 79, 113], develops consensus protocols that maximally use available network bandwidth. However, integrating SPEEDEX with any consensus protocol requires understanding the tradeoffs between batch size, transaction rate, and consensus frequency. Fig. 6 plots this tradeoff running SPEEDEX on the same transaction workload as in Fig. 3. We also ran SPEEDEX with more replicas on different hardware and observed the same scalability trends, as outlined in §L (albeit with lower overall throughput on weaker hardware).

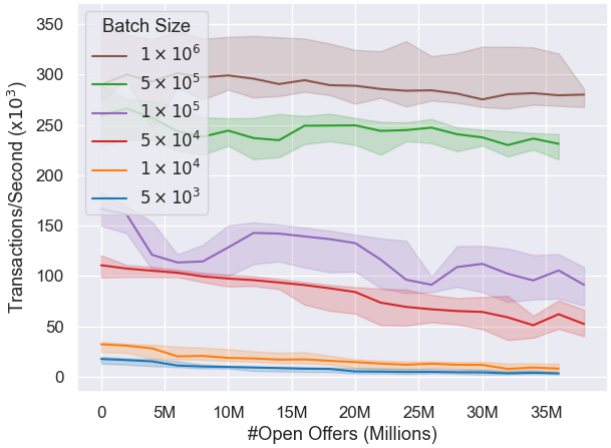


Fig. 6. *Median transaction rates, varying block size and number of open offers (grouped into buckets of 2M). Shaded areas plot 10th to 90th percentiles.*

Conclusions. To reiterate, SPEEDEX achieves these transaction rates while operating fully on-chain, with no offchain rollups and no sharding of the exchange's state. To make SPEEDEX faster, one can simply give it more CPU cores, without changing the transaction semantics or user interface. This scaling property is unique among existing DEXes.

7.1 Alternative Scaling Techniques

Traditional Exchange Semantics. The core logic of just an exchange system can be implemented extremely efficiently with almost no code. The logic of the constant product market maker UniswapV2 [24], for example, is less than 10 lines of simple arithmetic code. An orderbook-based exchange requires more code but can still be made very fast, as most operations modify only a small number of data objects. We implemented a bare-bones orderbook exchange with two assets using the same data structures as in SPEEDEX—each transaction checks the orderbook for a matching offer or offers and either makes appropriate transfers or adds the new offer to the

orderbook. These operations are extremely fast when the number of accounts is small; our implementation runs ∼1.7 million of these transactions per second when there are only 100 accounts. However, every database lookup becomes slower as the as the number of accounts grows; when there are 10 million accounts in the database (as in the above experiments), throughput falls 8x to ∼210,000 per second. Yet that is before adding all of the other SPEEDEX features one needs in a real DEX, such as state hashes, transaction fees, structures for simple payment verification [89], replication, or durable logging. The scalability of the full SPEEDEX implementation lets it surpass that rate even when slowed down by all of these features.

Note that every orderbook operation affects every subsequent transaction—each transaction influences the exchange rate observed in the next transaction—and as such, their execution cannot be parallelized. SPEEDEX's design, therefore, enables parallel execution of what would otherwise be a strictly serial workload. To isolate the effect of SPEEDEX's parallelizable semantics on its transaction throughput, we therefore turn to a workload that does not touch the DEX at all—one where every transaction is a payment between random accounts.

Optimistic Concurrency Control. A widely explored class of alternative designs for parallel transaction execution use optimistic concurrency control, and of these approaches the most closely related state of the art design appears to be Block-STM [70], which is deployed in Aptos [22]. This approach optimistically executes batches of transactions, retrying after conflicts as necessary.

We therefore design the measurements of Fig. 7 to mirror the experiments in [70]. The "Aptos p2p" transactions in [70] are payments between two random accounts, and consist of 8 reads of 5 writes. Each of our payments consists of two data reads (source account public key and last committed sequence number), two atomic compare_exchange operations (subtract payment and fee from source), an atomic fetch_xor (reserve sequence number), and an atomic fetch_add (add payment to destination)—implemented without atomics, this would be 6 reads and 4 writes. All payments are of the same asset.

Fig. 7 plots the throughput rates of SPEEDEX on this transaction workload for the parameter settings measured in Block-STM (Figs. 7 and 8, [70]). Note that for large batch sizes, the transaction throughput is largely independent of the number of accounts, even though every transaction in the two account setting contends with every other transaction. Furthermore, unlike Block-STM, SPEEDEX achieves near-linear scalability on sufficiently large batches. For small batch sizes, a large number of accounts actually slows down SPEEDEX, largely due to increased sensitivity to cache performance and our system's NUMA (two socket) architecture on small timescales. We also ran this experiment on a single-socket system (an AWS c5a.16xlarge, as in [70]), and found only negligle impact of the number of accounts on throughput. Fig. 7 was run with hyperthreading disabled, to compare against Block-STM experiments. The rest of our experiments were run with hyperthreading enabled (because of the many background tasks in SPEEDEX); enabling hyperthreading on this payments workload causes a negligible performance degradation for large batches (approximately 1-6%), and a larger (up to 25%) on small batches. As a baseline, §J graphs the performance of Block-STM on these parameter settings on our hardware.

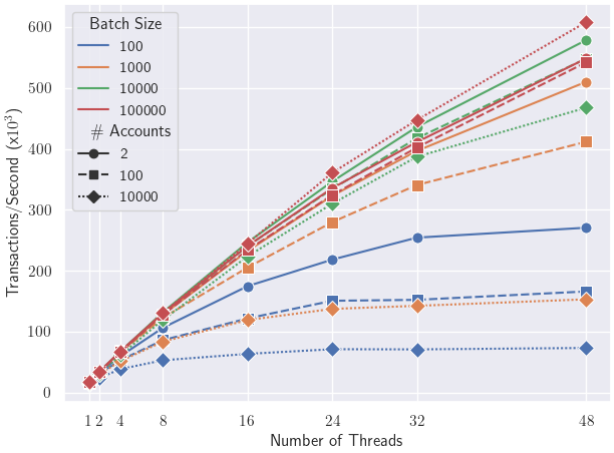


Fig. 7. *Throughput of SPEEDEX on batches of payment transactions with varying thread counts (average of 100 trials).*

We also ran SPEEDEX on an only-payments workload with 10 million accounts and 50 assets, and measured a throughput of approximately 375k, 215k, 114k, and 60k transactions per second using 48, 24, 12, and 6 threads, respectively (a 34.8x, 20.0x, and 10.6x, and 5.6x speedup over the single-threaded measurement). We disabled data persistence for these trials—again, the logging off of the critical path contends with SPEEDEX at these transaction rates, especially for payment transactions that modify two accounts, instead of just one (as when creating an offer). The throughput reached 255k transactions per second with data persistence enabled.

Production Systems. Finally, we ran the Ethereum Virtual Machine (Geth 1.10 [10]) on a workload of UniswapV2 [24] transactions, and measured a rate of ∼3000 transactions per second (a result in line with other Ethereum benchmarks [107]). The Loopring exchange, built as an L2 rollup on Ethereum, claims a maximum rate of ∼2000 per second [23], a number calculated from Ethereum's per-block computation limit [21], which is in turn set based on the real computational cost of serial transaction execution [26, 45, 47, 91]. Precise measurements of the Stellar blockchain's orderbook DEX suggest that its implementation could handle ∼4000 DEX trades per second.

8 Design Limitations and Mitigations

Latency. Batch trading inherently introduces latency (between order submission and order execution) not present on traditional, centralized exchanges, simply because an order cannot execute until a batch has been closed and clearing prices have been computed. This latency is already present in a blockchain context (a transaction is not finalized until

the consensus protocol adds it to a block), so in this context, SPEEDEX introduces no additional latency.

The latency may have downstream economic effects. Market-making may be more (or less) profitable operating in a batch system, which could lead to reduced (or increased) liquidity. Budish et al. [30] argue that batch trading (between 2 assets) would reduce costs for market-makers, which could lead to increased liquidity. However, they study a higher batch frequency (approximately once per millisecond); our lower batch frequency is less studied (see Q9, [41]).

Tâtonnement Nondeterminism. The algorithms evaluated in §6 can be viewed as a randomized approximation scheme, which raises the question of whether a malicious operator can manipulate the approximation. Note that the level of approximation error (as defined in §B) can be measured, so non-anonymous node operators can be penalized for malfeasance. When regulation is not possible, Tâtonnement can be made deterministic by fixing a set of control parameters for each instance and choosing the solution with the lowest approximation error (or lowest unrealized utility, §6.2). The Stellar implementation uses a static set of control parameters with one Tâtonnement instance. Node operators could also compete to compute prices accurately, as in [7].

Nondeterministic Overdraft Prevention. SPEEDEX needs to prevent an account from spending more than its balance of an asset. As discussed in §3, our implementation considers a proposal valid only if no account is overdrafted after applying the block. This design complicates pipelining of consensus with execution, gives plausible deniability for delaying transactions, and is incompatible with cryptographic commit-reveal schemes.

Instead, given a fixed block of transactions, an implementation could first compute, for each account, the total amount of each asset debited from the account (before applying any credits). If there is any possibility for an account to overdraft in this block, then this amount must exceed the account's balance. As such, to ensure that no accounts overdraft, the implementation can remove all transactions from accounts that might overdraft. Note that this determination is made on a per-account basis, before any transactions are removed, so this filtering requires only one, parallelizable pass over a block of transactions, adding only minimal overhead (§I). Furthermore, only accounts that attempt to overdraft are affected.

Other commutativity conflicts, such as cancelling an offer twice or reusing a sequence number, can be handled similarly, by removing all transactions involved in these conflicts. Note that using these filtering criteria, removing a transaction cannot cause a commutativity conflict. The Stellar blockchain plans this approach.

Other Types of Front-Running. The set of pending transactions is public in many blockchains. One might estimate the clearing prices in a future batch and arbitrage the batch against low-latency markets. This could lead to negative externalities (see [42], footnote 1), and could merit combining SPEEDEX with a commit-reveal scheme such as [52, 117]. Such a design requires the deterministic overdraft-prevention scheme above.

Malicious nodes might also delay transactions. An implementation could buffer several blocks of transactions from a consensus protocol into a single SPEEDEX batch. If even one of these consensus blocks is from an honest replica (that does not censor transactions), a user could ensure that their transaction cannot be delayed from one SPEEDEX batch to the next (by broadcasting to all replicas). This requires a consensus protocol with sufficient *chain quality* [67]. Alternatively, some DAG-based protocols [56, 79] simultaneously commit many blocks of transactions from different replicas. Grouping these blocks into one SPEEDEX batch, instead of ordering them arbitrarily, achieves the same censorship-resistance property. These designs would likewise require the deterministic overdraft-prevention scheme.

Linear Program Scalability. The runtime to solve the linear program increases dramatically beyond 60-80 assets, limiting the number of assets in a SPEEDEX batch. A deployment could take advantage of market structure—there are many assets (e.g., stocks) in the real world, but most are linked to one geographic area or economy, and are primarily traded against one currency. We formally show in §E that in this case, the price computation problem can be decomposed between core pricing (i.e., numeraire) currencies and the external stocks. After running Tâtonnement on the core currencies, each stock can be priced on its own relative to a core currency. This lets SPEEDEX support real-world transaction patterns with an arbitrary number of assets and a small number of pricing currencies.

§D points out that setting the commission to 0 simplifies the linear program to one that is more algorithmically tractable at larger numbers of assets. The Stellar implementation uses this version of the linear program.

Limited Trade Types. Trades on SPEEDEX are limited to trades selling a fixed amount of one asset in exchange for as much as possible of another. SPEEDEX does not implement offers to buy a fixed amount of an asset in exchange for as little as possible of another. These buy offers admit the same logarithmic transformation as in §5.1, but make the price computation problem PPAD-hard, a complexity class that is widely conjectured to be algorithmically intractable in polynomial time (§H). One could compute prices using only sell offers and integrate buy offers in the linear programming step.

Ramseyer et al. [96] show how to integrate Constant Function Market Makers (CFMMs) [28] into the exchange market framework and Tâtonnement. The Stellar implementation uses this integration with its own CFMMs.

9 Implementation Details

The standalone SPEEDEX evaluated in §6 and §7 is a blockchain using HotStuff [115] for consensus. A leader node periodically mints a new block from the memory pool and

feeds the block to the consensus algorithm. Other nodes apply the block once it has been finalized by consensus. A faulty node can propose an invalid block. Consensus may finalize invalid blocks, but these blocks have no effect when applied.

The implementation is available open source at https://github.com/scslab/speedex and consists of ∼30,000 lines of C++20, plus ∼5,000 lines for our Hotstuff implementation. It uses Intel's TBB library [8] to manage parallel work scheduling, the GNU Linear Programming Kit [86] to solve linear programs, and LMDB [50] to manage data persistence (for crash recovery).

Exchange state is stored in a collection of custom Merkle-Patricia tries; hashable tries allow nodes to efficiently compare state (to check consensus) and build short state proofs.

The rest of this section outlines additional design choices built into SPEEDEX. Additional design choices in §K. All optimizations (save §9.1) are implemented in the evaluated system.

9.1 Blockchain Integration

An existing blockchain with its own (non-commutative) semantics can integrate SPEEDEX by splitting block execution into phases: first applying all SPEEDEX transactions (in parallel), then applying legacy transactions (sequentially). SPEEDEX's scalability lets a blockchain charge only a marginal fee for transactions (to prevent spam). A proof-of-stake integration of SPEEDEX could penalize faulty proposals.

SPEEDEX's economic properties are desirable independent of scalability. The initial Stellar implementation uses two-phase blocks, but the SPEEDEX phase is still implemented sequentially. As a result, the initial implementation is simple (adding only ∼5,000 lines to the server daemon) and the primary benefits are economic. However, because the transaction semantics are commutative, engineers can work to parallelize the implementation as needed, without formally upgrading the protocol (which is more difficult than releasing a software update).

9.2 Caches and Tâtonnement

Tâtonnement spends most of its runtime computing demand queries. Each query consists of several binary searches over large lists, so the runtime depends heavily on memory latency and cache performance. Towards the end of Tâtonnement, when the algorithm takes small steps, one query reads almost exactly the same memory locations as the previous query, so the cache miss rate can be extremely low.

Instead of querying the offer tries directly, we precompute for each asset pair a list that records, for each unique limit price, the amount of an asset offered for sale below the price (§G). Laying out this information contiguously improves cache performance.

We also execute the binary searches of one Tâtonnement iteration in parallel. One primary thread computes price updates and wakes helper threads. However, each round of Tâtonnement is already fast on one thread—with 50 assets and millions of offers, one round takes 400–600μs. To minimize synchronization latency and avoid letting the kernel migrate threads between cores (which harms cache performance), we operate these helper threads via spinlocks and memory fences. In the tests of §6, we see minimal benefit beyond 4–6 helper threads, but this suffices to reduce each query to 50–150μs.

Finally, there is a tradeoff between running more copies of Tâtonnement with different settings and the performance of each copy. More concurrent replicas of Tâtonnement mean more cache traffic and higher cache miss rates.

We accelerate the rest of Tâtonnement by exclusively using fixed-point arithmetic (rather than floating-point).

9.3 Batched Trie Design

Our tries use a fan-out of 16 and hash nodes with the 32-byte BLAKE2b cryptographic hash [34]. Both the layout of trie nodes and the work partitioning are designed to avoid having multiple threads writing to the same cache line.

The commutativity of SPEEDEX's semantics opens up an efficient design space for our data structures, which need only materialize state changes once per block. Tries need only recompute a root hash once per block, for example, instead of after every modification. Threads locally build tries recording insertions, which are merged together in one batch operation (which is also parallelizable by redividing local tries into disjoint key ranges). Deletions (when offers are cancelled) are implemented via atomic flags on trie nodes; to enable efficient cleanup of deleted nodes, each node stores the number of deleted nodes beneath it. To facilitate efficient work distribution, each node also stores the number of leaves below it.

SPEEDEX builds in every block an ephemeral trie that logs which accounts are modified; specifically, it maps an account ID to a list of its transactions and to the IDs of transactions from other accounts that modified it. This enables construction of short proofs of account state changes. This trie also uses the same key space as the main account state trie, which lets SPEEDEX use the ephemeral trie to efficiently divide work on the (much larger) account trie.

Memory allocation for an emphemeral trie is trivial because no ephemeral trie node is carried over from one block to the next. Every thread has a local arena, allocation simply increments an arena index, and garbage collection means just setting the index to 0 at the end of a block. We find it to be not a problem if some of the memory in the arena is wasted; we allocate the potential children of an ephemeral trie node contiguously, so a node need only store a 4-byte base pointer (buffer index) and a bitmap denoting the active children. This lets each ephemeral trie node fit in one 64-byte cache line.

10 Related Work

Blockchain Scaling. Our approach is inspired by Clements et al. [51], who improve performance in the Linux kernel through commutative syscall semantics.

Chen et al. [49] speculatively execute Ethereum transactions

to achieve a ∼6x overall execution speedup. Other approaches to concurrent execution include optimistic concurrency control [70, 111], invalidating conflicting transactions [27], broadcasting conflict resolution information [29, 60], or partitioning transactions into nonconflicting sets [35, 74, 116]. This problem is related to that of building deterministic databases and software transactional memory [94, 105, 110]. Li et al. [83] build a distributed database where some transactions are tagged as commutative.

Empirical work [66, 98] finds that a small number of Ethereum contracts, often token contracts, are historically responsible for the majority of conflicts that limit optimistic execution. A recent Solana [112] outage resulted in part when many transactions conflicted on one orderbook contract [99].

Project Hamilton [85] develops a CBDC payments platform. The authors find that totally-ordered semantics become a performance bottleneck. Unlike SPEEDEX, which stores asset balances in accounts, this system requires the more restrictive unspent transaction output (UTXO) model.

Some systems move transaction execution off-chain, into so-called "Layer-2" networks, each with different capabilities, perfomance, interoperability, and security tradeoffs [11, 13, 18, 21, 78, 92, 93]. Other blockchains [6, 27, 103, 109, 118] split state into concurrently-running shards, at the cost of complicating cross-shard transactions.

(Distributed) Exchanges. Budish et al. [42, 43] argue that exchanges should process orders in batches to combat automated arbitrage and improve liquidity.

Other defenses against front-running include cryptographic commit-reveal schemes [52, 72, 100, 117] or "fair" ordering schemes that assume a bounded fraction of malicious nodes [40, 80, 119]. The front-running attacks that SPEEDEX prevents are not guaranteed to be blocked in these schemes. For example, a replica might plausibly front-run a transaction in [80] by investing in lower-latency network links between itself and other replicas than other replicas have with each other, and commit-reveal schemes do not prevent statistical front-running (guessing the contents of a transaction).

Some blockchains build limit-order DEX mechanisms natively [2, 16] or as smart contracts [14]. Smart contracts known as Automated Market-Makers (AMMs) [24, 64, 73, 87] facilitate passive market-making on-chain [28].

0x and a past version of Loopring [19, 108] allow settlement on-chain of orders matched off-chain, in pairs or in cycles. StarkEx [15, 36] gives cryptographic tools to prove correctness of an off-chain exchange.

CoWSwap [5, 7] uses mixed-integer programming to clear offers in batches of at most 100 [20]. Solvers compete to produce the best solution. The former Binance DEX [3] computed per-asset-pair prices in each block. The Penumbra DEX uses homomorphic encryption to privately make batch swaps against an AMM, but cannot let users set limit prices [12].

Price Computation. Our algorithms solve instances of the special case of the Arrow-Debreu exchange market [33] where every utility function is linear. Equilibria can be approximated in these markets using combinatorial algorithms such as those of Jain et al. [77] and Devanur et al. [58] and exactly via the ellipsoid method and simultaneous diophantine approximation [76]. Duan et al. [62] construct an exact combinatorial algorithm, which Garg et al. [69] extend to an algorithm with strongly-polynomial running time. Ye [114] gives a path-following interior point method, and Devanur et al. [57] construct a convex program. Codenotti et al. [53, 54] show that a version of the Tâtonnement process [32] converges to an approximate equilibrium in polynomial time. Garg et al. [68] give another algorithm based on demand queries.

11 Conclusion

SPEEDEX is a fully on-chain DEX that can scale to more than 200,000 transactions per second with tens of millions of open trade offers. SPEEDEX requires no offchain rollups and no sharding of the exchange's logical state. To make SPEEDEX faster, one can simply give SPEEDEX more CPU cores, without changing the semantics or user interface. Because SPEEDEX operates as a logically-unified platform, instead of a sharded network, SPEEDEX does not fragment liquidity between subsystems and creates no cross-rollup arbitrage.

In addition, SPEEDEX displays several independently useful economic properties. It eliminates risk-free front running; any user who can get their offer to the exchange before a block cutoff time can get the same exchange rate as every other trader. SPEEDEX also eliminates internal arbitrage, which disincentivizes network spam. And finally, SPEEDEX eliminates the need to transact through intermediate, reserve currencies, instead allowing a user to trade directly from one asset to any other asset listed on the exchange, with the same or better market liquidity as the trader would have gotten by trading through a series of intermediate currencies.

SPEEDEX is free software, available at https://github.com/scslab/speedex.

Acknowledgements

This research was supported by the Stanford Future of Digital Currency Initiative, the Stanford Center for Blockchain Research, the Office of Naval Research (ONR N00014-19-1-2268), and the Army Research Office (76412CSII). The Stellar blockchain integration was funded by and performed at the Stellar Development Foundation.

The authors wish to thank the anonymous reviewers and our shepherd Siddhartha Sen for their valuable feedback, and thank CloudLab [63] for providing resources for our experiments.

References

[1] Binance chain docs - fees. https://web.archive.org/web/20200617014623/https://

docs.binance.org/guides/concepts/fees.html. Accessed 10/18/2022.

[2] Binance chain docs - introduction. https://web.archive.org/web/20200616190856/https://docs.binance.org/guides/intro.html. Accessed 10/18/2022.

[3] Binance chain docs - match steps and examples. https://web.archive.org/web/20200617065916/https://docs.binance.org/match-examples.html. Accessed 10/18/2022.

[4] Coinbase pricing and fees disclosures. https://help.coinbase.com/en/coinbase/trading-and-funding/pricing-and-fees/fees. Accessed 04/10/2021.

[5] Cow protocol overview: The batch auction optimization problem. https://web.archive.org/web/20220614183101/https://docs.cow.fi/off-chain-services/in-depth-solver-specification/the-batch-auction-optimization-problem. Accessed 10/19/2022.

[6] Eth2 shard chains. https://ethereum.org/en/eth2/shard-chains/. Accessed 03/11/2021.

[7] An exchange protocol for the decentralized web. https://web.archive.org/web/20220825164405/https://docs.gnosis.io/protocol/docs/introduction1/ and https://github.com/gnosis/dex-research/blob/08204510e3047c533ba9ee42bf24f980d087fa78/dFusion/dfusion.v1.pdf and https://github.com/gnosis/dex-research/blob/c56235a3c79fbd85771760ca8826b757fb03eb1f/BatchAuctionOptimization/batchauctions.pdf.

[8] Intel oneapi threading building blocks. "https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html". Accessed 5/6/2021.

[9] The maker protocol: Makerdao's multi-collateral dai (mcd) system. https://makerdao.com/en/whitepaper/. Accessed 12/14/2021.

[10] Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum/tree/release/1.10. Accessed 10/13/2022.

[11] Optimistic rollups. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic$_{r}$ollups/. Accessed 03/11/2021.

[12] The penumbra protocol: Sealed-bid batch swaps. https://web.archive.org/web/20220614034906/https://protocol.penumbra.zone/main/zswap/swap.html. Accessed 10/19/2022.

[13] Polygon lightpaper: Ethereum's internet of blockchains. https://polygon.technology/lightpaper-polygon.pdf. Accessed 12/6/2021.

[14] Serum: Faster, cheaper, and more powerful defi. https://www.projectserum.com/. Accessed 12/6/2021.

[15] Starkex. https://starkware.co/product/starkex/.

[16] Stellar. https://www.stellar.org/.

[17] USDC: the world's leading digital dollar stablecoin. https://www.circle.com/en/usdc. Accessed 12/14/2021.

[18] Zk rollups. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/. Accessed 03/11/2021.

[19] Loopring: A decentralized token exchange protocol. September 2018.

[20] Gpv2 objective criterion. https://web.archive.org/web/20211019155516/https://forum.gnosis.io/t/gpv2-objective-criterion/1254, April 2021. Accessed 04/30/2021.

[21] Loopring 3 design doc. https://web.archive.org/web/20220411224154/https://github.com/Loopring/protocols/blob/master/packages/loopring$_{v}$3/DESIGN.md#results, 2021.

[22] The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. https://web.archive.org/web/20221020032330/https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf, August 2022. Accessed 10/20/22.

[23] Loopring protocol. https://web.archive.org/web/20220409050852/https://loopring.org/#/protocol, April 2022.

[24] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. 2020.

[25] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. Technical report, Tech. rep., Uniswap, 2021.

[26] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. The opbench ethereum opcode benchmark framework:

Design, implementation, validation and experiments. *Performance Evaluation*, 146:102168, 2021.

[27] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[28] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets. *Cryptoeconomic Systems Journal*, 2019.

[29] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 83–92. IEEE, 2019.

[30] Matteo Aquilina, Eric B Budish, and Peter O'Neill. Quantifying the high-frequency trading" arms race": A simple new methodology and estimates. Technical report, Working Paper, 2020.

[31] Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[32] Kenneth J Arrow, Henry D Block, and Leonid Hurwicz. On the stability of the competitive equilibrium, ii. *Econometrica: Journal of the Econometric Society*, pages 82–109, 1959.

[33] Kenneth J Arrow and Gerard Debreu. Existence of an equilibrium for a competitive economy. *Econometrica: Journal of the Econometric Society*, pages 265–290, 1954.

[34] Jean-Philippe Aumasson and Markku-Juhani O Saarinen. The blake2 cryptographic hash and message authentication code (mac). *RFC 7693*, 2015.

[35] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A true concurrent model of smart contracts executions. In *International Conference on Coordination Languages and Models*, pages 243–260. Springer, 2020.

[36] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. 2018.

[37] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, 182(2):418–477, 2002.

[38] Ivan Bogatyy. Implementing ethereum trading front-runs on the bancor exchange in python. https://web.archive.org/web/20220119154606/https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798, Aug 2017.

[39] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[40] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, et al. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. https://research.chain.link/whitepaper-v2.pdf, 2021. Accessed 12/14/2021.

[41] Eric Budish. Response to esma's call for evidence: "periodic auctions for equity instruments" (esma70-156-785). https://ericbudish.org/wp-content/uploads/2022/03/response_esmas_call_evidence_periodic_auctions.pdf, January 2019. Accessed 10/17/2022.

[42] Eric Budish, Peter Cramton, and John Shim. Implementation details for frequent batch auctions: Slowing down markets to the blink of an eye. *American Economic Review*, 104(5):418–24, 2014.

[43] Eric Budish, Peter Cramton, and John Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.

[44] Eric B Budish, Peter Cramton, Albert S Kyle, and Jeongmin Lee. Flow trading. *University of Chicago, Becker Friedman Institute for Economics Working Paper*, (2022-82), 2022.

[45] Vitalik Buterin. A quick explanation of what the point of the eip 2929 gas cost increases in berlin is. https://web.archive.org/web/20211017034159/https://www.reddit.com/r/ethereum/comments/mrl5wg/a_quick_explanation_of_what_the_point_of_the_eip/, April 2021.

[46] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.

[47] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *International conference on information security practice and experience*, pages 3–24. Springer, 2017.

[48] Xi Chen, Dimitris Paparas, and Mihalis Yannakakis. The complexity of non-monotone markets. *Journal of the ACM (JACM)*, 64(3):1–56, 2017.

[49] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum (full version). 2021.

[50] Howard Chu and Symas Corporation. Lightning memory-mapped database manager (lmdb). http://www.lmdb.tech/doc/. Accessed 04/29/2021.

[51] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):1–47, 2015.

[52] Dan Cline, Thaddeus Dryja, and Neha Narula. Clockwork: An exchange protocol for proofs of non front-running.

[53] Bruno Codenotti, Benton McCune, and Kasturi Varadarajan. Market equilibrium via the excess demand function. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 74–83, 2005.

[54] Bruno Codenotti, Sriram V Pemmaraju, and Kasturi R Varadarajan. On the polynomial time computation of equilibria for certain exchange economies.

[55] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[56] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

[57] Nikhil R Devanur, Jugal Garg, and László A Végh. A rational convex program for linear Arrow-Debreu markets. *ACM Transactions on Economics and Computation (TEAC)*, 5(1):1–13, 2016.

[58] Nikhil R Devanur and Vijay V Vazirani. An improved approximation scheme for computing Arrow-Debreu prices for the linear case. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 149–155. Springer, 2003.

[59] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[60] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Computing*, pages 1–17, 2019.

[61] Alexander Domahidi, Eric Chu, and Stephen Boyd. Ecos: An socp solver for embedded systems. In *2013 European Control Conference (ECC)*, pages 3071–3076. IEEE, 2013.

[62] Ran Duan and Kurt Mehlhorn. A combinatorial polynomial algorithm for the linear Arrow–Debreu market. *Information and Computation*, 243:112–132, 2015.

[63] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[64] Michael Egorov. Stableswap-efficient mechanism for stablecoin liquidity. *Retrieved Feb*, 24:2021, 2019.

[65] Stellar Development Foundation. Stellar for cbdcs. https://resources.stellar.org/hubfs/StellarCBDCWhitepaper.pdf. Accessed 2/24/2023.

[66] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. *arXiv preprint arXiv:2201.03749*, 2022.

[67] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310. Springer, 2015.

[68] Jugal Garg, Edin Husić, and László A Végh. Auction algorithms for market equilibrium with weak gross substitute demands and their applications. *arXiv preprint arXiv:1908.07948*, 2019.

[69] Jugal Garg and László A Végh. A strongly polynomial algorithm for linear exchange markets. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 54–65, 2019.

[70] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Yu Xia, Runtian Zhou, and Dahlia Malkhi. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. *arXiv preprint arXiv:2203.06871*, 2022.

[71] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[72] Christopher Goes, Awa Sun Yin, and Adrian Brink. Anoma: Undefining money. 2021.

[73] Eyal Hertzog, Guy Benartzi, and Galia Benartzi. Bancor protocol. 2018.

[74] Graydon Hoare. Core advancement protocol 53: Smart contract data, Mar 2022. https://github.com/stellar/stellar-protocol/blob/master/core/cap-0053.md.

[75] BIS Innovation Hub. Project helvetia phase ii: Settling tokenised assets in wholesale cbdc. https://www.bis.org/publ/othp45.pdf, 2022. Accessed 2/24/2023.

[76] Kamal Jain. A polynomial time algorithm for computing an Arrow–Debreu market equilibrium for linear utilities. *SIAM Journal on Computing*, 37(1):303–318, 2007.

[77] Kamal Jain, Mohammad Mahdian, and Amin Saberi. Approximating market equilibria. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 98–108. Springer, 2003.

[78] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.

[79] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

[80] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.

[81] Zoltán Király and Péter Kovács. Efficient implementations of minimum-cost flow algorithms. *arXiv preprint arXiv:1207.6381*, 2012.

[82] Yudi Levi. Bancor's response to today's smart contract vulnerability. https://web.archive.org/web/20210525131534/https://blog.bancor.network/bancors-response-to-today-s-smart-contract-vulnerability-dc888c589fe4?gi=5e2d9c4ff877, Jun 2020.

[83] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making {Geo-Replicated} systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[84] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 80–96, New York, NY, USA, 2019. Association for Computing Machinery.

[85] James Lovejoy, Cory Fields, Madars Virza, Tyler Frederick, David Urness, Kevin Karwaski, Anders Brownworth, and Neha Narula. A high performance payment processing system designed for central bank digital currencies.

[86] Andrew Makhorin. Glpk (gnu linear programming kit). *http://www.gnu.org/s/glpk/glpk.html*, 2008.

[87] Fernando Martinelli and Nikolai Mushegian. Balancer whitepaper. Technical report, 9 2019. Accessed 2/4/2022.

[88] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32, 2015.

[89] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. http://bitcoin.org/bitcoin.pdf.

[90] Working Group on E-CNY Research and Development of the People's Bank of China. Progress of research and development of E-CNY in china. http://www.pbc.gov.cn/en/3688110/3688172/4157443/4293696/2021071614584691871.pdf, Jul 2021. Accessed 12/14/2021.

[91] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in evm. *arXiv preprint arXiv:1909.07220*, 2019.

[92] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.

[93] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[94] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

[95] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? *arXiv preprint arXiv:2101.05511*, 2021.

[96] Geoffrey Ramseyer, Mohak Goyal, Ashish Goel, and David Mazières. Batch exchanges with constant function market makers: Axioms, equilibria, and computation. *arXiv preprint arXiv:2210.04929*, 2022.

[97] Daniël Reijsbergen and Tien Tuan Anh Dinh. On exploiting transaction concurrency to speed up blockchains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1044–1054. IEEE, 2020.

[98] Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376*, 2019.

[99] Leopold Schabel. Reflections on solana's sept 14 outage. https://web.archive.org/web/20211104012332/https://jumpcrypto.com/reflections-on-the-sept-14-solana-outage/, Oct 2021. Accessed 12/7/2021.

[100] Noah Schmid, Christian Cachin, Orestis Alpos, and Giorgia Marson. Secure causal atomic broadcast, 2021.

[101] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[102] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[103] NEAR Team. Near launches nightshade sharding, paving the way for mass adoption. https://web.archive.org/web/20221007081239/https://near.org/blog/near-launches-nightshade-sharding-paving-the-way-for-mass-adoption/, November 2021. Accessed 10/18/2022.

[104] Tether. Tether: Fiat currencies on the bitcoin blockchain. https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf. Accessed 12/14/2021.

[105] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[106] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[107] Gerui Wang, Shuo Wang, Vivek Bagaria, David Tse, and Pramod Viswanath. Prism removes consensus bottleneck for smart contracts. In *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 68–77. IEEE, 2020.

[108] Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain. 2017.

[109] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 21, 2016.

[110] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: a lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.

[111] Anatoly Yakovenko. Sealevel: Parallel processing thousands of smart contracts. https://web.archive.org/web/20220124143042/https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192. Accessed 12/6/2021.

[112] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. *Whitepaper*, 2018.

[113] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, Renton, WA, April 2022. USENIX Association.

[114] Yinyu Ye. A path to the Arrow–Debreu competitive market equilibrium. *Mathematical Programming*, 111(1-2):315–348, 2008.

[115] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

[116] Wei Yu, Kan Luo, Yi Ding, Guang You, and Kai Hu. A parallel smart contract model. In *Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence*, pages 72–77, 2018.

[117] Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galinanes, and Bryan Ford. Flash freezing flash boys: Countering blockchain front-running. In *The Workshop on Decentralized Internet, Networks, Protocols, and Systems (DINPS)*, 2022.

[118] Jianting Zhang, Zicong Hong, Xiaoyu Qiu, Yufeng Zhan, Song Guo, and Wuhui Chen. Skychain: A deep reinforcement learning-empowered dynamic blockchain sharding system. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[119] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 633–649, 2020.

Appendix A Mathematical Model Underlying SPEEDEX

Mathematically, SPEEDEX relies on a correspondence between a batch of trade offers and an instance of a linear Arrow-Debreu Exchange Market [33]. Specifically, SPEEDEX's batch computation is equivalent to the problem of computing equilibria in these markets.

A.1 Arrow-Debreu Exchange Markets

The Arrow-Debreu Exchange Market is a classic model from the economics and theoretical computer science literature. Conceptually, there exists in this market a set of independent *agents*, each with its own *endowment* of goods. Each agent has some set of preferences over possible collections of goods. These goods are tradeable on an open market, and agents, all at the same time, make any set of trades that they wish with *the market* (or *auctioneer*), not directly with each other.

Definition 1 (Arrow-Debreu Exchange Market)**.** *An Arrow-Debreu Exchange Market consists of a set of goods $\mathfrak{A}$ and a set of agents $j \in \{1,\ldots,M\}$. Every agent j has a utility function $u_j(\cdot)$ and an endowment $e_j \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$.*

When the market trades at prices $p \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$, every agent sells their endowment to the market in exchange for revenue $s_j = p \cdot e_j$, which the agent immediately spends at the market to buy back an optimal bundle of goods $x_j \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$ - that is, $x_j = \text{argmax}_{x:\sum_{\mathcal{A}\in\mathfrak{A}} x_{\mathcal{A}} p_{\mathcal{A}} \leq s_j} u_j(x)$.

There are countless variants on this definition. Typically the utility functions are assumed to be quasi-convex. Some variants include stock dividents, corporations, production of new goods from existing goods, and multiple trading rounds. SPEEDEX uses only the model outlined above—SPEEDEX looks only at snapshots of the market, i.e., once per block, and computes batch results for each block independently.

One potential objection to the above definition is that it assumes that the abstract market has sufficient quantities available so that every agent can make its preferred trades. We say that a market is at *equilibrium* when agents can make their preferred trades and the market does not have a deficit in any good.

Definition 2 (Market Equilibrium)**.** *An equilibrium of an Arrow-Debreu market is a set of prices p and an allocation x_j for every agent j, such that for all goods $\mathcal{A}$, $\sum_j e_{\mathcal{A},j} \geq \sum_j x_{\mathcal{A},j}$, and x_j is an optimal bundle for agent j. The inequality for asset $\mathcal{A}$ is tight whenever $p_{\mathcal{A}}$ is nonzero.*

Note that an equilibrium includes both a set of market prices and a choice of a utility-maximizing set of goods for each agent. Say, for example, there are two goods $\mathcal{A}$ and $\mathcal{B}$, and one unit of each is sold by other agents to the market. If two agents are indifferent to receiving either good, then the equilibrium must specify whether the first receives $\mathcal{A}$ or $\mathcal{B}$, and vice versa for the second. It would not be a market equilibrium for both of these agents to purchase a unit of $\mathcal{A}$ and no units of $\mathcal{B}$.

A.2 From SPEEDEX to Exchange Markets

SPEEDEX users do not submit abstract utility functions to an abstract market. However, most natural types of trade offers can be encoded as a simple utility function.

Specifically, our implementation of SPEEDEX accepts limit sell orders of the following form.

Definition 3 (Limit Sell Offer)**.** *A Sell Offer ($\mathcal{S}$, $\mathcal{B}$, e, α) is request to sell e units of good $\mathcal{S}$ in exchange for some number k units of good $\mathcal{B}$, subject to the condition that $k \geq \alpha e$.*

The user who submits this offer implicitly says that they value k units of $\mathcal{B}$ more than e units of $\mathcal{S}$ if and only if $k \geq \alpha e$. These preferences are representable as a linear utility function.

Theorem 2. *Suppose a user submits a sell offer ($\mathcal{S}$, $\mathcal{B}$, e, α). The optimal behavior of this offer (and the user's implicit preferences) is equivalent to maximizing the function $u(x_{\mathcal{S}}, x_{\mathcal{B}}) = \alpha x_{\mathcal{B}} + x_{\mathcal{B}}$ (for $x_{\mathcal{S}}, x_{\mathcal{B}}$ amounts of goods $\mathcal{S}$ and $\mathcal{B}$).*

Proof. Such an offer makes no trades if $p_{\mathcal{S}}/p_{\mathcal{B}} < \alpha$ and trades in full if $p_{\mathcal{S}}/p_{\mathcal{B}} > \alpha$.

The user starts with k units of $\mathcal{S}$. In the exchange market model, the user can trade these k units of $\mathcal{S}$ in exchange for any quantities $x_{\mathcal{S}}$ of $\mathcal{S}$ and $x_{\mathcal{B}}$ of $\mathcal{B}$, subject to the constraint that $p_{\mathcal{S}} x_{\mathcal{S}} + p_{\mathcal{B}} x_{\mathcal{B}} \leq k p_{\mathcal{S}}$.

The function $u(x_{\mathcal{S}}, x_{\mathcal{B}}) = \alpha x_{\mathcal{B}} + x_{\mathcal{S}}$ is maximized, subject to the above constraint, by $(x_B, x_S) = (0, k)$ precisely when $p_{\mathcal{S}}/p_B < \alpha$ and by $(x_{\mathcal{B}}, x_{\mathcal{S}}) = (k p_{\mathcal{S}}/p_{\mathcal{B}}, 0)$ otherwise (and by any convex combination of the two when $p_{\mathcal{S}}/p_{\mathcal{B}} = \alpha$). These allocations correspond exactly to the optimal behavior of a limit sell offer. □

Note that these utility functions have nonzero marginal utility for only two types of assets, and are not arbitrary linear utilities. Ramseyer et al. [96] find anecdotal evidence that this subclass of utility functions may be analytically more tractable than the case of general linear utilities.

A.3 Existence of Unique* Equilibrium Prices

Theorem 3. *All of the market instances which SPEEDEX considers contain an equilibrium with nonzero prices.*

Proof. All of the utilities of agents derived from limit sell offers are linear (Theorem 2), and have a nonzero marginal utility on the good being sold.

This means our market instances trivially satisfy condition (*) of Devanur et al. [57]. Existence of an equilibrium with nonzero prices follows therefore from Theorem 1 of [57]. □

In fact, all of the equilibria in a market instance contain the same equilibrium prices, unless there are two sets of assets across which no trading activity occurs. In such a case, one might be able to uniformly increase or decrease all the prices together on one set of assets, relative to the other set of assets.

Theorem 4. *Suppose there are two equilibria (p,x) and (p',x') and there exist two assets $\mathcal{A}$ and $\mathcal{B}$ for which $p_\mathcal{A}/p_\mathcal{B} < p'_\mathcal{A}/p'_\mathcal{B}$.*

Then it must be the case that there is a partitioning of the assets $\mathfrak{A}_1, \mathfrak{A}_2$ with $A \in \mathfrak{A}_1, B \in \mathfrak{A}_2$ such that both equilibria include no trading activity across the partition.

Proof. Consider the set of offers trading from $\mathcal{A}$ to $\mathcal{B}$. Let $Z_{\mathcal{A},\mathcal{B}}(r)$ be the set of amounts of asset $\mathcal{A}$ that may be sold (when every agent receives an optimal bundle) by these offers to the market at an exchange rate $r = p_\mathcal{A}/p_\mathcal{B}$. Observe that if $r_1 < r_2$, then every $z_1 \in Z_{\mathcal{A},\mathcal{B}}(r_1)$ is no more than than any $z_2 \in Z_{\mathcal{A},\mathcal{B}}(r_2)$ (as sell offers always prefer higher exchange rates).

At the equilibrium (p,x), let $z_{\mathcal{A},\mathcal{B}}$ be the total amount of $\mathcal{A}$ sold for $\mathcal{B}$ for every asset pair (and $z'_{\mathcal{A},\mathcal{B}}$ similarly for (p',x')). Note that $z_{\mathcal{A},\mathcal{B}} \in Z_{\mathcal{A},\mathcal{B}}(p_\mathcal{A}/p_\mathcal{B})$.

Suppose that there exists a pair of assets $\mathcal{A}, \mathcal{B}$ as in the theorem statement. Then there exists a set of assets $\mathfrak{A}_1$ such that for every asset pair $\mathcal{C} \in \mathfrak{A}_1$ and $\mathcal{D} \notin \mathfrak{A}_1$, $p_\mathcal{C}/p_\mathcal{D} < p'_\mathcal{C}/p'_\mathcal{D}$.

For each of these asset pairs, we must have that $z_{\mathcal{C},\mathcal{D}} \leq z'_{\mathcal{C},\mathcal{D}}$, $z_{\mathcal{D},\mathcal{C}} \geq z'_{\mathcal{D},\mathcal{C}}$, and $\frac{p_\mathcal{C}}{p_\mathcal{D}} z_{\mathcal{C},\mathcal{D}} \leq \frac{p'_\mathcal{C}}{p'_\mathcal{D}} z'_{\mathcal{C},\mathcal{D}}$. Combining these equations gives

$$p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \leq (p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}}) p_\mathcal{D}/p'_\mathcal{D}$$

Each of these inequalities is tight if and only if $z_{\mathcal{C},\mathcal{D}} = 0$.

It is without loss of generality to rescale p' so that $p_\mathcal{D}/p'_\mathcal{D} < 1$ for all $\mathcal{D} \notin \mathfrak{A}_1$. Thus,

$$p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \leq (p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}})$$

Because (p,x) and (p',x') are equilibria, we must have that

$$\begin{aligned} 0 &= \sum_{\mathcal{C} \in \mathfrak{A}_1} \sum_{\mathcal{D} \notin \mathfrak{A}_1} p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \\ &\leq \sum_{\mathcal{C} \in \mathfrak{A}_1} \sum_{\mathcal{D} \notin \mathfrak{A}_1} p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}} \end{aligned}$$

But the second inequality is tight only if each $z_{\mathcal{C},\mathcal{D}} = 0$.

Hence, (p',x') can only be an equilibrium if there exists a partitioning of the assets that separates $\mathcal{A}$ and $\mathcal{B}$, and for which there is no trading activity between the sets in either equilibrium. □

Corollary 1. *Let (p,x) be an equilibrium.*

Construct an undirected graph $G = (V,E)$ with one vertex for each asset, and an edge $e = (\mathcal{A},\mathcal{B}) \in E$ if, at equilibrium, any $\mathcal{A}$ is sold for $\mathcal{B}$ or any $\mathcal{B}$ is sold for $\mathcal{A}$.

If G is connected, then the market equilibrium prices p are unique (up to uniform rescaling).

Proof. If the theorem hypothesis holds, then for any other equilibrium (p',x'), it must be the case that for every asset pair $(\mathcal{A},\mathcal{B})$, $p_\mathcal{A}/p_\mathcal{B} = p'_\mathcal{A}/p'_\mathcal{B}$. By Theorem 4, if this did not hold, then there would exist a partitioning of V into two sets of assets, across which there is no trading at equilibrium (p,x) (contradicting the assumption that G is connected). □

Appendix B Approximation Error

SPEEDEX measures two forms of approximation error: first, every trade is charged a ε transaction commision, and second, some offers with in-the-money limit prices might not be able to be executed (while preserving asset conservation). Formally, the output of the batch price computation is a price $p_\mathcal{A}$ on each asset $\mathcal{A}$, and a trade amount $x_{\mathcal{A},\mathcal{B}}$ denoting the amount of $\mathcal{A}$ sold in exchange for $\mathcal{B}$.

Formally, we say that the result of a batch price computation is (ε,μ)-approximate if:

1. Asset conservation is preserved with an ε commission. The amount of $\mathcal{A}$ sold to the auctioneer, $\Sigma_\mathcal{B} x_{\mathcal{A},\mathcal{B}}$, must exceed the amount of $\mathcal{A}$ bought from the auctioneer, $\Sigma_\mathcal{B}(1-\varepsilon)\frac{p_\mathcal{B}}{p_\mathcal{A}} x_{\mathcal{B},\mathcal{A}}$.

2. No offer trades outside of its limit price. That is to say, an offer selling $\mathcal{A}$ for $\mathcal{B}$ with a limit price of r cannot execute if $\frac{p_\mathcal{A}}{p_\mathcal{B}} < r$.

3. No offer with a limit price "far" from the batch exchange rate does not trade. That is to say, an offer selling $\mathcal{A}$ for $\mathcal{B}$ with a limit price of r must trade in full if $r < (1-\mu)\frac{p_\mathcal{A}}{p_\mathcal{B}}$.

   Intuitively, the lower the limit price, the more an offer prefers trading to not trading.

This notion of approximation is closely related to but not exactly the same as notions of approximation used in the theoretical literature on Arrow-Debreu exchange markets (e.g., [53], Definition 1). In particular, we find it valuable in SPEEDEX to distinguish between the two types of approximation error (and measure each separately) and SPEEDEX must maintain certain guarantees exactly (e.g., assets must be conserved, and no offer can trade outside its limit price).

Appendix C Tâtonnement Modifications

C.1 Price Update Rule

One significant algorithmic difference between the Tâtonnement implemented within SPEEDEX and the Tâtonnement described in Codenotti et al. [53] is the method in which Tâtonnement adjusts prices in response to a demand query. Codenotti et al. use an additive rule that they find amenable to theoretical analysis. If $Z(p)$ is the market demand at prices p, they update prices according to the following rule:

$$p_\mathcal{A} \leftarrow p_\mathcal{A} + Z_\mathcal{A}(p)\delta \tag{1}$$

for some constant δ. The authors show that there is a sufficiently small δ so that Tâtonnement is guaranteed to move closer to an equilibrium after each step.

The relevant constant is unfortunately far too small to be usable in practice, and more generally, we want an algorithm that can quickly adapt to a wide variety of market conditions (not one that always proceeds at a slow pace).

First, we update prices multiplicatively, rather than additively. This dramatically reduces the number of required rounds, especially when Tâtonnement starts at prices that are far from the clearing prices.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+Z_{\mathcal{A}}(p)\delta) \tag{2}$$

Second, we normalize asset amounts by asset prices, so that our algorithm will be invariant to redenominating an asset. It is equivalent to trade 100 pennies or 1 USD, and our algorithm performs better when it can take that kind of context into account.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+p_{\mathcal{A}}Z_{\mathcal{A}}(p)\delta) \tag{3}$$

Next, we make δ a variable factor. We use a heuristic to guide the dynamic adjustment. Our experiments used the l^2 norm of the price-normalized demand vector, $\sum_{\mathcal{A}}(p_{\mathcal{A}}Z_{\mathcal{A}}(p))^2$; other natural heuristics (i.e. other l^p norms) perform comparably (albeit not quite as well). In every round, Tâtonnement computes this heuristic at its current set of candidate prices, and at the prices to which it would move should it take a step with the current step size. If the heuristic goes down, Tâtonnement makes the step and increases the step size, and otherwise decreases the step size. This is akin to a backtracking line search [31, 39] with a weakened termination condition.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+p_{\mathcal{A}}Z_{\mathcal{A}}(p)\delta_t) \tag{4}$$

Finally, we normalize adjustments by a trade volume factor $\nu_{\mathcal{A}}$. Without this adjustment factor, computing prices when one asset is traded much less than another asset takes a large number of rounds, simply because the lesser traded asset's price updates are always of a lower magnitude than those of the more traded asset. Many other numerical optimization problems run most quickly when gradients are normalized (e.g., see [37]).

$\nu_{\mathcal{A}}$ need not be perfectly accurate—indeed, knowing the factor exactly would require first computing clearing prices—but we can estimate it well enough from the trading volume in prior blocks and from trading volume in earlier rounds of Tâtonnement (specifically, we use the minimum of the amount of an asset sold to the auctioneer and the amount bought from the auctioneer). Real-world deployments could estimate these factors using external market data.

Putting everything together gives the following update rule:

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+p_{\mathcal{A}}Z_{\mathcal{A}}(p)\delta_t\nu_{\mathcal{A}}) \tag{5}$$

The step size is represented internally as a 64-bit integer and a constant scaling factor. As mentioned in §5.2, we run several copies of Tâtonnement in parallel with different scaling factors and different volume normalization strategies and take whichever finishes first as the result.

C.1.1 Heuristic Choice

A natural question is why do we use the seemingly theoretically unfounded l^2 norm of the demand vector as our line-search heuristic. A typical line search in an optimization context uses the convex objective function of the optimization problem (e.g., [39]). Devanur et al. [57] even give a convex objective function for computing exchange market equilibria, which we reproduce below (in a simplified form):

$$\sum_{i:mp_i<\frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}} p_{\mathcal{S}_i}E_i\ln\left(mp_i\frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}\right) - y_i\ln(mp_i) \tag{6}$$

for mp_i the minimum limit price of an offer i that sells E_i units of good $\mathcal{S}_i$ and buys good $\mathcal{B}_i$, and $y_i = x_i p_{\mathcal{S}_i}$ for x_i the amount of $\mathcal{S}_i$ sold by the offer to the market.

This objective is accompanied by an asset conservation constraint for each asset $\mathcal{A}$:

$$\sum_{i:\mathcal{S}_i=\mathcal{A}} y_i = \sum_{i:\mathcal{B}_i=\mathcal{A}} y_i \tag{7}$$

However, unlike the problem formulation in [57], Tâtonnement does not have decision variables $\{y_i\}$. Rather, Tâtonnement pretends offers respond rationally to market prices, and then adjusts prices so that constraints become satisfied. As such, mapping our algorithms onto the above formulation would mean that $y_i = p_{\mathcal{S}_i}E_i$ if $mp_i < \frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}$ and 0 otherwise (although §C.2 would slightly change this picture). This would make the objective universally 0, and thus not useful.

We could incorporate the constraints into the objective by using the Lagrangian of the above problem, which gives the objective

$$\sum_{\mathcal{A}}\lambda_{\mathcal{A}}\left(\sum_{i:\mathcal{S}_i=\mathcal{A}} y_i(p) - \sum_{i:\mathcal{B}_i=\mathcal{A}} y_i(p)\right) \tag{8}$$

for a set of langrange multipliers $\{\lambda_{\mathcal{A}}\}$.

We write $y_i(p)$ to denote that in this formulation, offer behavior is directly a function of prices. It appears difficult to use equation 8 directly as an objective to minimize, as it is nonconvex and the gradients of the functions $y_i(\cdot)$ are numerically unstable (even with the application of §C.2).

However, observe that equation 8 is another way of writing "the l^1 norm of the net demand vector" (weighted by the lagrange multipliers). We use the l^2 norm instead of the l^1 to sidestep the need to actually solve for these multipliers.

An observant reader might notice that the derivative of Equation 8 with respect to $\lambda_{\mathcal{A}}$ is the amount by which (the additive version of) Tâtonnement updates $p_{\mathcal{A}}$. This might suggest using $p_{\mathcal{A}}$ in place of $\lambda_{\mathcal{A}}$ in equation 8. However, that search heuristic performs extremely poorly.

C.2 Demand Smoothing

Observe that the demand of a single offer is a (discontinuous) step function; an offer trades in full when the market exchange rate exceeds its limit price, and not at all when the market rate is less than its limit price.

These discontinuities are difficult for Tâtonnement. (Analogously, many optimization problems struggle on non-differentiable objective functions.) As such, we approximate

the behavior of each offer with a continuous function.

Recall that §B measures one form of approximation error (using the parameter μ) which asks how closely realized offer behavior matches optimal offer behavior. Specifically, SPEEDEX wants to maintain the guarantee that every offer (selling A for B) with a limit price below $(1-\mu)\frac{p_A}{p_B}$ trades in full, and those with limit prices above $\frac{p_A}{p_B}$ trade not at all.

As such, SPEEDEX has the flexibility to specify offer behavior on the gap between $(1-\mu)\frac{p_A}{p_B}$ and $\frac{p_A}{p_B}$. Instead of a step function, SPEEDEX linearly interpolates across the gap. That is to say, if $\alpha = \frac{p_A}{p_B}$, we say that an offer with limit price $(1-\mu)\alpha \leq \beta \leq \alpha$ sells an $\frac{\alpha-\beta}{\mu\alpha}$ fraction of its assets.

Observe that as μ gets increasingly small, this linear interpolation becomes an increasingly close approximation of a step function. This explains some of the behavior in Figure 2, particularly why the price computation problem gets increasingly difficult as μ decreases.

C.3 Periodic Feasibility Queries

Tâtonnement's linear interpolation simplifies computing each round, but also restricts the range of prices that meet the approximation criteria, as it does not capitalize on the flexibility we have in handling offers within μ of the market price. As a result, Tâtonnement may arrive at adequate prices without recognizing that fact. To identify good valuations, SPEEDEX runs the more expensive linear program every 1,000 iterations of Tâtonnement.

Appendix D Linear Program

Recall that the role of the linear program in SPEEDEX is to compute the maximum amount of trading activity possible at a given set of prices. That is to say, Tâtonnement first computes an approximate set of market clearing prices, and then SPEEDEX runs this linear program taking the output of Tâtonnement as a set of input, constant parameters.

Throughout the following, we denote the price of an asset A (as output from Tâtonnement) as p_A, and the amount of A sold in exchange for B as $x_{A,B}$. We will also denote the two forms of approximation error as ε and μ, as defined in §B.

To maintain asset conservation, the linear program must satisfy the following constraint for every asset A:

$$\sum_B x_{A,B} \geq \sum_B (1-\varepsilon)\frac{p_B}{p_A} x_{B,A}$$

Define $U_{A,B}$ to be the upper bound on the amount of A that is available for sale by all offers with in the money limit prices (i.e., limit prices at or below $\frac{p_A}{p_B}$), and define $L_{A,B}$ to be the lower bound on the amount of A that must be exchanged for B if SPEEDEX is to be μ-approximate (i.e., execute all offers with minimum prices at or below $(1-\mu)\frac{p_A}{p_B}$, as described in §B).

Then the linear program must also satisfy the constraint, for every asset pair (A,B),

$$L_{A,B} \leq x_{A,B} \leq U_{A,B}$$

Informally, the goal of our linear program is to maximize the total amount of trading activity. Any measurement of trading activity needs to be invariant to redenominating assets; intuitively, it is the same to trade 1 USD or 100 pennies. As such, the objective of our linear program is:

$$\sum_{A,B} p_A x_{A,B}$$

Putting this all together gives the following linear program (let $\mathfrak{A}$ be the set of all assets):

$$\max \sum_{A,B} p_A x_{A,B} \tag{9}$$

$$s.t.\ p_A L_{A,B} \leq p_A x_{A,B} \leq p_A U_{A,B}(p) \quad \forall A,B \in \mathfrak{A}, (A \neq B) \tag{10}$$

$$p_A \sum_{B \in \mathfrak{A}} x_{A,B} \geq (1-\varepsilon) \sum_{B \in \mathfrak{A}} p_B x_{B,A} \quad \forall A \in \mathfrak{A} \tag{11}$$

From the point of view of the linear program, p_A is a constant (for each asset A). As such, this optimization problem is in fact a linear program.

It is possible that Tâtonnement could output prices where this linear program is infeasible (this is the case of the Tâtonnement timeout, as discussed in §6). In these cases, we set the lower bound on each $x_{A,B}$ to be 0 instead of $L_{A,B}$. This change makes the program always feasible (e.g., an assigment of each variable to 0 satisfies the constraints).

Observe that as written, every instance of the variable $x_{A,B}$ appears adjacent to p_A. We can simplify the program by replacing each occurrence of $p_A x_{A,B}$ by a new variable $y_{A,B}$. After solving the program, we can compute $x_{A,B}$ as $\frac{y_{A,B}}{p_A}$.

This substitution gives the following linear program:

$$\max \sum_{A,B} y_{A,B} \tag{12}$$

$$s.t.\ p_A L_{A,B} \leq y_{A,B} \leq p_A U_{A,B}(p) \quad \forall (A,B), (A \neq B) \tag{13}$$

$$\sum_{B \in \mathfrak{A}} y_{A,B} \geq (1-\varepsilon) \sum_{B \in \mathfrak{A}} y_{A,B} \quad \forall A \tag{14}$$

The Stellar implementation charges no transaction commission (i.e., sets ε to 0) in its SPEEDEX deployment. This makes the linear program into an instance of the maximum circulation problem (i.e., variable $y_{A,B}$ denotes the flow from vertex A to vertex B). It is well known that the constraint matrices of these problems are totally unimodular (Chapter 19, Example 4 [101]). This means that it always has an integral solution (Theorem 19.1, [101]) and can be solved by specialized algorithms (such as those outlined in [81]). Some of these algorithms run substantially faster than general simplex-based solvers.

Appendix E Market Structure Decomposition

Suppose that the set of goods could be partitioned between a set of numeraires, which might be traded with any other asset, and a set of stocks, which are only traded with one of the pricing assets.

Then SPEEDEX could compute a batch equilibrium by first computing an equilibrium taking into account only trades between pricing assets, then computing an equilibrium exchange rate for every stock between the stock and its pricing asset, and finally combining the results.

More specifically:

Theorem 5. *Let $\mathfrak{A}$ be the set of numeraires and $\mathfrak{S}$ the set of stocks. A stock $S \in \mathfrak{S}$ is traded with asset $a(S) \in \mathfrak{A}$.*

Suppose (p,x) is an equilibrium for the restricted market instance considering only the numeraires. For each $S \in \mathfrak{S}$, let (r,y) be an equilibrium for the restricted market instance considering only S and $a(S)$.

Then (p',x') is an equilibrium for the entire market instance, where

1. $p'_A = p_A$ *for* $A \in \mathfrak{A}$
2. $p'_S = (r_S / r_{a(S)}) p_{a(S)}$
3. $x'_{A,B} = x_{A,B}$ *for* $A, B \in \mathfrak{A}$
4. $x'_{S,a(S)} = y_{S,a(S)}$
5. $x' = 0$ *otherwise*

Proof. More generally, let G be a graph whose vertices are the traded assets and which contains an edge (A,B) if A and B can be traded directly.

Decompose G into an arbitrary set of edge-disjoint subgraphs $\{G_i\}$, such that any two subgraphs G_i, G_j share at most one common vertex. Then define a graph H whose vertices are the subgraphs G_i, and where a subgraph G_i is connected to G_j if G_i and G_j share a common vertex.

If H is acyclic, then an equilibrium can be reconstructed from equilibria computed independently on each G_i.

We reconstruct a unified set of prices iteratively, traversing along H. Given adjacent G_i and G_j sharing common vertex v_{ij}, let (p^i,x^i) and (p^j,x^j) be equilibria on G_i and G_j, respectively, rescale all of the prices p^j by $p^i_{v_{ij}} / p^j_{v_{ij}}$.

This rescaling constructs a new equilibria $(p^{j\prime},x^j)$for G_j that agrees with that of G_i on the price of the shared good. As such, the combined system $(p^i \cup p^{j\prime}, x^i \cup x^j)$ forms an equilibrium for $G_i \cup G_j$.

This iteration is possible precisely because H is acyclic (a cycle could prevent us from finding a rescaling of some subgraph that satisfied two constraints on the prices of shared vertices). □

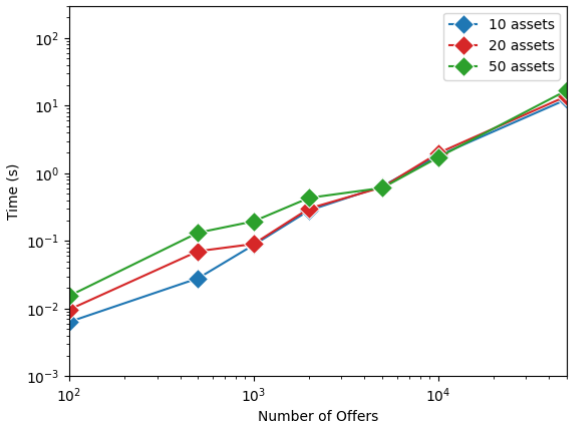


Fig. 8. *Time to solve the convex program of Devanur et al. [57] using the CVXPY toolkit [59], varying the number of assets and offers.*

Appendix F Alternative Batch Solving Strategies

F.1 Convex Optimization

We implemented the convex program of Devanur et al. [57] directly, using the CVXPY toolkit [59] backed by the ECOS convex solver [61]. Figure 8 plots the runtimes we observed to solve the problem while varying the number of assets and offers.

The runtimes are not directly comparable to those of Tâtonnement—namely, this strategy does not have the potential to shortcircuit operation upon early arrival at an equilibrium (our notions of approximation error also do not directly translate to the notions used interally in the solver), nor is it optimized for our particular class of problems.

The important observation is that the runtime of this strategy scales linearly in the number of trade offers. Instances trading 1000 offers, for example, take roughtly 10x as long as instances trading only 100 offers.

This is not a surprising result, given that the number of variables in the convex program scales linearly with the number of trade offers.

The choice of solver strategy does not, of course, change the structure of the input problem instances. The same observation used in §5.1 makes it possible to refactor the convex program so that the number of variables does not depend on the number of open offers, and so that the objective (and its derivatives) can be evaluated in time logarithmic in the number of open offers.

Unfortunately, this transformation makes the objective nondifferentiable. The demand smoothing tactic of §C.2 gives a differentiable but not twice differentiable objective (and presents challenges regarding numerical stability of the derivative). Construction of a convex objective that approximates that of [57] while maintaining sufficient smoothness and numerical stability is an interesting open problem.

F.2 Mixed Integer Programming

Gnosis (Walther, [7]) give several formulations of a batch trading system as mixed-integer programming problems. These formulations track token amounts as integers (instead of as real numbers, as used in Tâtonnement's underlying mathematical formulation), which enables strict conservation of asset amounts with no rounding error.

However, mixed-integer problems appear to be computationally difficult to solve. Walther [7] finds that the runtime of this approach scales faster than linearly. Instances with more than a few hundred assets appear to be intractable for practical systems.

Appendix G Tâtonnement Preprocessing

We include this section so that this paper can provide a comprehensive reference for anyone to develop their own Tâtonnement implementation.

Every demand query in Tâtonnement requires computing, for every asset pair, the amount of the asset available for sale below the queried exchange rate. As discussed in §9.2, Tâtonnement lays out contiguously in memory all the information it needs to return this result quickly.

For a version of Tâtonnement without the demand smoothing of §C.2, a demand query for exchange rate p (i.e. the ratio of the price of the sold asset to the price of the purchased asset)

$$\sum_{i:mp_i \leq p} E_i \tag{15}$$

where mp_i denotes the minimum price of an offer i and E_i denotes the amount of the asset offered for sale.

We can efficiently answer these queries by computing expression 15 for every price p used as a limit price

Demand smoothing complicates the picture. The result of a demand query (with smoothing parameter μ)

$$\sum_{i:mp_i < p(1-\mu)} E_i + \sum_{i:p(1-\mu) \leq mp_i \leq p} E_i * (p - mp_i)/(p\mu) \tag{16}$$

We can rearrange the second term of the summation into

$$1/(p\mu) \sum_{i:p(1-\mu) \leq mp_i \leq p} (pE_i - E_i mp_i) \tag{17}$$

With this, we can efficiently compute the demand query after precomputing, for every unique price p that is used as a limit price, both expression 15 and

$$\sum_{i:mp_i < p} mp_i E_i \tag{18}$$

The division in equation 16 can be avoided by recognizing that Tâtonnement normalizes all asset amounts by asset valuations (so equation 16 is always multiplied by p).

Appendix H Buy Offers are PPAD-hard

A natural type of trade offer is one that offers to sell any number of units of one good to buy a fixed amount of a good (subject to some minimum price constraint). We call these *limit buy offers*.

Example 2 (Limit Buy Offer)**.** *A user offers to buy 100 USD in exchange for EUR, selling as few EUR as possible and only if one EUR trades for at least* 1.1 *USD.*

These offers unfortunately do not satsify a property known as "Weak Gross Substitutability" (WGS, see e.g., [53]). This property captures the core logic of Tâtonnement. If the price of one good rises, the net demand for that good should fall, and the net demand for every other good should rise (or at least, not decrease). Limit sell offers satisfy this property, but limit buy offers do not.

Example 3. *The demand of the offer in of example 2, when* $p_{EUR} = 2$ *and* $p_{USD} = 1$*, is* $(-50\ EUR, 100\ USD)$.

If p_{USD} *rises to* 1.6*, then the demand for the offer is* $(-80\ EUR, 100\ USD)$.

Observe that the price of USD rose and the demand for EUR fell.

Informally speaking, if offers do not satisfy the core logic of Tâtonnement's price update rule, then Tâtonnement cannot handle them in a mathematically sound manner.

More formally, Chen et al. [48] show through Theorem 7 and Example 2.4 that markets consisting of collections of limit buy offers are PPAD-hard. These theorems are phrased in the language of the Arrow-Debreu exchange market model; see §A for the correspondence between SPEEDEX and this model. In fact, the utility functions used in Example 2.4 to demonstrate an example "non-monotone" (i.e., defying WGS) instance are of the type that would arise by mapping limit buy offers into the Arrow-Debreu exchange market model.

Appendix I Deterministic Filtering Performance

The deterministic transaction batch pruning system works by eliminating the transactions from all of the accounts that could create an unresolvable conflict. To be specific, if the sum of the amount of an asset used (either sent in a payment option or locked to create a offer) by all of an account's transactions exceeds that account's balance, then that account's transactions are removed. If an account sends two transactions with the same sequence number (both of which have valid signatures, and the sequence numbers are higher than the sequence number of the account's most recent transaction), or two transactions cancel the same offer ID, then that account's transactions are removed. If two transactions create the same account ID, then both transactions are removed.

We generated batches of 400,000 transactions from the same synthetic transaction model as in §7, and then duplicated 100,000 transactions at random to create a batch of 500,000.

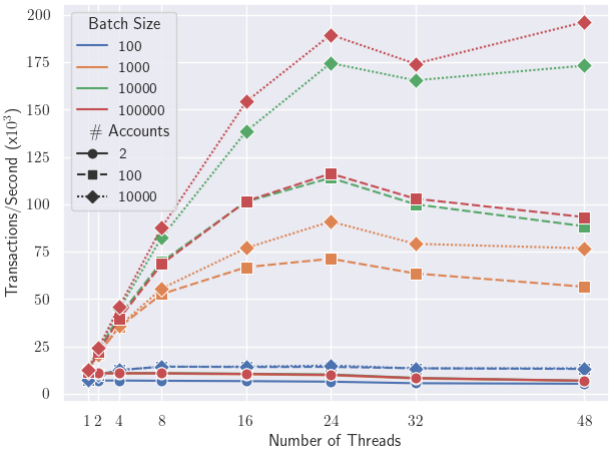


Fig. 9. *Throughput of Block-STM on batches of "Aptos p2p" transactions with varying thread counts (average of 100 trials).*

A small number of accounts (1000) send transactions with conflicting sequence numbers. We initialize the database (again, 10 million accounts) to give each account a small amount of money, and a small number (one or two hundred) of accounts attempt to overdraft.

This filtering takes 0.13s and 0.07s seconds with 24 and 48 threads, respectively (averaged over 50 trials, after a warmup), giving a 21.0× and 38.4× speedup over the serial benchmark. On a more contested benchmark, with only 10,000 accounts (almost all of which overdraft) the maximum speedup over the single threaded trial is only 5.3×, but the overall filtering runtime is still just 0.10s. Our implementation of the filtering is not heavily optimized, but in either parameter setting, the overhead is small.

Appendix J Block-STM Baseline

To provide a baseline for the measurements in Fig. 7, we also ran Block-STM on our hardware (with hyperthreading disabled, as in [70]). Fig. 9 displays the results.

These performance measurements are similar, quantitatively and qualitatively, to those reported in [70] (on different hardware). Note that performance appears to reach a maximum after approximately 16 to 24 threads, and, unlike SPEEDEX, does not effectively use additional hardware beyond this point, even on relatively low-contention workloads.

Appendix K Additional Implementation Details

K.1 Data Organization

Account balances are stored in a Merkle-Patricia trie. However, because a trie is not self-rebalancing, its worst-case adversarial lookup performance can be slow. As such, we store account balances in memory indexed by a red-black tree, with updates pushed to the trie once per block.

For each pair of assets ($\mathcal{A}$, $\mathcal{B}$), we build a trie storing offers selling asset $\mathcal{A}$ in exchange for $\mathcal{B}$. Finally, in each block, we build a trie logging which accounts were modified.

We store information in hashable tries so that nodes can efficiently compare their database state with another replica's (to validate consensus and check for errors), and construct short proofs for users about exchange state.

K.2 Data Storage and Persistence

SPEEDEX uses a combination of an in-memory cache and ACID-compliant databases (several LMDB [50] instances). This choice suffices for our experiments, but a database that persists data in epochs, like Silo [106], or is otherwise optimized for batch operation might improve performance.

Our implementation uses one LMDB instance for the set of open offers, one instance for Hotstuff logs, one instance for storing block headers, and 16 instances for storing account states. LMDB is single-threaded, and we find that the throughput of one thread generating database writes does not keep up with SPEEDEX. Accounts are randomly divided between these instances, according to a hash function keyed by a (persistent) secret key (which is different per blockchain node). This key must be kept secret so as to prevent nodes from denial of service attacks.

Processing transactions in a nondeterministic order complicates recovery from a database snapshot where a block has been partially applied. Cancellation transactions, in particular, refund to an account the remainder of an offer's asset amount. We therefore cannot recover if the snapshot of the orderbooks is more recent than the snapshot of the set of account balances, and our implementation takes care to commit updates to the account LMDB instances before committing updates to the orderbook LMDB.

K.3 Follower Optimizations

A block proposal includes the output of Tâtonnement and the linear program in (the prices and trade amounts, as in §4.2). This permits the nondeterminism in Tâtonnement (§5.2), and lets the other nodes skip the work of running Tâtonnement.

Proposals also include, for every pair of assets, the trie key of the offer with the highest minimum price that trades in that block. When executing a proposal from another node, a follower can compare the trie key of a newly created offer with this marginal key and know immediately whether to make a trade or add the offer to the resting orderbooks. A node also defers all checks that an account balance is not overdrafted to after it has executed all the transactions in a block.

K.4 Replay Prevention

Transactions have per-account sequence numbers to ensure a transaction can execute only once. Many blockchains require sequence numbers from an account to increase strictly sequentially. Our implementation allows small gaps in sequence numbers, but restricts sequence numbers to increase by at most an arbitrary limit (64) in a given block. Allowing

gaps simplifies some clients (such as our open-loop load generator), but more importantly lets validators efficiently track consumed sequence numbers out of order with a fixed-size bitmap and hardware atomics.

The Stellar implementation requires strictly consecutive sequence numbers, mostly for backwards compatibility.

K.5 Fast Offer Sorting

The running times of §6 do not include times to sort or preprocess offers. Naïvely sorting large lists takes a long time. Therefore, we build one trie storing offers per asset pair, and we use an offer's price, written in big-endian, as the first 6 bytes of the offer's 22-byte trie key. Constructing the trie thus automatically sorts offers by price.

Additionally, SPEEDEX executes offers with the lowest minimum prices, so a set of offers executed in a round forms a dense (set of) subtrie(s), which is trivial to remove.

K.6 Nondeterministic Block Assembly

As discussed in §3, SPEEDEX must assemble blocks of transactions in a manner that guarantees no account is overdrafted after applying all of the transactions in the block. The block proposal system (Fig. 1, 2) manages this by carefully controlling writes to shared state.

The proposal module takes as input a set of unconfirmed transactions (the "mempool", in typical blockchain parlance) and outputs a proposed block containing a subset of the unconfirmed transactions. For each candidate unconfirmed transaction, a thread reserves the ability to perform all necessary modifications by "locking" all relevant data elements. Once a transaction acquires all of its locks, it performs its necessary state modifications and finally releases the locks. If it cannot acquire all necessary locks, it releases any locks and excludes the transaction from the proposed block.

Conceptually, a transaction offering a trade or sending a payment must lock the number of units of assets that could be debited from the account if the operation succeeds. However, doing this with spinlocks would preclude the scalability displayed in Figure 7. Instead, most reservations are performed with hardware atomics to decrement the number of available units. Crediting an account can never fail because SPEEDEX caps the total amount of any asset issued at `INT64_MAX`. This process is conservative in that it may reject transactions that could have executed safely.

Unique offer IDs ensure that no offer is created twice, and atomic boolean flags ensure an offer cannot be cancelled twice. Sequence numbers can be reserved by atomic bitmaps (as in §K.4). For simplicity, our implementation does use exclusive locks when creating new accounts (which we assume occurs relatively infrequently).

Appendix L Additional Replicas

SPEEDEX invokes a consensus protocol no more than once per second in our experiments. To demonstrate that this overhead is negligible, we ran SPEEDEX with 10 replicas, although with weaker hardware per replica, due to resource limitations. Each replica is one AWS c5ad.16xlarge instance, with one AMD EPYC 7R32 processor (48 CPUs @ 2.8Ghz per physical chip, 32 of which are allocated to our instances), 128 GB of memory, and two 1.1TB NVMe drives in a RAID0 configuration. Performance measurements are plotted in Figure 10.

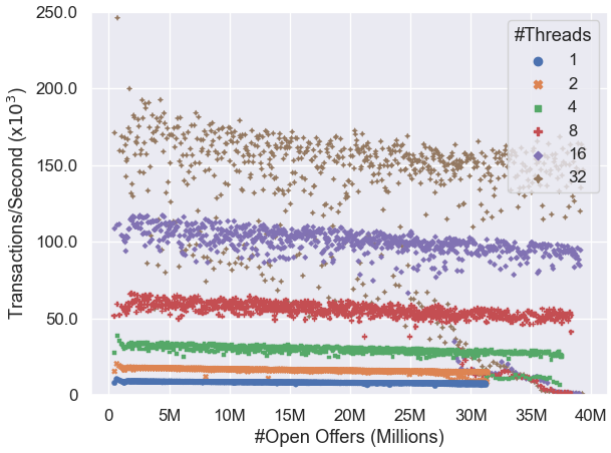


Fig. 10. *Transactions per second on SPEEDEX when running with 10 replicas (on weaker hardware than in Fig. 3), plotted over the number of open offers.*

The overall throughput numbers are lower here than in Figure 3 due to the weaker hardware, but the scalability trends are the same. Doubling the thread count increases performance by a factor of between 1.8x and 1.9x, except that the jump from 16 to 32 gives a roughly 1.4x increase due to contention with background tasks (particularly logging to persistent storage).

This graph also highlights how SPEEDEX responds to insufficient hardware resources. As the number of open offers increases, SPEEDEX's memory requirements increase. Eventually, memory starts to be paged to disk, which dramatically increases disk usage and contends with the logging to persistent storage. SPEEDEX slows down in response, to ensure for safety that data in peristent storage is never too far out of sync.

Boomerang: Metadata-Private Messaging under Hardware Trust

Peipei Jiang[1,2] Qian Wang[1,*] Jianhao Cheng[1] Cong Wang[2] Lei Xu[3]
Xinyu Wang[4] Yihao Wu[1] Xiaoyuan Li[1] Kui Ren[5]
[1] *School of Cyber Science and Engineering, Wuhan University* [2] *City University of Hong Kong*
[3] *Nanjing University of Science and Technology* [4] *Tencent Inc.* [5] *Zhejiang University*

Abstract

In end-to-end encrypted (E2EE) messaging systems, protecting communication metadata, such as who is communicating with whom, at what time, etc., remains a challenging problem. Existing designs mostly fall into the balancing act among security, performance, and trust assumptions: 1) designs with cryptographic security often use hefty operations, incurring performance roadblocks and expensive operational costs for large-scale deployment; 2) more performant systems often follow a weaker security guarantee, like differential privacy, and generally demand more trust from the involved servers. So far, there has been no dominant solution. In this paper, we take a different technical route from prior art, and propose Boomerang, an alternative metadata-private messaging system leveraging the readily available trust assumption on secure enclaves (as those emerging in the cloud). Through a number of carefully tailored oblivious techniques on message shuffling, workload distribution, and proactive patching of the communication pattern, Boomerang brings together low latency, horizontal scalability, and cryptographic security, without prohibitive extra cost. With 32 machines, Boomerang achieves 99th percentile latency of 7.76 seconds for 2^{20} clients. We hope Boomerang offers attractive alternative options to the current landscape of metadata-private messaging designs.

1 Introduction

In E2EE messaging systems [59], the exposure of privacy-revealing communication metadata, including the identities of the communicating parties, and the timing and volume of the traffic, remains a big concern [26,39,61]. Communication metadata can not only be used to target whistleblowers and journalists [37, 70], but also serve as a surveillance means to reveal intimate details of a person's life [38]. Designing a metadata-private messaging system is a challenging problem, given the powerful attackers that can monitor and actively interfere with the network traffic [31,39]. The popular system in practice today that hides communication metadata, namely Tor [34], is not resilient against even passive traffic analysis attacks (e.g., through timing, volume patterns, etc.) [39]. Because of this, academic research systems have been developed recently with well-defined security guarantees for improved metadata privacy [3–6, 8, 11, 15, 16, 27–29, 36, 37, 52, 54, 56–58, 70, 72, 88, 91]. Ideally, such a system must ensure that any pair of connected clients might be communicating from the view of powerful attackers. This implies two necessary requirements: 1) unlinking senders and receivers during any message exchange; and 2) maintaining the same communication pattern across all connected clients, to resist traffic analysis. Roughly speaking, most prior art on metadata-private messaging falls into the balancing act among security, performance, and trust assumptions, while trying to meet these two requirements (more discussions in §7).

Among the diverse landscape of metadata-private messaging designs, there are two commonalities of state-of-the-art systems: 1) leveraging an intermediate "virtual address" to facilitate obfuscated message "drop" and "fetch" between the communicating pairs; and 2) operating in rounds. Designs with cryptographic security follow technical routes of sophisticated cryptographic operations that either obliviously "write to (drop)" [5, 27, 37, 52, 54, 70] or obliviously "read from (fetch)" the virtual address [4, 8]. The hefty operations, however, often present performance roadblocks and unfavorable operational dollar costs, hindering large-scale voluntary adoptions in practice. More performant systems [11, 56, 87, 88] choose to relax the security guarantee and use random noise to disguise the observable "drop" / "fetch" at the virtual address in the framework of differential privacy. Generally, these systems need to trust a fraction of servers for the claimed security, where the latency would be increased if trusting fewer servers. So far there has yet to be any dominant solution.

Motivations. In this paper we propose Boomerang, an alternative metadata-private messaging system leveraging secure enclaves. Boomerang takes a different technical route from prior art, and is partially motivated by the readily available

*Qian Wang is the corresponding author.

trust assumption on hardware enclaves. As emerging in the cloud [78, 81], secure enclaves provide a convenient and native implementation choice to build secure yet sophisticated systems in practical settings.

Like much prior art [24, 32, 35, 42, 49, 50, 74], Boomerang leverages hardware enclaves for both performance and security. Performance is arguably one of the key reasons for Tor's wide adoption in practice [39]. Yet, this is what most prior art on metadata-private messaging is still lacking. We believe that using hardware enclaves to improve performance could be the key missing ingredient for wider adoption of metadata-private messaging systems in practice. Under hardware trust, Boomerang hides users' online communication behaviors with a strong guarantee of metadata privacy and resilience against powerful attackers, while retaining good enough performance.

Besides, we envision that there could be broader options of technical routes for metadata-private messaging, with different trust assumptions, performance, and security guarantees. For whistleblowers [37, 70], their most reasonable choice is to go for designs with a cryptographic guarantee and zero trust in the servers [8, 39], where performance might be much less concerning. For the mass general users who want more than E2EE messaging (e.g., due to concerns about metadata tracking [38, 47]), we hope Boomerang could offer a better option with lowered operational cost and improved performance. These benefits can potentially attract a large user base, which is crucial in private communication systems [33].

Challenges. While trusting the enclaves brings easy-to-see benefits, such as using fewer servers for traffic mixing than prior art for reduced latency, it does not entirely solve the problems in building a scalable metadata-private messaging system. This is because: 1) secure enclaves exhibit their own threat models and attack surfaces, especially the leakage of memory access patterns [32, 90], which demand tailored system structure and oblivious algorithm designs; 2) as acknowledged by prior art [39, 54, 57, 88], even with just one trusted server, dealing with the thorny problems of active attackers that may selectively interfere with traffic, e.g., disconnecting selected clients to gain advantages in identifying targeted communicating pairs, remains hard to address; and 3) as privacy loves company [33], the need to support more clients suggests the necessity of pushing for horizontal scaling designs, which is also a hot topic in recent years [11, 52, 54, 56, 57, 87]. Notably, scalability is not only a usability requirement but also a security demand [33, 39, 54, 56, 87].

Technical overview. For metadata-private messaging, Boomerang draws many insights from the prior art, and is designed to be a performant system with cryptographic security under hardware trust. It operates in rounds for bi-directional conversations, and centers around the paradigm of adopting a private "virtual address" to facilitate obfuscated message exchange that unlinks the sender and receiver. As we overview below, its technical instantiation involves tailored oblivious algorithms for message shuffling, proactive resistance against active attacks, and horizontal scaling. For ease of presentation, we present a basic single-server Boomerang (§3) and a scalable multi-server Boomerang+ (§4).

1) Basic single-server Boomerang. From a high level, the system operates as follows: Upon proper setup, in each round, each connected client sends a message, tagged with a "private label", which is randomly derived from a pairwise shared secret with his communicating buddy, to the Boomerang server. The Boomerang server obliviously checks all the messages and swaps any pair of messages sharing the same labels for the relevant communicating pair. In this regular case, each label shows up twice each round. But active attackers might block selected clients or control a subset of clients to disrupt this regular pattern, causing each label to show up once or more than twice each round. The problem is quite subtle, because we need to fix these irregular patterns, without giving attackers any advantage in linking the remaining clients.

Boomerang's key insight is to preserve the same observable receiving pattern for each connected client, no matter how the sending pattern changes. We build oblivious algorithms for enclaves to detect messages with irregular label patterns and proactively patch them (§3.3.2) by returning those messages back to the corresponding senders, like a "boomerang". In this way, we can contain the influence attempts within the problematic clients themselves, isolated from the remaining clients. Based on a library of basic general-purpose oblivious primitives [2, 71], we design specialized oblivious algorithms (§3.3.2, §3.3.3, §4.2) for all enclave operations in Boomerang, including proactive resistance designs against active attacks and horizontal scaling in Boomerang+.

2) Scalable multi-server Boomerang+. For horizontal scalability, directly replicating the basic single-server Boomerang and letting each server process a subset of communicating pairs would not work, because pairwise clients connecting to one server have a higher possibility to communicate than those across different servers. Introducing a load balancer, known as an entry node in Boomerang+, to obliviously distribute batches of messages to a group of Boomerang nodes for message exchange would fulfill the need for "global mixing", where any pair of connected clients at the entry node might be communicating. But as pointed out by recent art [32, 89], two requirements remain: 1) a centralized proxy [82] can be error-prone and a scalability bottleneck of the underlying system; and 2) any batch structure from a load balancer must be generated using public information, so as to ensure that no sensitive information can be observable from the load balancing.

Note that these requirements are generic for any security-aware scalable system design. Answering them can be quite design specific, especially on setting the bound on batch size without triggering an overflow. Inspired by the balls-into-bins analysis [32], we model the problem of setting the maxi-

mum batch size in Boomerang+ as a weighted balls-into-bins game, where we partition the messages by their private labels and each message's weight is determined by its label pattern (showing once, twice, or more than twice). We prove an upper bound on the maximum batch size and show its marginal overhead on our horizontal scaling design. Following the oblivious load balancer design [32], we derive tailored oblivious algorithms to generate sub-batches whose distribution across rounds is indistinguishable (§4.2).

2 Threat Model and Security Goals

2.1 Threat Model and Assumptions

We consider the following threats. Attackers can: 1) observe the global network states (including the timing, volume, and link states); 2) actively tamper with the network, such as selectively dropping messages, blocking selected connections, and controlling a subset of clients; and 3) access the server-side components beyond hardware enclaves, such as the memory, files, and networks, as well as the operating system.

Boomerang is built on servers equipped with secure enclaves, where memory access patterns can be observed [32, 49, 62, 66]. Boomerang is designed to work with generic enclaves [18, 30], and we adopt Intel SGX for our implementation. Our oblivious algorithms can deal with side-channel attacks against SGX leveraging memory access patterns to extract secrets, such as the cache attacks [19, 41, 67] and paging-based attacks [20]. We assume a public key infrastructure (PKI) to help manage the public keys of clients. Communications among clients and enclaves are securely established via TLS integrated with remote attestation [1, 46, 51]. In Appendix A, we elaborate in more detail on how a client can establish the trust on a multi-enclave system like Boomerang through remote attestation, following common practices suggested from the prior art [9, 23, 35, 74, 79].

Communication model. Like many previous metadata-private messaging systems [4, 8, 54, 56, 57, 87, 88], Boomerang operates in rounds to ensure the uniformity of communication pattern among clients, and focuses on pairwise messaging among online clients who have coordinated their conversations. Processing message exchanges on round boundaries implies that clients of Boomerang send encrypted messages with a fixed rate and size, independent of their true communication activities, which allows the dealing against traffic analysis attacks. This can be done at Boomerang clients by generating "blank" messages if a user types nothing or too slow, and queuing/splitting messages if a user types too fast or sends a message of large size [88].

Under this communication model, Boomerang does not hide the fact that clients are using the system. Boomerang offers online anonymity by supporting a large scale of clients. The anonymity set includes both active clients in real conversations with their buddies and online idle clients who do not have a conversation buddy but can voluntarily send "blank" messages to themselves as cover traffic to further enlarge this anonymity set [56, 88]. We suggest the clients always keep online to disguise the real communication actions [33, 39]. Because all clients connected in the system behave the same at each communication round, we can hide the communication metadata with cryptographic security against powerful attackers. We formalize this security notion in Definition 2.1.

Bootstrapping Boomerang. Besides operating in rounds, Boomerang adopts the existing practices of a bootstrapping phase for clients to start conversations [11, 37, 52, 58]. Particularly, clients should run an "add-friend" protocol (where clients can verify each other's identity and share their secrets) and a "dialing" protocol (where pairwise clients coordinate the time to have the conversation and exchange session keys) [7]. To add a friend, the common practice is to: 1) exchange secrets in person (e.g., by showing a QR code at a coffee shop [7, 37]); or 2) use an online metadata-private add-friend protocol [58]. To dial a friend, the common practice is to adopt an out-of-band metadata-private dialing system, e.g., Alpenhorn [58]. Dialing brings additional costs to clients, which will be amortized over multiple conversation rounds. Note that similar bootstrapping phases have been adopted by prior art [54, 56, 57, 87]. Thus, throughout the paper, we keep our focus on the conversation protocol design, which is where much prior art is differentiated [7], and make simplified assumptions that the bootstrapping phase is properly done. Specifically, upon proper bootstrapping, we will narrow down the problem of metadata-private messaging to how to design an obfuscated message exchange (aka conversation) protocol among pairwise clients under hardware trust.

Attacks out of scope. Similar to other enclave-based systems [32, 35, 49, 75], we don't consider denial-of-service attacks. The enclave code is assumed to be correct and faithfully fulfilling our oblivious algorithms. Orthogonal countermeasures to recent noteworthy leakage attacks through power consumption channels [25, 68] and transient executions [48, 77] are also beyond the scope of this work. One generic limitation to all metadata-private messaging systems is the long-term intersection threat [14]. Recall that we do not hide whether clients are using the systems or not. Thus, an attacker, who observes the anonymity set of online clients changing across rounds, might infer the linkage of communicating pairs. For example, two clients that simultaneously get online or offline are more likely to be communicating. The common practice to mitigate this concern is to let the clients always stay online [56, 88] or keep the same communication pattern [11] and send enough cover traffic. We also suggest adopting orthogonal mitigation techniques [60, 92] and using more cover traffic, as also noted by Clarion [36].

2.2 Security Goals

Like much prior art, our security goal follows the unobservability concept in anonymous communication [43]. Specifi-

cally, for any two online clients, Alice and Bob, we want to ensure that an attacker cannot distinguish whether they are real buddies in a conversation or not. We want to guarantee this is true even in a malicious setting, where an attacker can control up to $m-2$ clients for m as the total number of clients connected to the system. We formalize this property below as communication pattern indistinguishability.

Definition 2.1 *Let λ be the security parameter and m be the number of connected clients in the system. Define an experiment EXP with an attacker $\mathcal{A}$ who controls $m-2$ clients:*

- *The attacker creates a pair of clients in a conversation as $P_0 = (c_0, c_1)$, and a pair of clients not in a conversation as $P_1 = (c'_0, c'_1)$ (e.g., being idle or missing buddies).*
- *The challenger randomly chooses $b \in \{0,1\}$ and plays the role of the pair of clients P_b to simulate interactions between them and the server. During this procedure, the challenger needs to simulate the payload (aka contents that can be represented by a random string) for the chosen pair of clients.*
- *The attacker observes their transcriptions and outputs a bit $b' \in \{0,1\}$ to guess the challenger's choice.*

We define the attacker $\mathcal{A}$'s advantage with respect to EXP as

$$Adv_{EXP,\mathcal{A}} = \left|\Pr[b = b'] - \Pr[b \neq b']\right|.$$

We say that a system is communication pattern indistinguishable if the advantage $Adv_{EXP,\mathcal{A}}$ is negligible in λ for all probabilistic polynomial time (PPT) attackers.

This definition implies that seemingly any pair of connected clients might be communicating with equal chance. We will show this is indeed the case in Boomerang and Boomerang+.

3 Boomerang: Basic Instantiation

3.1 Overview

Boomerang runs in rounds with a single enclave-based server. Following the same practice in previous round-based designs [54, 56, 57, 87, 88], Boomerang requires a coordinator to announce round numbers across the server and clients. In each round, each client sends and receives one message respectively to and from the Boomerang server. To facilitate oblivious message swapping, every message is tagged with a "private label", which is a pseudorandom string generated from a pairwise session secret between a communicating pair. (See §2 for our system setup assumptions.) Any pair of messages sharing the same private label will be swapped, which indicates a regular case when each private label shows up exactly twice each round. Messages with irregular private label patterns will be detected and looped back (§3.3.2).

Figure 1 shows an overview of our Boomerang design. The Boomerang server first (❶) sorts the packets and detects the irregular pattern of the private labels. Then, (❷) it proactively fixes the irregular pattern via the proactive pattern patching algorithm (§3.3.2). Finally, (❸) it sorts the messages by receiver IDs and sends them to the clients (§3.3.3).

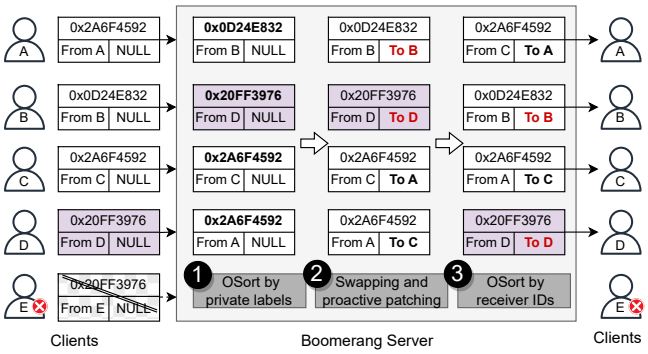


Figure 1: Overview of Boomerang. In this example, A and C are talking. B is idle. D and E are talking, but E is blocked as described at the end of §3.3.2. Boomerang proactively patches the pattern for D and B (Step ❷).

3.2 Client

Figure 2 shows the pseudocode for client operations. We consider two modes of clients: active clients in conversations with their buddies and idle clients sending dummy cover traffic (as discussed in §2.1). Regardless of being active or idle, each connected client needs to send one message to the Boomerang server in each round.

Packet preparation. A packet `Pkt` includes the following fields: 1) private label with l bits, `priv_label`; 2) receiver's identifier, `R`; 3) sender's identifier, `S`; 4) round number, `round_num`; and 5) content encrypted by the session key shared with the buddy (Lines 4-6). When preparing the packet, the client fills in all fields except for the receiver's identifier (e.g., set to `NULL` by default). If the active client has nothing to deliver to her buddy, she should fill the content anyway, e.g., with a message saying "this is a blank message". The client then encrypts the packet using the session key shared with the secure enclaves (Line 7). Finally, the client sends the encrypted message to the Boomerang server.

Idle clients. To hide real communication actions, clients need to keep sending messages even if they are not in an active conversation [56, 88]. We let idle clients randomly generate a private label (not shared with others). By design, the server will loop all unpaired messages back to senders.

Exception alert. Normally, after sending the prepared packet to the Boomerang server, the client is expected to receive one packet carrying the message from the buddy/herself every round. Otherwise, the system will raise exceptions for abnormal cases. If there is no message returned by the decryption procedure, it means that the packet to/from the server has been lost. In this case, both active and idle clients should be

```
1  def active_client(round_num, priv_label, my_id,\
2                content, ses_key_encl, ses_key_buddy):
3      # Prepare the packet
4      Pkt pkt; pkt.round_num = round_num
5      pkt.priv_label = priv_label; pkt.S = my_id
6      pkt.enc_content = Encrypt(content, ses_key_buddy)
7      enc_pkt = Encrypt(pkt, ses_key_encl)
8      # Send the packet to the Boomerang server \
9      #                   running boomerang_server()
10     send_msg = (my_id, enc_pkt)
11     # Receive the packet from the Boomerang server
12     recv_msg = rpc.request(send_msg)
13     pkt = Decrypt(recv_msg, ses_key_encl)
14     # Exception alert
15     # If there is message loss
16     if pkt ==  None: raise("Message loss")
17     # If the original message is looped back
18     if pkt.R == my_id: raise("Buddy blocked")
19
20     return Decrypt(pkt.enc_content, ses_key_buddy)
21
22 def idle_client(round_num, my_id, ses_key_encl):
23     # Prepare the packet
24     Pkt pkt; pkt.round_num = round_num
25     pkt.priv_label = random(); pkt.S = my_id
26     pkt.enc_content = ciphertext.random()
27
28     send_msg = (my_id, Encrypt(pkt, ses_key_encl))
29     recv_msg = rpc.request(send_msg)
30
31     if Decrypt(recv_msg, ses_key_encl) ==  None:
32         raise("Message loss")
33
34     return True
```

Figure 2: Pseudocode for client operations.

alerted about their unreliable network conditions (Lines 13-16 and Lines 31-32). For those successfully decrypted packets, active clients then check whether their messages are from their buddies (Line 18). Note that in Boomerang design, if the receiver ID is the same as the sender ID (indicating that the message is sent back like a "boomerang"), the client should be alerted that his/her buddy has been blocked and can choose to resend the message in the next round or stop the conversation immediately (by changing to idle mode).

3.3 Server

Figure 3 shows the pseudocode for oblivious server operations, including two main functions: 1) oblivious irregular pattern detection and 2) oblivious proactive pattern patching. Below we introduce the background of oblivious primitives and then describe the main ideas of our designs.

3.3.1 Background of Oblivious Primitives

We build Boomerang's oblivious algorithms over an existing library of general-purpose oblivious primitives developed by XGBoost contributors [2, 55], which is also based on libraries provided in previous noteworthy enclave-based data analytic systems [71, 73]. The library offers basic oblivious functions, including comparisons, assignments, sorting, etc. These oblivious functions are fundamentally built on register-to-register operators, which are private to the processor and immune to memory access pattern leakages [55, 71, 73].

Oblivious comparisons/assignments. This set of primitives can conditionally assign or compare values on the register level without revealing the results of the comparison or assignment. In this paper, we use `O_Equal(a,b)`, `O_Less(a,b)`, and `O_Choose(cond,a,b)` [2] for comparison and conditional assignment. `O_Equal(a,b)` outputs `True` if `a` = `b` (otherwise outputs `False`). `O_Less(a,b)` outputs `True` if `a` $\leq$ `b` (otherwise outputs `False`). `O_Choose(cond,a,b)` chooses from two values given a boolean condition without leaking which value is chosen. If `cond` = `True`, it outputs value `a` (otherwise outputs value `b`).

Oblivious sort (`O_Sort(key,array)`). An oblivious sort algorithm outputs ordered data without revealing any information (e.g., the original order) about the input data [10]. In our instantiation, we choose bitonic sort [12], which is highly parallelizable with $O(n\log^2 n)$ computational complexity. Bitonic sort compares and swaps the items in a fixed and data-independent order, and thus is oblivious by design.

3.3.2 Oblivious Proactive Pattern Patching

Detecting irregular pattern. When receiving a batch, the server first needs to identify all the regular and irregular messages for subsequent processing. The server first obliviously sorts the batch by `priv_label` (Step 2 in Figure 3, Line 12). This step is to map the messages of the same label (which implies a communicating pair) together for further pattern detection. Before introducing the detection algorithm, we first show the possible label patterns after sorting. Since the private label is a l-bit pseudorandom string shared between the pair, the possibility that attackers can guess it and forge messages is negligible if l is sufficiently large (e.g., 256 bits). Therefore, a communicating pattern indicates a double-accessed label, which we regard as a regular pattern. Otherwise, the pattern is regarded as irregular. There are three possible label patterns after `O_Sort`, as shown below

- double: messages from communicating pairs;
- single: messages from idle clients or incomplete pairs (caused by client churn or malicious blocking);
- more-than-two: multiple messages with repeating labels controlled by an attacker.

To detect irregular patterns, the server linearly scans the ordered packets using oblivious comparison primitives to identify the above three patterns. Basically, we can achieve this by scanning the messages with a sliding window of three each time, as shown in Step 3.1 in Figure 3. For each message, we compare its private label with its previous, next, and next next messages, respectively (Lines 17-22). This will give us the relationship of the packets in the sliding window.

We first clarify how to determine more-than-two cases. If the label of a message is the same as its two subsequent messages (`is_next2_same` = `True`), this implies that this private

```
1  def boomerang_server(round_num, recv_msgs, session_keys):
2      pkts = [], R_list = [] # Receiver IDs
3      recv_msgs = Dedup(recv_msgs)
4      # Step 1: decrypt the messages
5      for (S, enc_pkt) in recv_msgs:
6          pkt = Decrypt(enc_pkt,session_keys[S])
7          # Filter out messages not in this round
8          if pkt.round_num != round_num:
9              continue
10         pkts.append(pkt)
11     # Step 2: Pair the private labels
12     pkts.O_Sort(key=pkt.priv_label)
13     # Step 3: Oblivious proactive pattern patching
14     is_mtt = False # mtt: more_than_two
15     for pkt in pkts:
16         # Step 3.1: Detect irregular pattern
17         is_prev_same = O_Equal(pkt.priv_label,\
18                                Prev(pkt).priv_label)
19         is_next_same = O_Equal(pkt.priv_label, \
20                                Next(pkt).priv_label)
21         is_next2_same = O_Equal(pkt.priv_label, \
22                            Next(Next(pkt)).priv_label)
23         # Detect more-than-two (mtt) pattern
24         is_mtt = is_mtt and is_prev_same or is_next2_same
25         # Detect and patch single patterns in Lines 27-28
26         # Step 3.2: Swapping and patching
27         pkt.R = O_Choose(is_next_same, Next(pkt).S, pkt.S)
28         pkt.R = O_Choose(is_prev_same, Prev(pkt).S, pkt.R)
29         pkt.R = O_Choose(is_mtt, pkt.S, pkt.R)
30     # Step 4: Re-order messages by receiver IDs
31     pkts.O_Sort(key=pkt.R)
32     # Step 5: Encrypt and send the messages
33     send_msgs = {}
34     for pkt in pkts:
35         enc_pkt = Encrypt(pkt, session_keys[pkt.R])
36         send_msgs[pkt.R] = enc_pkt
37         R_list.append(R)
38     rpc.response(R_list, send_msgs)
```

Figure 3: Pseudocode for server operations.

label must occur at least three times. Then, we can identify this message as a more-than-two case (by setting the flag `is_mtt` = `True`). Another observation is that, if a message's label repeats as a detected more-than-two case, this message must belong to the same case. We identify this by checking whether a message's previous neighbor belongs to a more-than-two case (based on the flag `is_mtt` as set in the previous iteration), and whether the message's label repeats that of its previous neighbor (based on the flag `is_prev_same`).

To determine single cases, we check both the message's previous and next neighbors. If inequality holds for both neighbors (`is_prev_same` = `False` and `is_next_same` = `False`), this message belongs to the single case. To save operational costs, we integrate this logic with the swapping process in Step 3.2 (Lines 27-28). This finishes the identification part. In this step, the server has obtained information about the relationship of the private labels in an ordered sequence and determined whether the label pattern falls into an irregular case. Next, the Boomerang server will swap the regular messages and patch the irregular ones.

Swapping and patching. For regular patterns, the server swaps the sender IDs of the two messages and assigns their values to the receiver ID fields (Lines 27-28). Specifically, if the current label is the same as that of the previous (next) packet, we assign its receiver ID field to have the value of the sender ID of its previous (next) packet. The oblivious choose function helps us conditionally assign the sender ID values to the right packets. It ensures that the server will not learn which part is actually copied to the receiver ID field.

For irregular patterns, Boomerang proactively patches them by looping back such "single" and "more-than-two" messages to its sender (like a boomerang), i.e., setting `pkt.R` = `pkt.S`, where `R` and `S` denote the identifiers of the receiver and sender, respectively. For single cases, the server obliviously assigns `pkt.R` to have the value of `pkt.S`, if the conditions of `is_next_same` and `is_prev_same` are both `False` (Lines 27-28). Finally, for more-than-two cases, where `is_mtt` = `True`, the server also loops back the packet (Line 29). Note that the original messages do not explicitly carry the information of the receivers. This makes sure that only expected "collisions" on private labels can push forward message delivery. In other words, the server will not obtain the receiver IDs from messages with irregular label patterns.

This step patches the receiving pattern of idle clients, active clients with blocked buddies, and clients controlled by an attacker. With Boomerang, messages with irregular label patterns will not influence their receiver's receiving pattern. For the example in Figure 1, the message from E to D is blocked, in which case D would not receive any message if there were no pattern patching. The Boomerang server loops back D's message by setting D as the receiver (as shown in Step ❷), defeating active attacks.

3.3.3 Oblivious Re-order

After the proactive patching, Boomerang needs to re-order the label-ordered sequence. This step is necessary, because adjacent messages in the label-ordered sequence will imply a high possibility that they share the same private label, indicating that the receivers are talking to each other. Hence, we choose to use `osort` with the order of receiver identifiers (IDs) to obliviously re-order the sequence.

Sorting the outgoing messages by receiver IDs would reveal the set of receiver IDs (along with their order), which are essential fields to be reported to the server for message transmission anyway. Recall that in the proactive patching step, the server carefully assigns the values of sender IDs to receiver IDs via swapping and patching. This ensures that the receiver set is exactly the same as the sender set (publicly observable to attackers), and each receiver will receive exactly one message. Therefore, sorting the messages by receiver IDs, which reveals the ordered receiver ID list, would not reveal additional information about the conversation metadata.

Remark. This completes our round-based oblivious message exchange in Boomerang design. Note that at the beginning of each round, we can further employ a textbook deduplication procedure in enclaves to filter repeated packets, just in case a malicious host might try to cause an exchange failure

(DoS attack) by injecting replicated packets from the network stack to let the packets' private labels appear more than twice. Similar treatment has also been done in prior art [32, 56, 57].

4 Boomerang+: Horizontal Scalability

4.1 Overview

As noted in §1, for horizontal scalability, directly replicating the single-server Boomerang, with each server processing a subset of clients, would not work because it immediately implies that clients connecting to different servers are not able to talk to each other. To address the problem, one direct idea is to employ an entry node as a load balancer to obliviously distribute batches of messages from all clients to a group of Boomerang nodes for message exchange. Here, the oblivious design is supposed to hide the mapping between the messages and the Boomerang nodes from an attacker. While this opens up a possible pathway to scale Boomerang, subtle issues remain: how to set up the batch structure?

Recent studies on distributed oblivious data stores [32, 89] have pushed forward the understanding on security-aware scaling designs. Particularly, an oblivious load balancer must set up the batch structure only using public information, independent of the input distribution. In this way, an attacker would not observe any sensitive information from the load balancing. We note that these requirements for setting up an oblivious load balancer are generic to any security-aware scalable system designs. Yet, answering them can be quite design specific. Indeed, a batch structure of data requests generated by an oblivious balancer for a distributed data store, where data partitions are fixed at each server across all rounds [32], would be ill-suited for obliviously distributing batches of messages in Boomerang+, where messages might be mapped to different Boomerang nodes each round.

Setting a batch size for Boomerang+. This has motivated us to search for solutions specific to our Boomerang+ design. In our context, the public information at an entry node is the m messages from m connected clients, and n Boomerang nodes (the back-end nodes for message exchange). Functionality-wise, we need to partition the messages by their private labels, which are random (§4.1), to facilitate the exchange of messages sharing the same labels at the same servers. For security, we need to ensure that the batch structures reveal nothing about the input distribution. For performance, we must set the batch size B as small as possible, but without triggering the overflow (otherwise, there will be dropped messages).

Inspired by the balls-into-bins analysis [32], our problem of finding the bound on batch size B in distributing m messages by their random labels (balls) to n servers (bins) can be translated to: what is the maximum load of balls into any bin? In §3.3.2, we have shown each message's label pattern as single, double, or more-than-two. This suggests that each message carries a different weight in the distribution, and

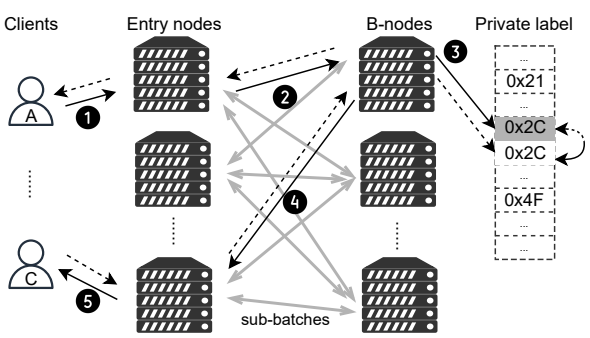


Figure 4: Overview of the scalable Boomerang+ instantiation.

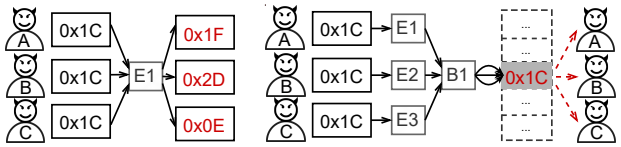
thus we need to answer the question in a *weighted* balls-into-bins game. In Appendix B, we show the complete analysis and proofs to derive the maximum batch size in Boomerang+ (results listed below for easy reference), by applying classic results [13, 76] from the balls-into-bins literature to our problem context.

Theorem 4.1 *For any set of m messages, n Boomerang nodes, and a security parameter λ, satisfying $m \gg n(\ln n)^3$ and $\lambda/\log_2 n > 1$, let $B(m,n)$ be a function that outputs the maximum batch size B for each node in Boomerang+. Then the probability of overflow is negligible in λ if we choose*

$$B = \left\lceil \frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}\left(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)} \right\rceil .$$

Based on the formula, the maximum batch size will be dominated by m/n, with the extra padding size per batch varying with different choices of messages m and Boomerang nodes n. Note that $\lambda/\log_2 n > 1$ can always hold in practice because n is always smaller than 2^λ for any reasonable security parameter λ. As shown in Appendix B, Figure 13, with $\lambda = 128$ and $m = 2^{16}$, when we scale the number of Boomerang nodes n from 4 to 28, the ratio of extra paddings over real messages ranges from 2% to 8%. With the maximum batch size settled, we will describe our oblivious "load balancers" (entry nodes) and the architecture of Boomerang+ next.

Architecture. Figure 4 shows Boomerang+. We leverage the classic two-layer architecture, consisting of entry nodes and Boomerang nodes (B-node for short) to share traffic and computation workload. The message transmission flow is summarized as follows. (❶) Each client connects to one entry node. (❷) Entry nodes generate oblivious sub-batches for B-nodes (§4.2). (❸) Each B-node merges the sub-batches from entry nodes and processes messages like a single Boomerang server (e.g., proactive irregular pattern detection and patching, §3), except for one additional step to swap the entry node identifiers of the pairs (§4.3). (❹) Upon done with the processing, B-node sends the swapped messages back to entry nodes. (❺) Finally, entry nodes merge the sub-batches, pad for possible lost messages, and send them back to receivers.



(a) Case 1 (b) Case 2

Figure 5: An illustration of more-than-two label patterns on (a) entry nodes and (b) B-nodes. The red dotted arrows in (b) imply that the messages to "0x1C" will be looped back to the malicious clients ultimately.

4.2 Entry Nodes

4.2.1 Irregular Pattern Patching

Similar to the basic single-server Boomerang, when receiving a batch, the entry nodes first decrypt and obliviously sort the batch by `priv_label`, and then detect and patch irregular label patterns. But different from the proactive pattern patching algorithm at the basic Boomerang (§3.3.2), here the entry nodes only need to handle the more-than-two cases and leave the single ones to B-nodes.

Patching the more-than-two cases on entry nodes, in case of multiple clients controlled by an attacker sending multiple messages carrying the same private label, is essential. Because such a threat will cause workload skew [32] and potentially influence the non-overflow guarantee of our weighted balls-into-bins algorithm. To eliminate the skew, we let the entry node replace the redundant private labels with new random labels. To detect the more-than-two label patterns, entry nodes adopt the detection algorithm in Boomerang (Step 3.1 in Figure 3, Lines 17-25). Next, instead of looping the irregular messages back, the entry node reassigns new random private labels to them, as shown in Figure 5(a). To set it obliviously, we assign `pkt.priv_label` to have the value of `O_Choose(is_mtt, random(), pkt.priv_label)`.

Since the fresh private label is randomly chosen with negligible possibility of collision, the irregular messages will be regarded as a single pattern to be looped back in subsequent processes on the Boomerang node. In this way, we can eliminate workload skew without revealing the number of malicious messages or changing the communication pattern.

4.2.2 One-time Message Assignment

The message assignment function has two main goals:

- (function goal) assigning the messages with the same private labels to the same B-node, no matter which entry nodes the clients are connecting to, and
- (security goal) ensuring the assignment is (pseudo)random, and the distribution of the sub-batches across rounds will not leak the communication pattern.

For the function goal, the intuition is to derive B-node ID

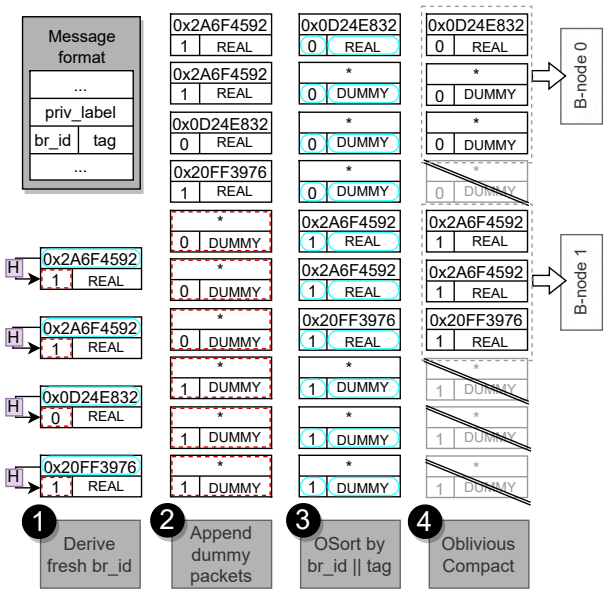


Figure 6: An example of entry node operations. "H" in Step 1 refers to the mapping function (Eq. (1)). B is set to 3. The blue box with rounded corners refers to read operations, and the red box with dotted lines refers to write operations.

(denoted as `br_id`) from the private label using the same deterministic function across all entry nodes. For example, we can simply compute the identifier from the private label modulo the number of B-nodes: $\texttt{br_id} = \texttt{priv_label}\%n$, where n refers to the number of B-nodes. For the security goal, we make the mapping function change across rounds.

Specifically, we use a keyed hash function $\mathsf{H}_k(\cdot)$ to derive a fresh string from the private label, round number, and a secret key k shared among the enclaves of all entry nodes. We apply the modulo function to the fresh string and the number of B-nodes. The assigned B-node is:

$$\texttt{br_id} = \mathsf{H}_k(\texttt{priv_label}||\texttt{round_num})\%n. \qquad (1)$$

The keyed hash function provides a fresh mapping from the private label to the B-node every round, so attackers cannot predict any future assignments in new rounds.

4.2.3 Oblivious Sub-batch Padding

Although the distribution is fresh across rounds, the true size of each sub-batch is revealed, which brings security concerns in long-term communications [32, 52, 87]. Based on our derived maximum batch size in §4.1, we opt to pad the sub-batches to equal size B, which is calculated from public information, namely m messages and n B-nodes, and will not carry any private information about the content.

The remaining step is to obliviously pad the sub-batches without leaking the original sizes. We here follow the oblivious padding algorithm in Snoopy [32]. The padding steps are

shown in Figure 6 and described as follows. Firstly (❷), the entry node appends B dummy messages (with `tag` = `DUMMY`) for each sub-batch to the message sequence. The private labels in dummy messages are randomly generated. Secondly (❸), it obliviously sorts the message sequence by `br_id||tag`. After the sorting, we can get a group of sub-batches, each of which is formed of real messages followed by B dummy messages. Finally (❹), it squeezes extra dummy messages using oblivious compaction. An oblivious compaction algorithm [40] (`O_Compact(flag,array)`) can remove the items in an array with certain flags without leaking which items are removed. The complexity is $O(n \log n)$.

To decide which messages to send or remove obliviously (that is, to set the `flag` for each message), the entry node linearly scans the sorted batch and keeps a counter (`c`) to record the relative position of the message in a sub-batch. If the message is among the first B messages in its sub-batch ($c \leq B$), the message should be sent (`flag = True`). Otherwise, the flag should be set to `False`. We next introduce how to obliviously iterate `c` and assign `flag`. The counter is initially set to 1. When iterating through the batch, the node accumulates the counter ($c+1$) if the current `br_id` repeats its previous one. Otherwise, set the counter to 1, meaning that the node has finished processing the current sub-batch and encounters the first message for a new sub-batch. With the right counter c, the node can obliviously assign `flag` accordingly. The pseudocode for the above step is as follows.

```
is_br_same = O_Equal(pkt.br_id, Prev(pkt).br_id)
c = O_Choose(is_br_same, c+1, 1)
flag = O_Less(c, B)
```

B is the upper bound calculated from our weighted balls-into-bins algorithm, which guarantees that real messages assigned to the same B-node will not exceed size B (except with negligible probability). Therefore, all real messages will be marked with `flag` = `True` and not be dropped. Finally, the node uses oblivious compaction to remove extra dummy messages (those marked with `flag` = `False`).

With the steps above, the size of the sub-batch for each group is exactly B, and the attacker cannot differentiate the dummy messages from the real ones. The entry nodes then send sub-batches to B-nodes for further processing.

4.3 Boomerang Node

Like the basic Boomerang, B-nodes also need to: 1) detect irregular access patterns and patch them (i.e., the patching algorithm in §3.3.2); and 2) swap the messages carrying the same private label. Figure 7 shows operations on B-nodes, among which most operations are the same as Boomerang.

Similarly, there are two types of irregular label patterns on B-nodes: 1) single pattern and 2) more-than-two pattern. Besides idle clients and incomplete pairs as discussed before, single patterns may also come from the irregular messages

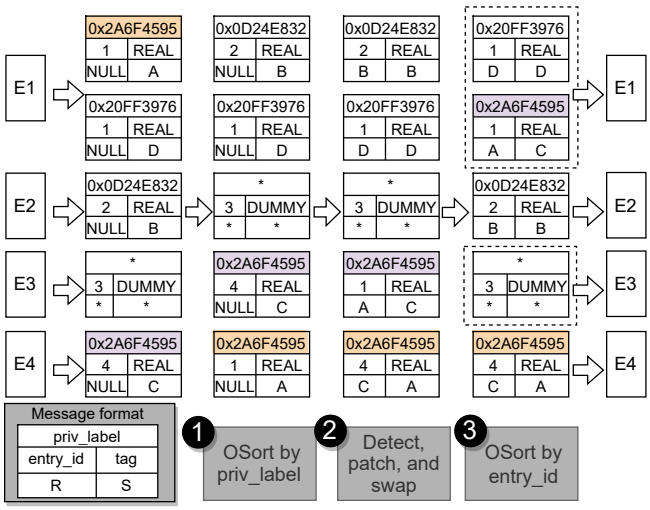


Figure 7: An example of B-node operations. In this example, A and C are talking to each other, B talks to herself (idle client), and the message from someone to D is blocked.

with private labels reassigned by entry nodes. Interestingly, although entry nodes have eliminated workload skew, more-than-two patterns might still appear on B-nodes. As shown in Figure 5(b), malicious clients could send messages with the same private label through multiple entry nodes, with only one message to each entry node, evading the irregular pattern detection. Ultimately, these messages will end at some B-node and appear as more-than-two patterns. To detect such irregular patterns, B-nodes follow the same detection and patching algorithm as Boomerang (Steps 2 and 3 in Figure 3).

Different from Boomerang, B-nodes cannot directly send the processed messages to clients, because this reveals the identities of clients connecting to the same B-node, indicating they are more likely to be talking. Instead, we let B-nodes send each message back to the entry node first, and then let the entry node send the message back to the receiver connecting to it. The entry node ID of the receiver can be obtained from the paired message (if any) with which it was swapped. Hence, (❷) the B-node can swap entry node IDs (`entry_id`) of the paired messages during the linear scan. Similar to the receiver ID swapping step in Boomerang, the B-node swaps (or loops back) both the entry node IDs and receiver IDs using oblivious choose. Finally (❸), the B-node obliviously sorts the sequence by entry node IDs (to group the batch for each entry node together) and sends the messages back to the corresponding entry nodes.

In Boomerang, the last step is to re-order messages. We accordingly move this step to entry nodes. When receiving the sub-batches from B-nodes, entry nodes merge the sub-batches, sort the batch by receiver IDs, pad for possible missing messages (as we will discuss in §4.4), remove extra dummy messages, and send the messages to receivers.

4.4 Server Churn

So far, we have introduced our mechanisms (e.g., one-time message assignment and oblivious sub-batch padding) to make the communication pattern among servers oblivious to attackers. However, such mechanisms only consider the "accidents" from clients. For completeness, here we discuss some possible accidents that may happen to servers and introduce Boomerang+'s enhancing strategies.

B-node churn. B-nodes can be blocked (e.g., under DoS attacks) or accidentally go offline without sending back the messages, leading to missing patterns on the clients whose messages are processed on the corresponding B-node. This gives an attacker a chance to link the clients with certain B-nodes. Towards this threat, we let the entry node proactively patch the dropping pattern due to the lost connection with some B-nodes. The patching algorithm is very similar to the oblivious sub-batch padding algorithm in §4.2.3. Firstly, the entry node appends one dummy message for each client connecting to itself. This is feasible because the node can record the list of connecting clients when it processes the incoming messages in the very beginning. Secondly, the entry node obliviously sorts the batch by receiver identifiers. Finally, it linearly scans the sequence, marks which messages to send, and obliviously compacts the batch. After the patching, we can ensure that all clients connecting to the entry node will receive one message in each round, no matter whether there is any B-node churn or not. For efficiency, we only trigger this patching algorithm when a connection loss occurs. Once a B-node goes offline, entry nodes will ignore this node and not assign messages to it in the next round (according to the instructions from the coordinator). In this way, we make sure that B-node churn will not cause severe service denial.

Entry node churn. This case is similar to the case where the clients are blocked, but on a larger scale. The direct impact is that the buddies of clients connected to this entry node will fail to form a paired private label, and thus their messages will be looped back (by other entry nodes). In this case, attackers observe nothing but a predictable situation where a set of clients cannot link to the churned entry node.

5 Analysis

We now show Boomerang and Boomerang+ achieve communication pattern indistinguishability as we claimed in §2. Thanks to the oblivious designs, we ensure that all connected clients behave the same in every round, either exchanging messages with a buddy or sending messages to themselves.

Theorem 5.1 *Given oblivious comparison/assignment operations, an oblivious sort algorithm, and an oblivious patching and swapping algorithm, Boomerang achieves communication pattern indistinguishability presented in Definition 2.1.*

Proof sketch. We focus on the setting defined in Definition 2.1, where $m-2$ clients are compromised by the attacker. As Boomerang operates in rounds, in view of the attacker, the only way to distinguish the communication pattern of two given clients is from the observation of memory access patterns during the "obfuscated" message exchange procedure. Thus, the security of Boomerang hinges on the oblivious algorithms designed for proactive pattern patching and re-order (in §3.3.2 and §3.3.3). As the re-order algorithm is essentially the oblivious sorting [10], we mainly pay attention to proving the obliviousness of proactive pattern patching, captured by Lemma C.1 in Appendix C.1. When all involved algorithms are oblivious, it is easy to derive that the attacker cannot identify whether two clients are communicating or not within each round, except with negligible probability. For the fixed system configuration, it follows that the attacker's view will remain the same across rounds. The full proof of the Theorem 5.1 can be seen in Appendix C.1.

We next show that Boomerang+ achieves the same communication pattern indistinguishability as Boomerang, even though multiple servers are introduced.

Theorem 5.2 *Given a cryptographic hash function, oblivious comparison/assignment operations, an oblivious sort algorithm, an oblivious patching and swapping algorithm, an oblivious sub-batch padding algorithm, and an oblivious compaction algorithm, Boomerang+ achieves communication pattern indistinguishability presented in Definition 2.1.*

Proof sketch. Following the proof sketch in Theorem 5.1, here we also focus on the settings where $m-2$ clients are controlled by the attacker. With multiple entry nodes and B-nodes deployed, we will first show that Boomerang+ assigns a message to each B-node uniformly due to the deployment of the cryptographic hash function and our security-aware load-balancing design. Then, we show that for the two targeted clients, whether they are communicating or not, the probabilities that their messages are assigned to one single B-node are always equal. Thus, combined with Theorem 5.1, we have that an attacker cannot identify whether two clients are communicating or not in Boomerang+. The detailed analysis can be seen in Appendix C.2.

It is intuitive to see that Boomerang+ is horizontally scalable. See Appendix B for the scalability analysis.

6 Implementation and Evaluation

We implement Boomerang and Boomerang+ in about 4000 lines of C++ code. We build secure enclaves using the framework of Intel SGX v2.16 on Intel SGX DCAP Driver v1.14 and use Intel's AVX-512 SIMD instructions for basic oblivious primitives. We build our oblivious algorithms using the oblivious library from XGBoost [2]. Clients and Boomerang(+) servers communicate using gRPC v1.35 on asynchronous RPC mode over TLS. The Boomerang(+) prototype is available online at https://github.com/CongGroup/boomerang.

6.1 Evaluation Overview

We experimentally answer the following questions: 1) How fast are Boomerang and Boomerang+? 2) Can Boomerang+ scale by adding servers? We highlight some results below:

- Boomerang achieves 99th percentile latency of 1.41 second for 2^{16} clients on one 16-core server. For space, we present the results in Appendix D.
- Boomerang+ achieves 99th percentile latency of 615 ms for 2^{16} on 16 servers and 7.76 second latency for 2^{20} clients on 32 servers, respectively.

Experiment setup. We evaluate Boomerang(+) on Tencent Cloud M6ce VMs [83], with Intel Xeon Ice Lake processors with Intel SGX support [45]. For Boomerang, we use one `M6ce.4XLARGE128` instance (16 vCPU, 128 GB of memory, and 13 Gbps of network bandwidth). For Boomerang+, we assign 12 `M6ce.4XLARGE128` instances as entry nodes and 4 `M6ce.4XLARGE128` instances as B-nodes by default. For completeness, we also evaluate three metadata-private communication systems: Pung (XPIR) [8], XRD [54] and Addra [4]. According to Azure pricing [65], instances with TEEs are roughly twice as expensive as those without TEEs at the same level of computing power. Therefore, to compensate for the machine cost of the trusted hardware Boomerang uses, we allocate twice the number of machines (or total CPU cores) for these systems. For XRD, we use 32 `M6.4XLARGE128` instances (16 vCPU, 128 GB of memory, and 13 Gbps of network bandwidth) to construct the chain-based architecture. For Pung, we use one `M6.4XLARGE128` instance and run 1/32 total traffic over it, following the setting in XRD [54]. Since Pung is directly parallelizable, letting one server share 1/N of the total traffic is the best performance it can possibly achieve. For Addra, we use one `S4.8XLARGE128` instance (32 vCPU, 128 GB of memory, and 11 Gbps of network bandwidth) as the master and 30 `M6.4XLARGE128` instances as the workers. For clients in the four systems, we use one `C5.26XLARGE368` instance (104 vCPU, 368 GB of memory, 36 Gbps of network bandwidth) to run simulated clients, each sending/receiving one RPC request at a time. We run instances in the same data center to save bandwidth and simulate client-server round trip latency of 100 ms by using Linux `tc` command. Results are averaged 20 times for each experiment.

Parameters. We set the message size to 256 Bytes and the private label to 256 bits. We set the conversation round to 12 seconds, according to the latency results from our experiment on Boomerang+ dealing with 2^{20} messages. We set the batch size B according to Theorem 4.1, with the security parameter $\lambda = 128$. For XRD, we construct 32 chains, each of which consists of 30 machines. The length ensures that the probability of the existence of a group of malicious servers is less than 2^{-64} if 20% of the servers are malicious. We let each client send 8 messages to 8 chains, following XRD's recommendations. For Pung, we use recursion with a depth of 2 and set the bucket size to 64.

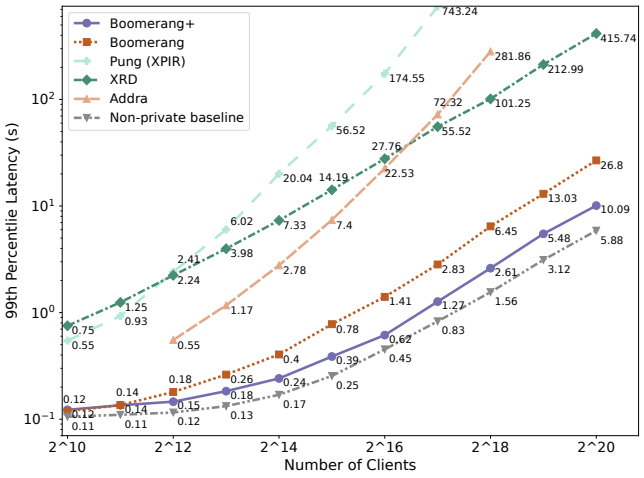


Figure 8: 99th percentile latency of Boomerang, Boomerang+, and other baselines with a varying number of clients.

6.2 Boomerang+ Performance

Latency and throughput. Figure 8 depicts the latency of Boomerang, Boomerang+, Pung (XPIR) [8], XRD [54], Addra [4], and a non-private baseline. Pung, XRD, and Addra are the latest work achieving pairwise metadata-private communication under cryptographic security, but with different trust assumptions. XRD operates on fractional trusted servers, and Addra and Pung operate on fully untrusted servers. We notice that the latency results of Addra and XRD are higher than those reported in their papers. This is perhaps because we used fewer and less powerful machines. Besides, we only ran Addra over up to 2^{18} clients, as our testbed (one 32-core master and 30 16-core workers) could not afford a workload for clients more than 2^{18}.

Boomerang+ achieves 615 ms latency for 2^{16} clients and 10.09 second latency for 2^{20} clients. The throughput of Boomerang+ reaches 85.4K messages per second under 16 machines. Compared to the prior systems based on cryptographic primitives, Boomerang+ is significantly more efficient thanks to leveraging the trusted hardware. For example, for 2^{16} clients, Boomerang+ is 36× faster than Addra and 45× faster than XRD. We also evaluate a non-private version of Boomerang+ to show the cost of non-oblivious operations (most of which is from the network operation cost). In this baseline, we keep the same round-based design and traffic transmission flow (i.e., the same two-layer network architecture) but remove all security-enhancing operations in enclaves (e.g., oblivious sort, padding, patching, etc.). Results (the grey dotted line in Figure 8) indicate that the computational overhead of the oblivious operations for metadata hiding over that of other basic operations is small. Interestingly, compared to Boomerang (the basic instantiation), Boomerang+ does not have an overwhelming advantage when the clients are less than 2^{15}. This is because Boomerang+ involves more obliv-

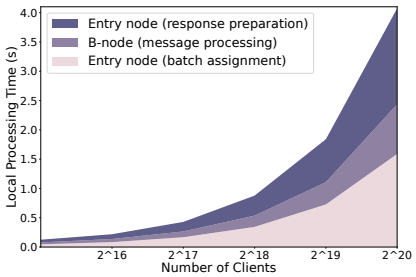


Figure 9: Breakdown of Boomerang+ operation costs.

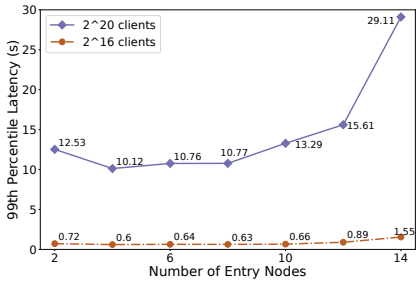


Figure 10: Latency with varying numbers of entry and B-nodes (sum fixed to 16).

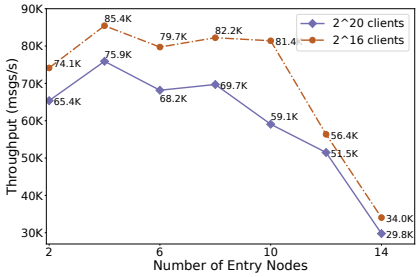


Figure 11: Throughput with varying numbers of entry nodes and B-nodes.

ious operations (e.g., the padding, oblivious sort, and compaction on the entry nodes) to take care of potential threats due to the scalable design. When the clients scale to 2^{20}, the benefits of horizontal scaling gradually show up: Boomerang+ runs 2.66× faster than Boomerang. In §6.3, we will further show the horizontal scalability of Boomerang+; that is, it can achieve lower latency by adding more servers.

Bandwidth cost. Boomerang+ requires one message per round for both active and idle clients. On the server side, Boomerang+ involves dummy paddings, and the overhead is relatively small. According to our calculation, for a workload of 2^{15} messages on one entry node to four B-nodes, dummy messages account for less than 4% of all messages (details about the padding overhead in Figures 13 and 14 in Appendix B). Luckily, on Azure Cloud and many other clouds, data transfer within one VPNet is free [63].

6.3 Microbenchmarks

Breakdown of Boomerang+ operational cost. To evaluate the computational cost on entry nodes and B-nodes, we break down Boomerang+ latency into three parts: 1) entry node batch assignment, 2) B-node message processing, and 3) entry node response preparation. Figure 9 shows the processing time for 8 entry nodes plus 8 B-nodes, each of which handles (almost) an equivalent volume of messages. Besides, we only record the local processing time on each node and eliminate the network cost (e.g., RPC request and response with clients) here. In this way, we can clearly see the computational overhead at each stage. Note that the total operation latency is smaller than the end-to-end latency presented in Figure 8. This is because processing RPC requests is time-consuming in our implementation, especially under a large number of requests. Generally, facing the same volume, the entry nodes (both the first and third stages) operate longer than B-nodes. We conjecture that assigning a few more machines as entry nodes than B-nodes with a fixed total number of machines may save latency, as shown below.

Resource allocation on entry nodes and B-nodes. As discussed before, resource allocation can be important for Boomerang+'s overall performance. We have tested the latency and throughput under different allocations on entry nodes and B-nodes using 16 machines in total, as shown in Figures 10 and 11, respectively. We find that Boomerang+ performs the best over 12 entry nodes and 4 B-nodes. This is because the operational cost is higher on entry nodes, and assigning more entry nodes reduces the average workload on each entry node. This observation is consistent with the breakdown results, where processing on entry nodes takes longer than that on B-nodes. We also report the performance of different allocations given different total numbers of servers in Appendix E.

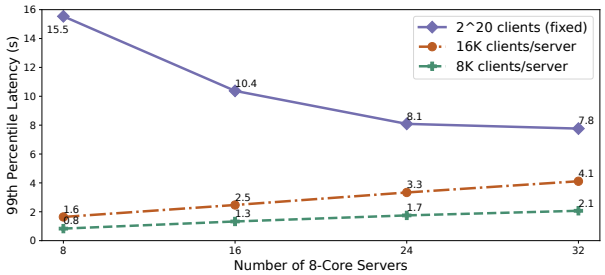


Figure 12: Latency with a varying number of servers.

Scaling by adding servers. With horizontal scalability, Boomerang+ should, ideally, achieve the same level of latency by adding more servers when handling more clients. In this experiment, we adopt 32 8-core `M6ce.2XLARGE64` instances to test the scalability of Boomerang+ over more servers.

We first set a fixed total number of clients to 2^{20} and gradually increase the number of servers from 8 to 32 to see how the latency can be reduced by adding servers. We then evaluate the scalability by proportionally adding servers and clients, by keeping a fixed number of clients on one server group. Specifically, we let every 8 servers handle 2^{17} and 2^{16} clients (amortized to about 16K and 8K clients per server, respectively). For different numbers of servers, we choose the best ratio of entry nodes to B-nodes according to the supplementary resource allocation experiment (Appendix E). The ratio is 5:3 for 8 servers and 3:1 for 16, 24, and 32 servers. As reported in Figure 12, the latency reduces from 15.5 seconds to 7.8 seconds when Boomerang+ scales from 8 to 32 servers. Adding more servers to decrease the amortized workload helps re-

duce latency when client size scales, but this also increases the system dollar cost (as discussed in Appendix F). This is a trade-off between performance and cost. For example, with the same total number of clients, supporting an average of 8K clients per server doubles the machine cost of supporting 16K clients per server, while gaining a 2× speed-up. Fortunately, the host cost for a 16-core SGX instance on Azure cloud is only 1121 USD/month, which we believe is affordable when amortized to 8K clients (0.14 USD/month per client).

7 Related Work

Mix-nets and follow-up enhancements. There has been significant progress in metadata-private communication designs recently. One category of results follows the concept of mix networks (mix-nets) [21]. Based on that, recent results [15,16,52–54,56–58,72,87,88] have proposed a number of noteworthy security and privacy enhancement strategies to deal with powerful attackers, including: 1) batch processing of messages from all clients in synchronous rounds to mitigate traffic analysis threats [31, 39]; 2) carefully structuring protocols to reveal less observable variables (and thus exposing reduced useful information) to attackers, with representative examples of adopting private virtual addresses for obfuscated message exchanges [54, 56, 87, 88]; 3) generating cover traffic with calibrated parameters to obscure the communication patterns among users (as well as users) [56, 87, 88]; 4) adding verifiability to the message shuffling to defeat misbehaving servers among the mix-nets [54, 56, 87]; and 5) designing proactive self-recovering schemes (usually relying on the assumption of honest servers) in the face of inadvertent user disconnections or even active network disruptions [54, 57]. One latest effort has tried to shift the online burden of clients [11] through oblivious delegation to untrusted proxy servers.

Boomerang's design draws insights from these strategies but differs from them on instantiations with hardware enclaves. Trusting the enclaves enables Boomerang to use fewer servers for traffic mixing with reduced latency. But the unique enclave security context demands Boomerang to bring together tailored oblivious designs for message shuffling, horizontal scaling, and proactive patching against active attacks.

Metadata-private messaging via cryptographic designs. There have been cryptographic designs to facilitate metadata-private communications. One category of results follows the dining cryptographers network (DC-net) [22], which demands broadcasting data linear to the size of the participating clients in the anonymous communication at a high level [28]. Later systems [3, 29, 91] propose scalability improvement and resilience designs against unreliable clients and untrusted servers. Another category of cryptographic designs utilizes private information retrieval [4, 6, 8], MPC [5] and distributed point function techniques [3, 27, 37, 70] to facilitate oblivious read / write to a database with private mailboxes, based on which metadata-private messaging (and broadcast) system can be constructed. Recently, Clarion [36] gives an MPC-based shuffling design for anonymous communication.

Despite providing cryptographic security (sometimes even under fully untrusted server [4, 8]), these systems do not easily scale to more than hundreds of thousands of users while maintaining low latency and high throughput. The inherent cryptographic operations also present unfavorable operational dollar cost, which might yet be attractive for voluntary adoptions in practice.

Security-aware scaling of oblivious data stores. Recent results have studied how to scale the oblivious data access systems without leaking information about the data requests [32, 89]. The key is an oblivious load balancer design that distributes access batches independent of the input distribution (with security-aware paddings) and sets the batch size using only public information. The entry-node design in Boomerang+ is inspired by these generic observations. Yet, with context-specific modeling and analysis, we have derived the bound of the maximum batch size that best suits our metadata-private messaging system, through a weighted balls-into-bins game.

Enclave-based network systems. While enclave-based network applications are many, e.g., SGX-Tor [49, 50], SGX-middleboxes [35, 42, 74], to our best knowledge, usage of enclaves is rarely attempted for metadata-private communications, except for one recent work DAEnet [80]. The focus of DAEnet is different from Boomerang, as it tends to hide the conversation route through a peer-to-peer infrastructure, where each peer as a personal computer is assumed to be equipped with an enclave. Unfortunately, this assumption seems no longer in line with the industry movement [44].

8 Conclusion

We have presented Boomerang, a metadata-private messaging system that leverages the readily available trust assumption on hardware enclaves. Boomerang draws many insights from the prior art and achieves efficient pairwise communication with cryptographic security. Its technical instantiation involves tailored oblivious algorithms for message shuffling, horizontal scaling, and proactive resistance against active attacks. We hope Boomerang's comparably high efficiency and low operational cost could make metadata-private messaging systems one step closer to mass adoption in practice.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Dr. Jay Lorch, for their helpful and valuable feedback, and Tencent Yunding lab for providing the Intel SGX cluster and generous technical support. This work was partially supported by the NSFC under Grants U20B2049, U21B2018, 62202228, and 62032021, the HK RGC under Grants N_CityU139/21, RFS2122-1S04, C2004-21GF, R1012-21, and R6021-20F, and the Natural Science Foundation of Jiangsu Province under Grant BK20210330.

References

[1] Confidential Computing Zoo repository. https://github.com/intel/confidential-computing-zoo. Accessed Sept. 2022.

[2] XGBoost repository. https://github.com/mc2-project/secure-xgboost. Accessed Sept. 2022.

[3] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder - scalable, robust anonymous committed broadcast. In *Proc. of ACM CCS*, pages 1233–1252, 2020.

[4] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *Proc. of USENIX OSDI*, 2021.

[5] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *Proc. of USENIX Security*, pages 1217–1234, 2017.

[6] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *Proc. of IEEE S&P*, pages 962–979, 2018.

[7] Sebastian Angel, Sampath Kannan, and Zachary B. Ratliff. Private resource allocators and their applications. In *Proc. of IEEE S&P*, pages 372–391, 2020.

[8] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *Proc. of USENIX OSDI*, pages 551–569, 2016.

[9] Apache. Mutual Attestation: Why and How. https://teaclave.apache.org/docs/mutual-attestation/. Accessed Jan. 2023.

[10] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *Proc. of SOSA*, pages 8–14, 2020.

[11] Ludovic Barman, Moshe Kol, David Lazar, Yossi Gilad, and Nickolai Zeldovich. Groove: Flexible metadata-private messaging. In *Proc. of USENIX OSDI*, pages 735–750, 2022.

[12] Kenneth E. Batcher. Sorting networks and their applications. In *Proc. of AFIPS*, volume 32, pages 307–314, 1968.

[13] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell A. Martin. On weighted balls-into-bins games. *Theor. Comput. Sci.*, 409(3):511–520, 2008.

[14] Oliver Berthold and Heinrich Langos. Dummy traffic against long term intersection attacks. In *Proc. of PETS*, pages 110–128, 2002.

[15] Stevens Le Blond, David R. Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proc. of ACM SIGCOMM*, 2015.

[16] Stevens Le Blond, David R. Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. In *Proc. of ACM SIGCOMM*, 2013.

[17] Alexandra Boldyreva, David Cash, Marc Fischlin, and Bogdan Warinschi. Foundations of non-malleable hash and one-way functions. In *Proc. of ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 524–541, 2009.

[18] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor. In *Proc. of MICRO*, pages 42–56, 2019.

[19] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proc. of WOOT*, 2017.

[20] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proc. of USENIX Security*, pages 1041–1056, 2017.

[21] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[22] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.

[23] Guoxing Chen and Yinqian Zhang. Mage: Mutual attestation for a group of enclaves without trusted third parties. In *Proc. of USENIX Security*, pages 4095–4110, 2022.

[24] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. In *Proc. of NDSS*, 2020.

[25] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *Proc. of USENIX Security*, pages 699–716, 2021.

[26] David Core. We kill people based on metadata. *The New York Review*, 2014.

[27] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proc. of IEEE S&P*, pages 321–338, 2015.

[28] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *Proc. of ACM CCS*, pages 340–350, 2010.

[29] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *Proc. of USENIX Security*, pages 147–162, 2013.

[30] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proc. of USENIX Security*, pages 857–874, 2016.

[31] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *Proc. of IEEE S&P*, pages 108–126, 2018.

[32] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proc. of ACM SOSP*, pages 655–671, 2021.

[33] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect. In *Proc. of WEIS*, 2006.

[34] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *Proc. of USENIX Security*, pages 303–320, 2004.

[35] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. LightBox: Full-stack protected stateful middlebox at lightning speed. In *Proc. of ACM CCS*, pages 2351–2367, 2019.

[36] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *Proc. of NDSS*, 2022.

[37] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *Proc. of USENIX Security*, pages 1775–1792, 2021.

[38] Electronic Frontier Foundation. Why metadata matters. https://ssd.eff.org/module/why-metadata-matters. Accessed Jan. 2023.

[39] Yossi Gilad. Metadata-private communication for the 99%. *Commun. ACM*, 62(9):86–93, 2019.

[40] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proc. of SPAA*, pages 379–388, 2011.

[41] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *Proc. of USENIX ATC*, pages 299–312, 2017.

[42] Juhyeng Han, Seong Min Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proc. of APNet*, 2017.

[43] Alejandro Hevia and Daniele Micciancio. An indistinguishability-based characterization of anonymous channels. In *Proc. of PETS*, volume 5134, pages 24–43, 2008.

[44] Intel. Intel SDP for desktop based on Alder Lake S - 12th Generation Intel Core Processors. https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/009/. Accessed Feb. 2023.

[45] Intel. Intel Xeon scalable platform built for most sensitive workloads. https://www.intel.com/content/www/us/en/newsroom/news/xeon-scalable-platform-built-sensitive-workloads.html. Accessed Feb. 2023.

[46] Intel. SGX remote attenstation services. https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf. Accessed Sept. 2022.

[47] Bastien Inzaurralde. The cybersecurity 202: Leak charges against treasury official show encrypted apps only as secure as you make them. *The Washington Post*, 2018.

[48] Van Bulck Jo, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *Proc. of IEEE S&P*, pages 54–72, 2020.

[49] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Enhancing security and privacy of Tor's ecosystem by using trusted execution environments. In *Proc. of USENIX NSDI*, pages 145–161, 2017.

[50] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. SGX-Tor: A secure and practical Tor anonymity network with SGX enclaves. *IEEE/ACM Transactions on Networking*, 26(5):2174–2187, 2018.

[51] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *CoRR*, 2018.

[52] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proc. of ACM SOSP*, pages 406–422, 2017.

[53] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. In *Proc. of PETS*, pages 115–134, 2016.

[54] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable messaging system with cryptographic privacy. In *Proc. of USENIX NSDI*, pages 759–776, 2020.

[55] Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. Secure collaborative training and inference for XGBoost. In *Proc. of PPMLP*, pages 21–26, 2020.

[56] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proc. of USENIX OSDI*, pages 711–725, 2018.

[57] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: Strong metadata security for voice calls. In *Proc. of ACM SOSP*, pages 211–224, 2019.

[58] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proc. of USENIX OSDI*, pages 571–586, 2016.

[59] Alleyne Llanor. Enterprise end-to-end encryption is on the rise. *IT Business Edge*, 2021.

[60] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proc. of PETS*, pages 17–34, 2004.

[61] Jonathan R. Mayer, Patrick Mutchler, and John C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proc. Natl. Acad. Sci. USA*, 113(20):5536–5541, 2016.

[62] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proc. of HASP*, page 10, 2013.

[63] Microsoft. Azure bandwidth pricing. https://azure.microsoft.com/en-us/pricing/details/bandwidth/. Accessed Sept. 2022.

[64] Microsoft. Azure cloud messaging services. https://azure.microsoft.com/en-us/solutions/messaging-services/#overview. Accessed Sept. 2022.

[65] Microsoft. Azure virtual machine pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/#pricing. Accessed Sept. 2022.

[66] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proc. of IEEE S&P*, pages 279–296, 2018.

[67] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *Proc. of CHES*, pages 69–90, 2017.

[68] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proc. of IEEE S&P*, pages 1466–1482, 2020.

[69] Netsfere. Netsfere pricing. https://www.netsfere.com/Product/Free-Pro-Custom-Enterprise-Messaging-Pricing. Accessed Sept. 2022.

[70] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *Proc. of USENIX NSDI*, pages 229–248, 2022.

[71] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *Proc. of USENIX Security*, pages 619–636, 2016.

[72] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *Proc. of USENIX Security*, pages 1199–1216, 2017.

[73] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-preserving video analytics as a cloud service. In *Proc. of USENIX Security*, pages 1039–1056, 2020.

[74] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding network functions in the cloud. In *Proc. of USENIX NSDI*, pages 201–216, 2018.

[75] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A secure database using SGX. In *Proc. of IEEE S&P*, pages 264–278, 2018.

[76] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In *Proc. of International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 159–170, 1998.

[77] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Proc. of IEEE S&P*, pages 1852–1867, 2021.

[78] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward confidential cloud computing. *Commun. ACM*, 64(6):54–61, 2021.

[79] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. of IEEE S&P*, pages 38–54, 2015.

[80] Tianxiang Shen, Jianyu Jiang, Yunpeng Jiang, Xusheng Chen, Ji Qi, Shixiong Zhao, Fengwei Zhang, Xiapu Luo, and Heming Cui. DAENet: Making strong anonymity scale in a fully decentralized network. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2286–2303, 2021.

[81] Jatinder Singh, Jennifer Cobbe, Do Le Quoc, and Zahra Tarkhani. Enclaves in the clouds. *Commun. ACM*, 64(5):42–51, 2021.

[82] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *Proc. of IEEE S&P*, pages 253–267, 2013.

[83] Tencent. Tencent Cloud instance type documentation. https://intl.cloud.tencent.com/document/product/213/11518. Accessed Sept. 2022.

[84] Tencent. Tencent cloud instant messaging pricing. https://intl.cloud.tencent.com/products/im. Accessed Sept. 2022.

[85] Tencent. Tencent Cloud virtual machine pricing. https://intl.cloud.tencent.com/pricing/cvm/overview. Accessed Sept. 2022.

[86] Trillian. Trillian instant messaging pricing. https://trillian.im/pricing/. Accessed Sept. 2022.

[87] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proc. of ACM SOSP*, pages 423–440, 2017.

[88] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proc. of ACM SOSP*, pages 137–152, 2015.

[89] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. SHORTSTACK: Distributed, fault-tolerant, oblivious data access. In *Proc. of USENIX OSDI*, pages 719–734, 2022.

[90] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proc. of ACM CCS*, pages 2421–2434, 2017.

[91] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *Proc. of USENIX OSDI*, pages 179–182, 2012.

[92] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. Hang with your buddies to resist intersection attacks. In *Proc. of ACM CCS*, pages 1153–1166, 2013.

A Discussion and Limitations

Establishing trust on Boomerang. Boomerang relies on the trust assumption on secure enclaves, which provide confidentiality and integrity for data and codes. While trusting a single enclave node can be done through the standard remote attestation, like the one from Intel SGX [46], trusting multiple-enclave applications would additionally involve mutual attestation, which is necessary among interacting enclaves to establish a trust relationship [23, 79]. One common practice is to rely on a trusted third party (TTP) to facilitate mutual attestation [9]. The TTP can perform remote attestation with each enclave individually and serve as a trusted anchor to bootstrap the mutual trust among those enclaves. In the case of Boomerang, we suggest following the above common practice for any client connecting to the system to establish trust on Boomerang. The TTP that all clients need to rely on can be the trusted developer of Boomerang or his/her delegated server in a trusted domain [9, 23]. A very recent work has proposed a new way for mutual attestation among enclaves without relying on a TTP [23], which can be beneficial to the trust establishment in the Boomerang system. In the future, to mitigate the concern on the centralized trust of a single enclave vendor (e.g., Intel), we can further consider employing a mix of enclaves from different vendors, e.g., Intel SGX, ARM

TrustZone, AMD SEV, etc., to distribute the trust, which is also a trendy subject in recent literature [23].

Reduce client online burden. Metadata-private messaging systems usually assume clients should always keep online and send messages at regular rates to hide the real communication behavior [4, 33, 39, 54, 57, 87, 88]. To simplify the problem statement, Boomerang's security analysis also follows this assumption. Although Boomerang achieves acceptable bandwidth cost, we acknowledge that this "always-online" requirement may be a barrier to the practical use of Boomerang. The latest work Groove [11] studied this issue and proposed an oblivious delegation mechanism to reduce client online burden, by introducing proxies between clients and mixnets. To improve Boomerang's client flexibility, a feasible way is to integrate the oblivious delegation mechanism and the proxy design with Boomerang, considering that Boomerang can be an alternative to mixnets for message shuffling. We believe our Boomerang can serve as a performant and secure backend for metadata-private message shuffling.

B Balls into Bins

To analyze the overflow probability in Boomerang+, we introduce the balls-into-bins game to estimate the probability that a message may be dropped during batch distributions.

Balls-into-bins studies the allocation problem that throws m balls into n bins by placing each ball into a bin chosen independently and uniformly at random [76]. One natural question in this area is to ask for the maximum number of balls in any bin. According to prior art [13, 76], an interesting result about the maximum number of balls in any bin problem is introduced below, which will be used to estimate the maximum load bound of Boomerang+.

Lemma B.1 (Maximum load [76]) *Let ℓ be the random variable that counts the maximum number of balls in any bin, if we throw m balls independently and uniformly at random into n bins and $m \gg n(\ln n)^3$. Then we have*

$$\Pr[\ell > \frac{m}{n} + \sqrt{\frac{2m\ln n}{n}\left(1 - \frac{1}{\alpha}\frac{\ln\ln n}{2\ln n}\right)}] = 1 - \frac{1}{n^\alpha},$$

where α is a positive constant larger than 1.

Here α serves a role to ensure the tightness of the bound of the maximum load [13, 76]. Next, we prove Theorem 4.1 below.

Theorem 4.1 *For any set of m messages, n Boomerang nodes, and a security parameter λ, satisfying $m \gg n(\ln n)^3$ and $\lambda/\log_2 n > 1$, let $B(m,n)$ be a function that outputs the maximum batch size B for each node in Boomerang+. Then the probability of overflow is negligible in λ if we choose*

$$B = \left\lceil \frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}\left(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)} \right\rceil.$$

Proof. Like most existing works, allocating a set of m messages to a given number of servers can be formalized as a balls-into-bins game. But a bit different from their works, the balls are weighted in Boomerang+ because messages with the same private labels will be allocated to the same servers. Suppose m_1 is the number of messages with single-pattern private labels (see §3.3.2 for possible causes), and m_2 is the number of messages with double-pattern private labels (i.e., regular messages from communicating pairs). Since assigning messages with single-pattern private labels and assigning messages with regular private labels are independent, the task of allocating m messages can be separated as two subtasks: 1) allocating m_1 messages with single-pattern private labels; and 2) allocating m_2 messages with double-pattern private labels. It's clear that $m_1 + m_2 = m$.

For the task of allocating m_1 messages to n B-nodes, it can be formalized as the game throwing m_1 balls into n bins. For the task of allocating m_2 messages, a pair of messages with the same regular private labels will be allocated to the same B-node. Thus, we can tie m_2 balls together in pairs (by their double-pattern labels), and throw $m_2/2$ times. Namely, it is a $m_2/2$-balls-into-n-bins problem. Let ℓ_1 and ℓ_2 be two random variables that count the maximum messages assigned in any B-node in the above two tasks, respectively. Let

$$B_{m_1} = \frac{m_1}{n} + \sqrt{\frac{2m_1\ln n}{n}(1 - \frac{1}{\alpha}\frac{\ln\ln n}{2\ln n})}$$

and

$$B_{m_2} = \frac{m_2}{n} + 2\sqrt{\frac{m_2\ln n}{n}(1 - \frac{1}{\alpha}\frac{\ln\ln n}{2\ln n})}.$$

According to Lemma B.1, we have

$$\Pr[\ell_1 > B_{m_1}] = 1 - \frac{1}{n^\alpha} \text{ and } \Pr[\ell_2 > B_{m_2}] = 1 - \frac{1}{n^\alpha}.$$

To prevent overflow, the probability of a message being dropped should be confined to a negligible function in the security parameter λ. Thus we have

$$1 - \frac{1}{n^\alpha} = 1 - \frac{1}{2^\lambda} \Rightarrow \alpha = \frac{\lambda}{\log_2 n}.$$

Let $B' = B_{m_1} + B_{m_2}$, then

$$B' = \frac{m}{n} + \left(\sqrt{m_1} + \sqrt{2m_2}\right)\sqrt{\frac{2\ln n}{n}\left(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)}. \quad (2)$$

With the arithmetic-geometric mean inequality, we have that

$$\left(\sqrt{m_1} + \sqrt{2m_2}\right) \le \sqrt{2(\sqrt{m_1})^2 + 2(\sqrt{2m_2})^2)},$$

which attains its equality if and only if $\sqrt{m_1} = \sqrt{2m_2}$.

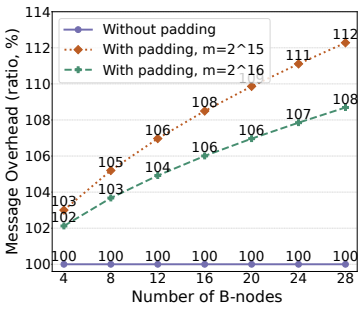


Figure 13: Message overhead (ratio), which describes the ratio of overall padded messages to real messages, i.e., nB/m.

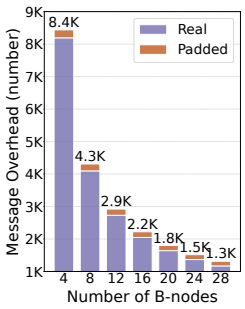


Figure 14: Message overhead (number) under 2^{15} real messages concretely.

Combined with the condition $m_1 + m_2 = m$, we can solve for $m_1 = 2m/3$ and $m_2 = m/3$. Then

$$(\sqrt{m_1} + 2\sqrt{m_2}) \leq \sqrt{2\left[\left(\sqrt{\frac{2m}{3}}\right)^2 + \left(\sqrt{\frac{2m}{3}}\right)^2\right]} \leq 2\sqrt{\frac{2m}{3}}. \tag{3}$$

Applying Eq. (3) to Eq. (2), it is easy to get

$$B' \leq \frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}\left(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)} \leq B.$$

Finally, we compute the probability that a message is dropped on a server. It equals the probability that the maximum number of messages allocated to a node is larger than the maximum load. As mentioned, the above two subtasks are independent, and thus we have

$$\begin{aligned}
\Pr[\ell > B] &\leq \Pr[\ell > B'] = \Pr[\ell > B_{m_1} + B_{m_2}] \\
&\leq 1 - \Pr[\ell \leq B_{m_1} + B_{m_2}] \\
&\leq 1 - \Pr[\ell_1 \leq B_{m_1} \wedge \ell_2 \leq B_{m_2}] \\
&\leq 1 - (1 - \Pr[\ell_1 > B_{m_1}])(1 - \Pr[\ell_2 > B_{m_2}]) \\
&\leq \frac{2}{n^{\alpha}} - \frac{1}{n^{2\alpha}} = \frac{2}{2^{\lambda}} - \frac{1}{2^{2\lambda}}.
\end{aligned}$$

That is to say, the probability that a message is dropped (aka overflow) is negligible in λ. This completes the proof.

In practice, we usually round the above bound up to an integer that is greater but nearest to it. Figures 13 and 14 have demonstrated that our maximum batch size only incurs marginal overhead (with extra paddings) on our horizontal scaling design.

Scalability Analysis. We borrow the idea from XRD [54] to define the scalability of the designed system. Specifically, we say that a system is scalable if the number of requests that one server needs to handle trends to zero when the number of deployed servers increases to infinite. Without loss of generality, we start with one entry node that obliviously distributes the incoming messages to a set of B-nodes in Boomerang+. Let m and n denote the number of messages and deployed B-nodes, respectively. According to Theorem 4.1, we know that the upper bound of the number of messages sent to a B-node server is $\frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2})}$. It is clear that

$$\lim_{n\to\infty} \frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}\left(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)} = 0.$$

Now let us consider having more entry nodes in the system. It is intuitive to see that in Boomerang+ each entry node can run independently and in parallel, which effectively eliminates a potential bottleneck at a single entry node. Similar to the observations in Snoopy [32], in Boomerang+, adding more entry-nodes is not entirely free, because it would increase the total amount of messages sent to each B-node server. Suppose we have ν entry-nodes, n B-nodes, and each entry-node has m incoming messages (out of the total $\nu \times m$ messages) for oblivious distribution. The upper bound of the total amount of messages from ν entry-nodes sent to a B-node server is $\nu \times (\frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2})})$, which would still approach 0 when $n \to \infty$. Thus, Boomerang+ can scale with more clients and messages by adding more entry nodes and more B-nodes.

Our analysis mainly focuses on the heavy load cases, where $m \gg n(\ln n)^3$ generally holds. For example, for those deployments in practice with good anonymity, m at each entry node can easily reach at least in the order of 10^5, while the overall workload can largely be handled with no more than n = 100 B-nodes. If in the extreme cases where m becomes small, we have some fallback options: 1) setting the maximum load $B = m$; or 2) adaptively falling back to single-server Boomerang mode. We leave this adaptive switching design as our future work.

Note that our scalability analysis does not give specific configuration guidelines on how to add entry-nodes and B-nodes, because this would be highly dependent on specific requirements on performance, e.g., latency, throughput, etc., and cost, e.g., expenses of adding respective nodes, bandwidth, etc. We would resort to the abstract configuration planner in Snoopy [32] as a good starting point when we push Boomerang+ to a more practical realm in the future.

C Security Analysis

C.1 Proof of Theorem 5.1

As mentioned, due to the deployment of secure enclaves and a round-based communication model in Boomerang, the only thing left is to prove that memory access patterns in the obfuscated message exchange are oblivious. From the description in §3, Boomerang is designed by combining several oblivious algorithms. Here we prove the indistinguishability of memory access patterns in Boomerang in a modular way. First, we demonstrate that our proactive pattern patching (denoted

```
Real_PPP(pkts) : // pkts: the real input packets prepared by clients
  Parse pkts as (pkt_1,...,pkt_m), where m is the number of packets
  is_mtt = False
  for pkt in pkts do
    is_prev_same = O_Equal(pkt.priv_label, Prev(pkt).priv_label)        ▷ Step 3.1: Detect irregular pattern
    is_next_same = O_Equal(pkt.priv_label, Next(pkt).priv_label)
    is_next2_same = O_Equal(pkt.priv_label, Next(Next(pkt)).priv_label)
    is_mtt = is_mtt and is_prev_same or is_next2_same                   ▷ Detect more-than-two pattern
    pkt.R = O_Choose(is_next_same, Next(pkt).S, pkt.S)                  ▷ Step 3.2: Swapping and patching
    pkt.R = O_Choose(is_prev_same, Prev(pkt).S, pkt.R)
    pkt.R = O_Choose(is_mtt, pkt.S, pkt.R)
  end for
Ideal_PPP(pkts') : //pkts': the dummy input generated using public information on the number of packets and the packet size
  Parse pkts' as (pkt_1,...,pkt_m), where m is the number of packets
  is_mtt = False
  for pkt in pkts' do
    is_prev_same = Sim_O_Equal(pkt.priv_label, Prev(pkt).priv_label)    ▷ Step 3.1: Detect irregular pattern
    is_next_same = Sim_O_Equal(pkt.priv_label, Next(pkt).priv_label)
    is_next2_same = Sim_O_Equal(pkt.priv_label, Next(Next(pkt)).priv_label)
    is_mtt = is_mtt and is_prev_same or is_next2_same                   ▷ Detect more-than-two pattern
    pkt.R = Sim_O_Choose(Next(pkt).S, pkt.S)                            ▷ Step 3.2: Swapping and patching
    pkt.R = Sim_O_Choose(Prev(pkt).S, pkt.R)
    pkt.R = Sim_O_Choose(pkt.S, pkt.R)
  end for
```

Figure 15: Real and ideal experiments for an oblivious proactive pattern patching algorithm

as PPP) algorithm built on existing oblivious algorithms is oblivious according to Definition C.1 below. Then we prove that our Boomerang system protects metadata privacy when built on the oblivious PPP. We first give the definition of a secure PPP algorithm.

Definition C.1 *The proactive pattern patching algorithm* PPP *is secure if for any PPT attacker, there exists a PPT simulator such that*

$$|\Pr[\text{Real}_{\text{PPP}}(\lambda)=1]-\Pr[\text{Ideal}_{\text{PPP}}(\lambda)=1]|\leq \text{negl}(\lambda), \quad (4)$$

where λ *is the security parameter,* Real_{PPP} *and* $\text{Ideal}_{\text{PPP}}$ *are experiments defined in Figure 15.*

Below we show that our PPP algorithm satisfies the above definition by proving the Real_{PPP} and $\text{Ideal}_{\text{PPP}}$ experiments are indistinguishable.

Lemma C.1 *Given the oblivious primitives for comparison and assignments* O_Choose *and* O_Equal*, the proactive pattern patching protocol described in §3 and formally defined in Figure 15 is also an oblivious algorithm.*

Proof. To simplify the proof and our description of the simulator, we assume that the packets (aka encrypted messages) received by the server are indistinguishable by size and traffic patterns, which the attacker cannot exploit to distinguish the memory access patterns. Then we need to demonstrate that the memory accesses of the simulated experiment $\text{Ideal}_{\text{PPP}}$ that takes public information as input are indistinguishable from those of the real experiment Real_{PPP}. As shown in Figure 15, we leverage the following oblivious building blocks in the Real_{PPP} experiment.

- O_Choose(cond, a, b): If cond = True, it outputs value a. Otherwise, it outputs value b.
- O_Equal(a, b): Obliviously assigns True to the output if a equals b. Otherwise, it outputs False.

The simulated experiment $\text{Ideal}_{\text{PPP}}$ is built on top of simulations of the above oblivious building blocks.

- Sim_O_Choose(a, b): Simulates choosing from (a, b) as the output, given a hidden bit.
- Sim_O_Equal(a, b): Simulates testing whether a equals b and outputs True if they are equal.

With these building blocks, we show that the memory access patterns in the real and ideal experiments defined in Figure 15 are indistinguishable. Specifically, if an attacker can distinguish the $\text{Ideal}_{\text{PPP}}$ and Real_{PPP}, then the distinguishability must occur in at least one of the steps. But this happens only with negligible probability because: 1) the simulator uses the public information to simulate indistinguishable

dummy input from real input (i.e., the number of packets m and the packet size), and accordingly follows the same `for` loop structure as the $\mathsf{Real}_{\mathsf{PPP}}$; 2) for each iteration inside the `for` loop structure, the security of oblivious building blocks `O_Choose` and `O_Equal` ensures that the corresponding simulations `Sim_O_Choose` and `Sim_O_Equal` produce indistinguishable memory access patterns; and 3) the operations on direct assignment to `is_mtt` in both $\mathsf{Ideal}_{\mathsf{PPP}}$ and $\mathsf{Real}_{\mathsf{PPP}}$ are also indistinguishable. This completes the proof of Lemma C.1.

Proof of Theorem 5.1. Based on the fact that the proactive pattern patching (`PPP`) algorithm is oblivious, we continue to prove Theorem 5.1 under the security notion defined in Definition 2.1. Let b and b' be the choices of the challenger and the attacker $\mathcal{A}$, respectively, in the experiment EXP defined in Definition 2.1.

From the attacker's view, as all messages are encrypted, the only way to distinguish the communication patterns of two given clients is by observing memory access patterns during the "obfuscated" message exchange procedure. We assume all involved oblivious algorithms are computationally indistinguishable, except with negligible probability in λ (aka $\mathrm{negl}(\lambda)$). As seen, Boomerang leverages an oblivious proactive pattern patching algorithm (as shown in Lemma C.1) and an oblivious sorting algorithm [10]. Let "`M` fails" denote the event that the memory access patterns of at least one of the algorithms mentioned above fail to achieve obliviousness, which happens only with negligible probability in λ. Thus, we have

$$\begin{aligned}
&\mathsf{Adv}_{\mathsf{EXP},\mathcal{A}} = \left|\Pr[b = b'] - \Pr[b \neq b']\right| \\
&\leq \max\{\Pr[b = b', \texttt{M fails}], \Pr[b \neq b', \texttt{M fails}]\} \\
&= \max\{\Pr[b = b' | \texttt{M fails}], \Pr[b \neq b' | \texttt{M fails}]\} \cdot \Pr[\texttt{M fails}] \\
&\leq \Pr[\texttt{M fails}] = \mathrm{negl}(\lambda).
\end{aligned}$$

The above shows that the attacker cannot identify whether two clients are communicating or not within each round, except with negligible probability. For the fixed system configuration across rounds, it is easy to see that the attacker's view will remain the same across rounds. This completes the proof of Theorem 5.1.

C.2 Proof of Theorem 5.2

The proof of Theorem 5.2 is analogous to that of Theorem 5.1. The only difference between Boomerang and Boomerang+ is that Boomerang+ employs entry nodes as load balancers to distribute batches of messages from all clients to a group of B-nodes for message exchange. Thus, the key is to show that this distribution procedure is oblivious.

We assume that the system configuration is fixed across rounds, including the number of entry nodes, B-nodes, and connected clients to each entry node. First, we show that Boomerang+ assigns a message to each B-node uniformly. Note that Boomerang+ assigns a message to each B-node by computing $\texttt{br_id} = \mathsf{H}_k(\texttt{priv_label}||\texttt{round_num})\%n$, where H is a keyed cryptographic hash function, and n refers to the number of B-nodes. According to the classical simulation-based security definition of a keyed hash function [17], the result of a hash function and a random value is computationally indistinguishable. It implies that the distribution of `br_id` is uniform, and further confirms that allocating messages to different B-nodes can indeed be formulated as a random balls-to-bins assignment. Therefore, we can apply the batch size derived from Theorem 4.1 to set up the batch structure without overflow. Moreover, the batch size is determined by the public information (as shown in §4.2.3) only, independent of the input.

Based on the above initial result, it follows that the probability that a message is assigned to any individual B-node is always equal. Without loss of generality, we assume that the number of deployed entry nodes is ν. Let c be an encrypted message sent by a client and e be its refreshed copy by the entry node. The probability for the message assigned to the t-th B-node is

$$\Pr[c \to \mathbb{B}_t] = \sum_{j=1}^{\nu} \Pr[c \to \mathbb{E}_j] \cdot \Pr[e \to \mathbb{B}_t] = \frac{1}{n},$$

where $\mathbb{B}_t$ denotes the t-th B-node, $\mathbb{E}_j$ denotes the j-th entry node, and $c \to \mathbb{B}_t$ denotes that message c is assigned to $\mathbb{B}_t$ finally. This result holds as long as no empty B-node exists during one communication round, which is guaranteed by our uniform message assignment and oblivious sub-batch padding algorithms. Our oblivious sub-batch padding algorithm is largely based on existing building blocks, including the oblivious padding algorithm in Snoopy [32]. Thus, we omit the proofs for its obliviousness here. With the above results, we can obtain that messages from any two clients i and j, whether they are communicating or not, are assigned to the same B-node t with the same probability $1/n^2$. In other words, whether the two clients are communicating or not, their message assignment from entry node(s) to a single B-node is indistinguishable from the attacker.

Now let's focus on messages assigned to each individual B-node. According to Theorem 5.1, an attacker cannot identify whether any pair of clients are communicating or not at a single-server Boomerang. Thus, it follows that at each individual B-node in Boomerang+, an attacker cannot identify whether any pair of messages are from two communicating clients or not. It holds in any single round. As the assignment mapping (through the keyed hash function H) is refreshed for every round, it is easy to see that the attacker's view will remain the same across rounds, with a fixed system configuration. It ensures the indistinguishability of whether two clients communicate or not in Boomerang+.

Finally, we need to prove that the memory access patterns in Boomerang+ are oblivious. To achieve such obliviousness, we build Boomerang+ on top of a group of oblivious primitives

Table 1: Latency with varying numbers of entry nodes (#E) and B-nodes (#B). The best ratios with the lowest latency are marked in bold.

	#E	#B	Latency	#E	#B	Latency	#E	#B	Latency	#E	#B	Latency
8 Servers	7	1	20.10	6	2	16.08	**5**	**3**	**15.54**	4	4	16.19
	3	5	20.57	2	6	28.71	1	7	58.47			
16 Servers	14	2	12.19	**12**	**4**	**10.37**	10	6	10.45	8	8	10.71
	6	10	11.65	4	12	17.16	2	14	27.54			
24 Servers	22	2	12.62	20	4	10.18	**18**	**6**	**8.09**	16	8	9.59
	14	10	8.41	12	12	9.11	10	14	9.46	8	16	10.71
	6	18	12.12	4	20	14.95	2	22	27.81			
32 Servers	30	2	11.77	28	4	9.35	26	6	8.45	**24**	**8**	**7.76**
	22	10	8.26	20	12	8.41	18	14	8.78	16	16	8.23
	14	18	8.75	12	20	8.99	10	22	9.27	8	24	9.83
	6	26	12.47	4	28	16.49	2	30	28.46			

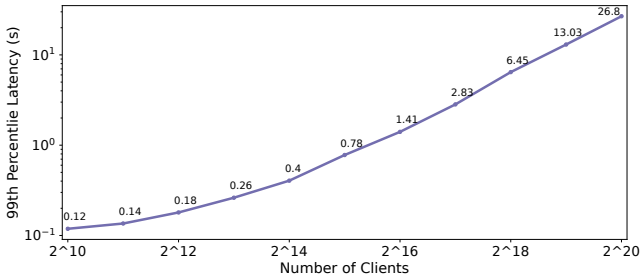


Figure 16: 99th percentile latency of Boomerang.

including oblivious comparisons, assignments, compaction, and sorting functions. Based on these oblivious primitives, we develop the oblivious batch generation and distribution procedures, and those based on the oblivious PPP algorithm with slight modifications in Boomerang+, just like the way we develop Boomerang. Thus, we do not spend more space repeating that their algorithms are oblivious. By the above two derivations, we show that Boomerang+ is secure according to Definition 2.1. This completes the proof.

D Boomerang Performance

Latency. We evaluate Boomerang on one 16-core server and test the latency over up to 2^{20} clients. Figure 16 shows the 99th percentile latency of Boomerang with a varying number of clients. For 2^{15} clients, the latency is 778 ms, which is also enough for VoIP communication. Notably, Boomerang achieves 1.41 second latency for 2^{16} clients using only one server. We can observe that the latency increases (almost) linearly with the number of clients (messages). There are mainly two factors: 1) with the increasing of clients, the server needs to handle more RPC requests; and 2) the most expensive computation in enclaves is oblivious sort (twice), which is of $O(n(\log n)^2)$ complexity. When the client number increases to 2^{20}, the latency reaches 26.8 seconds, which is no longer suitable for latency-sensitive applications. The increased latency also shows the need for horizontal scalability.

Bandwidth cost. Boomerang requires each client to send a message of 256 Bytes every round. It occupies 512 Bps bandwidth for each client if we set a 500 ms round. If the client keeps online on Boomerang for one month, the bandwidth cost is 2.47 GB (including sending and receiving data), which we believe is affordable for ordinary users.

E Supplementary Experiments for Resource Allocation

Recall that in §6.3 we show the best resource allocation for entry nodes and B-nodes with 16 servers that can achieve the lowest latency. This section further reports the best allocation with 8, 16, 24, and 32 servers. We have exhaustively tried different combinations of entry nodes and B-nodes for each set of servers and let them handle 2^{20} messages. The results are reported in Table 1. The best allocation for entry nodes and B-nodes for 8, 16, 24, and 32 servers are 5:3, 12:4, 18:6, and 24:8, respectively. Note that this is a brute-force way to get the best ratio. A possibly more efficient way to configure the system is to design a configuration planner [32], which we will leave as our future work.

F System Dollar Cost

We here report the estimated cost in US dollars of running Boomerang+ servers. Leaving human-operation costs aside, we calculate the total cost for hosting Boomerang+ servers and data egress costs and show how much each user would (at least) pay for joining Boomerang+ for one month.

Server cost. Since the Tencent Cloud machines we use did not announce the international pricing for `M6ce.4XLARGE124` [85], we alternatively choose `Standard_Dc16s_v3` instances from Azure Cloud for price estimation, which have the same properties as `M6ce.4XLARGE128`. According to the prices for Azure Cloud VMs [65], the host (machine) cost is 1121 USD/month. The cost varies with different performance goals. For example, running 16 instances for 2^{16} clients achieves 615 ms latency, which results in $0.274 amortized monthly cost per client. In the scaling experiment, Boomerang+ runs 32 8-core instances for 2^{20} clients, resulting in $0.017 amortized cost and 7.76 second latency based on the price for the alternative instance `Standard_Dc8s_v3` (560 USD/month). Ideally, adding more servers will further reduce the latency, but it will also increase the overall server cost.

Bandwidth cost. We assume that each client sends a 256 Byte message every 500 ms, adding up to 1.24 GB data ingress to (or egress from) the server if the client stays online for one month. To save data transfer costs, we can set the servers in the same availability zone, within which the transferred data is free of charge. If the clients and servers transfer data between different continents, the price is at most 0.16 USD/GB [63]. Then, the server-side bandwidth cost is 0.198 USD/month for each client. Overall, we believe Boomerang+ is affordable for clients while maintaining good privacy and high-performance services. Its promising cost is comparable to non-private cloud-based IM services today [64, 69, 84, 86].

Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies

James Lovejoy
Federal Reserve Bank of Boston

Madars Virza
MIT Media Lab

Cory Fields
MIT Media Lab

Kevin Karwaski
Federal Reserve Bank of Boston

Anders Brownworth
Federal Reserve Bank of Boston

Neha Narula
MIT Media Lab

Abstract

Over 80% of central banks around the world are investigating central bank digital currency (CBDC), a digital form of central bank money that would be made available to the public for payments. We present Hamilton, a transaction processor for CBDC that provides high throughput, low latency, and fault tolerance, and that minimizes data stored in the transaction processor and provides flexibility for multiple types of programmability and a variety of roles for financial intermediaries. Hamilton does so by decoupling the steps of transaction validation so only the validating layer needs to see the details of a transaction, and by co-designing the transaction format with a simple version of a two-phase-commit protocol, which efficiently applies state updates in parallel. An evaluation shows Hamilton achieves 1.7M transactions per second in a geo-distributed setting.

1 Introduction

Central banks are increasingly investigating general-purpose central bank digital currency (CBDC), a digital currency that would be broadly available to users making retail payments, could provide interoperability and programmability depending on how it is designed, and, because it would be a direct liability of the central bank, reduces risk [7, 8, 14, 15, 19, 22, 23, 24, 37, 47, 65].

Figure 1 summarizes the different properties of a CBDC as compared to other forms of payment instruments [9]. A CBDC could help address public policy objectives such as ensuring public access to central bank money, fostering payment competitiveness and resilience, supporting financial inclusion, and offering privacy-preserving digital payments [3, 7, 15, 42, 70].

Technical designs for CBDC vary depending on specific policy requirements and goals. For example, a CBDC could provide low value payments in an anonymous, peer-to-peer fashion, or be distributed and accessed only through accounts at approved financial institutions. To better inform policy discussions, central banks are recognizing the importance of technical experimentation in understanding the implications and trade-offs of different CBDC models and other policy choices. Importantly, the feasibility, operating performance and impact of different CBDC policy choices are dependent upon the technical design of the underlying transaction processor.

This paper presents a collaboration with a major central bank in the design of Hamilton, a high-performance transaction processing system flexible enough to support experimentation with different choices around data storage, programmability, and intermediation. Hamilton processes payments from users (or financial institutions) who address and sign transactions using cryptographic keys stored in digital *wallets*. Wallets submit transactions to the Hamilton transaction processor to move *unspent funds*—a representation of money containing an amount and the rules required to spend it (in our case, a public key indicating ownership). Our initial goals (set by the central bank) were a centralized transaction processor for CBDC with high performance and geo-replicated resiliency. Three additional goals emerged within the collaboration:

Intermediary and custody flexibility. An open question in CBDC design is that of the role of the central bank and other intermediaries, and determining how (if at all) CBDC access can be moderated. These roles will likely vary by jurisdiction, due to policymaker decisions and consumer preferences. Currently, people who want to digitally store funds and make payments must open accounts with financial institutions or payment service providers which are linked to the identity of the owner and are responsible for processing transactions on behalf of their customers, interfacing with payment networks, and safeguarding customer funds. In contrast, cash can be held directly by the public and used to conduct transactions without the need for a financial institution to process the payment on their behalf. A CBDC could be designed to offer similar functionality to cash and provide users the ability to spend their own funds without the need for an account provider or custodian to generate transactions, it could be designed more like existing digital payment

The views expressed in this paper are those of the authors and do not necessarily reflect the views of the Federal Reserve Bank of Boston or the Federal Reserve System.

Property	Cash	Bank deposits	Cryptocurrency	Central bank reserves	CBDC
Electronic		✓	✓	✓	✓
Central-bank issued	✓			✓	✓
Universally accessible	✓	✓	✓		✓

Figure 1: Table describing the properties of various monetary instruments, summarized from Graph 3 in [9].

systems, or it could even support a combination of the two models [17].

Interoperability and programmability. Many payment processor systems provide high throughput and low latency, but unfortunately, they generally provide limited application programming interfaces (APIs) and do not natively *interoperate*. For example, if a Venmo user wants to pay a user who only has Square Cash, they would need to transfer their funds outside the application into bank accounts and send the money via traditional banking rails (inside the United States using ACH or FedWire; across borders, even in the same currency, using SWIFT and correspondent banking), which incurs bank and application fees and could take days to complete. Central bank Real-Time Gross Settlement Systems (RTGS), or instant payment systems, help reduce interoperability latency but do not preclude the requirement for multi-step transfers between applications and bank accounts (where fees are charged) or provide interoperability and programmability, especially between payment applications. Contrast these systems with cryptocurrencies which do not scale well but natively provide interoperability and programmability.

Preserving privacy and minimizing data retention. There is strong user demand for financial privacy [37] and central banks would prefer not to collect and store user-identifying information or sensitive transaction details.

Challenges. Building Hamilton to achieve these goals required addressing the following challenges. First, we had to build a flexible platform that could support multiple designs without explicit policy requirements or well-defined tradeoffs. For example, it is unclear what balance to target between end-user privacy and data storage requirements for users at the central transaction processor. We take a layered approach with a design where additional functionality can be built outside the core transaction processor. Our design supports a range of intermediary roles including one where users custody their own funds. Hamilton does not store personally identifiable information (PII), transaction addresses or amounts in the core of the system.

The second challenge is in providing strong consistency, geographic fault tolerance, high throughput, and low latency, all with a workload that consists of 100% read/write, multi-server transactions. Since Hamilton is unaware of the mapping between users and unspent funds, we cannot rely on user locality for partitioning, which is often exploited by traditional database systems to make workloads predominantly single-partition transactions.

Key ideas. We address these challenges in Hamilton by carefully co-designing the transaction format, data model, and distributed transaction commitment protocol to achieve the above goals while getting good performance. This involves three parts: First, we decouple transaction validation from fund existence checks; only user wallets and a *validating layer* see transaction details. Hamilton only ever stores funds as opaque 32 byte hashes, in an *Unspent funds Hash Set*, or UHS [41] (§3.3). This hides details about the funds (like amounts and addresses) from the UHS storage, reduces storage requirements, and creates opportunities to improve performance, described below.

Next, we next create a UHS-designed transaction format (§3.4), which is extensible and secure against double spends, inflation attacks, replay attacks, and malleability, and also has the benefit of supporting future layer 2 designs for even higher throughput in the future. It borrows from Bitcoin's transaction format but is designed to be validated without looking up data from the UHS, which we term *transaction-local validation*.

Our design choices let us exploit payment application semantics to create a streamlined commit protocol for distributed transactions. In particular, our transaction format guarantees that Hamilton knows the read and write sets for every valid transaction *before* its execution; similarly, our cryptographically-generated UHS identifiers (hashes) are globally unique. These two properties guarantee that Hamilton does not need any reads or locks before commit time, and that valid transactions (those that do not try to spend the same funds twice) will never conflict.

Our evaluation shows that co-design achieves improved performance over more general, commercial databases. We measured Hamilton's throughput at 1.7M txns/sec, 26× that of PostgreSQL on the same workload, though with higher latency (§6). We also compare against Rolis [64], a replicated in-memory database, and show that Hamilton achieves close performance, even though it stores data on disk for whole system crash recovery.

The UHS design, in combination with our transaction format, also affords us substantial flexibility. We believe that the abstractions our system provides and the assumptions it makes are compatible with most ideas underlying certain types of programmability and cryptographic privacy-preserving designs [10, 52, 69, 71]. In addition, we can upgrade the scripting language or add a cryptographic privacy-preserving protocol (even supporting multiple concurrent designs), as long as they are com-

patible with 32-byte hash storage, without needing any changes to the backing UHS, making it possible to defer decisions on specific programmability features. However, our design choices have implications on what data users or intermediaries need to store in their wallets and what messages are required to confirm a payment (§3.7).

In summary, the contributions of this paper are the following:

- Hamilton, a flexible transaction processor design that supports a range of models for a CBDC and minimizes data storage in the core transaction processor while supporting self-custody or custody provided by intermediaries
- A transaction format and implementation for a UHS which together support modularity and extensibility
- An implementation and evaluation which shows good performance in comparison to centralized databases. Hamilton and the benchmarks are available at https://github.com/mit-dci/opencbdc-tx.

2 System model and security goals

This section describes the actors in Hamilton, their roles, and the security properties we want Hamilton to satisfy. In our description, we make the simplifying assumption that users directly custody their money without help of an intermediary. Hamilton supports intermediaries, and adding an intermediary would not change the core security properties of the transaction processor.

2.1 Actors

We distinguish three types of actors: the *transaction processor*, the *issuer*, and *users*. The transaction processor keeps track of *funds* which are owned by different users. Funds are a representation of money and as such refer to an amount of money (such as dollars) and a condition that must be satisfied to move this amount (say, to another user or users). The funds enter and exit the system through acts of the *issuer* who can *mint* and *redeem* funds to add and remove them from the transaction processor, respectively. Users can execute *transfer* operations (transactions or payments) that atomically change the ownership of funds, with the requirement that the total amount of funds stored in the transaction processor has not changed. A user does so by submitting their transaction to the transaction processor over the Internet, which the processor then validates and executes. Figure 2 shows the high-level system model and potential communication channels between users and the transaction processor. Users run *wallet* software (e.g. on mobile phones or specialized hardware in smart cards) to manage cryptographic keys, track funds, and facilitate transactions. An important piece of future work is preventing spam and denial of service attacks.

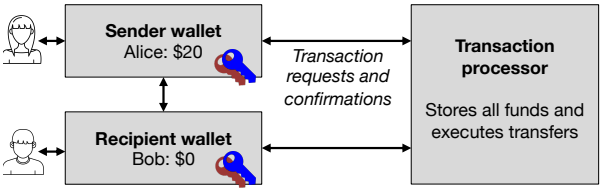


Figure 2: Data flows between all participants in a transaction.

2.2 Threat model

Our goal is that each user's funds and the integrity of the monetary system are safe from interference of an external actor. We assume that the transaction processor is faithfully executing our design, that users' wallets are able to maintain secret keys, and that the users are able to use a secure channel to communicate with the transaction processor. Our design is a cryptographic system so we assume the security of standard cryptographic primitives such as hash functions and digital signatures.

We aim to protect against an adversary who can freely interact with the system as a regular user, and as such make no additional assumptions about an adversary's capabilities or behavior. For example, the adversary is free to create arbitrarily many identities and wallets, receive funds from other users, and engage in elaborate transaction patterns. Our designs are multi-server systems and the adversary is free to attempt concurrent attacks against all externally-exposed parts of the system.

2.3 Data representation

The two most common ways to represent funds are the account balance model and the UTXO model.

Account model. Traditional payment systems and several cryptocurrencies, like Ethereum [73] use an account model where the system stores unspent funds as balances associated with unique account identifiers. Users make payments by issuing requests to the transaction processor to move balance to another identifier (decrementing their balance and incrementing another identifier's balance).

UTXO model. Another choice is to track discrete pieces of outstanding funds without explicitly consolidating them in a single balance. For example, Bitcoin maintains an append-only ledger of accounting entries (sometimes called "coins") each of which records a value and conditions to spend the funds. Furthermore, each entry is either marked as "spent" or "unspent". Users make payments by issuing transactions that mark some entries (*inputs*) as spent, and appends new unspent entries (*outputs*) to the ledger. In Bitcoin these are called *UTXOs* or Unspent Transaction Outputs. Importantly, UTXOs are never modified and must be spent in their entirety. Therefore, Alice wishing to use a $10.00 UTXO to send $4.99 to Bob

will create a transaction with two outputs: a $4.99 output meant for Bob and a $5.01 *change output* meant for herself.

We derive Hamilton's data model from the UTXO model for two reasons: First, because it offers greater transaction execution parallelism; inputs can only be spent once, and so in the common case there should never be conflicting transactions, unlike concurrent transactions against a popular account. We leverage this in our data storage design (§3.4). Second, UTXOs are the leading choice for privacy designs [10, 16, 26, 45, 48, 69, 71] (including for those deployed on top of account-based currencies [61, 72]). UTXOs can be less intuitive to the user, but note that the transaction processor's internal data representation is distinct from the user interface; wallets can still present a user account balance on top of the UTXO data representation.

2.4 System operations

Logically, Hamilton maintains a record of all unspent funds in existence and in order to spend funds, they must be present in the set of unspent funds. Our system supports three kinds of operations: Mint, Redeem, and Transfer, all of which are atomic and serialized.

Minting and redeeming. The Mint operation creates new unspent funds and adds these to the set of unspent funds, whereas the Redeem operation removes unspent funds from the system, making them unspendable. These operations also have semantics outside Hamilton: minting would normally correspond to funds in other forms of central bank money being set aside for use in Hamilton, whereas redeeming would make them available again.

Value transfers. The Transfer operation both consumes unspent funds and creates new unspent funds. This transaction is specified by a list of funds to be spent (inputs), a list of new funds to be created (outputs), and a list of witnesses (i.e., digital signatures) authorizing spends of each input. A successful Transfer completely consumes its inputs; these are removed from the system and cannot be used again, whereas the new outputs are available to be used as inputs to other Transfer or Redeem operations.

No editing of unspent funds. The set of unspent funds can only be modified via the above three operations, and funds tracked in the system cannot be modified to change their ownership or value.

Payment discovery. In public blockchains users can search the publicly available history of transactions to see if they have received payment. Transaction history in Hamilton is not public, and the sender must give the recipient the information about newly created unspent funds so that the recipient can further spend them. To ensure users know a Transfer is complete and has been applied, the transaction processor is also responsible for responding to user queries about the existence of unspent funds.

2.5 Security properties

In brief, the system must faithfully execute transactions, ensuring that each was authorized by the owner of the input funds, and safeguard that transactions do not disturb the overall balance of funds (outside of minting and redemption). Hamilton's transaction processor ensures this by satisfying the following four security properties.

Authorization. Hamilton only accepts and executes Mint and Redeem operations authorized by the issuer, i.e., only the issuer can mint and redeem funds. We use digital signature authorization for these. Similarly, we require that each Transfer transaction is signed by owners of all inputs the transaction attempts to spend.

Authenticity. The set of unspent funds tracked in Hamilton only contains *authentic* funds, as we now define. Define unspent funds created by authorized Mint operations to be *authentic*. Moreover, define unspent funds created by Transfer operations to also be authentic if and only if all inputs consumed by the transaction were authentic and the transaction preserves balance. Note that the recursive authenticity property depends on both the contents of the transaction itself, as well as the set of unspent funds when Transfer is applied.

Durability. Mint, Redeem, and Transfer are the only operations in Hamilton that change the set of unspent funds.

As a consequence of the three integrity properties defined above the set of unspent funds always remains authentic and transactions in Hamilton cannot be reverted. We further require that the transaction processor makes the following availability guarantee and always makes progress:

Availability. The transaction processor will always accept an authorized transaction spending authentic funds.

3 Transaction design and processing

This section first reviews Bitcoin's UTXO model in more detail and explains the challenges associated with using this data model in our setting. It then describes the UTXO hash set (UHS), a different idea that we choose as a basis for Hamilton's data model and the motivation behind our choice. Finally, it introduces Hamilton's transaction format, describes how to securely create and process transactions in this model, and discusses implications on future functionality.

3.1 Bitcoin's UTXO model

Bitcoin uses the UTXO model, where each output utxo has a value and an encumbrance: The value v is an integer multiple of the smallest subdivision of Bitcoin and an encumbrance is a *script*, an executable program which evaluates the conditions for a valid spend. An encumbrance expresses a predicate P taking two arguments: a transaction tx seeking to spend this utxo, and a witness wit. A

transaction is a list of references to input UTXOs to be consumed together with a list of corresponding witnesses, and lists of values and encumbrances for new UTXOs to be created. Each encumbrance predicate returns true if and only if its corresponding witness signifies that this spending transaction should be authorized.

A common encumbrance is that of digital signature authorization, known as pay-to-pubkey (P2PK). Here the predicate P hard-codes a public key pk and $P(\mathsf{tx}, \mathsf{wit})$ checks that wit consists of a valid signature under the public key pk where the message is the serialized spending transaction tx. To spend such a utxo, the user creates a transaction tx having the utxo as an input and signs tx with the corresponding secret key sk.

3.2 UTXO model challenges

Adopting Bitcoin's design in our setting comes with a number of challenges. First, Bitcoin's UTXO model requires maintaining a copy of the entire UTXO set in full detail. This has unwanted implications for privacy as Bitcoin node operators are both privy to values and encumbrances (i.e., users' public keys), as well as for data storage, especially when using complex or post-quantum secure encumbrances, which might be large. Second, in Bitcoin, transactions only refer to their inputs by specifying a hash txid of a prior transaction together with a particular output index idx of that transaction. Thus, to validate a transaction a node has to look up the input encumbrances and values in its local UTXO set, and only trivial validation checks can be done statelessly. This is reasonable when the UTXO set is small and nodes store it locally, but becomes more challenging when it must be partitioned across many machines to achieve higher performance, while still being accessed consistently.

3.3 UTXO hash set

A key observation in the design of Hamilton is that we can divide transaction validation checks into two parts – *transaction-local validation*, which does not require access to shared state, and *existence validation*, which does. We can then scale these two tasks independently. This is useful because they have different scalability profiles, with transaction-local validation requiring mostly compute resources (i.e., verifying digital signatures used in spend authorization) and existence validation requiring mostly persistent storage I/O.

By doing this, we can go even further and observe that after transaction-local validation, instead of processing and storing the entire UTXO, we can operate on cryptographic *commitments* to the UTXOs. In Hamilton we replace the UTXO set with a *UTXO hash set* (UHS), extending an idea first proposed as a Bitcoin storage and scalability improvement [41]. That is, our transaction processor stores unspent funds as a set of opaque 32-byte cryptographic hashes of UTXOs, not UTXOs themselves. We refer to hashes of UTXOs as UHS IDs or simply hashes. Instead of looking up the transaction input data (which we do not have), we ask the (untrusted) user to provide full input UTXOs in a transaction. However, a malicious user might lie and claim to have more funds to spend than they actually do. To catch this, we reduce the problem of checking UTXO correctness to *UHS commitment existence*—Do the funds the user is claiming they can spend actually exist? As we'll see in §4, this affords us the opportunity to piggyback existence validation with actual execution inside the distributed transaction commit protocol.

UTXOs must be stored in the user wallets and are supplied as part of transactions. We also note that while in Bitcoin the UTXO set is derived by processing the Bitcoin blockchain and keeping the set of unspent UTXOs, Hamilton's backend is a transactional database that maintains the UHS without operating a ledger.

Using a UHS has a number of benefits. First, as described above, a UHS-based transaction format lets us decouple transaction validation from funds existence checks and affords us opportunities for performance improvement in the backend. Second, it lowers storage requirements, as the transaction processor only stores a 32-byte hash per UTXO, independent of a UTXO's size. Third, it increases flexibility, as the UHS abstraction makes no assumptions about what hashes represent: it is easy to adapt a high-performance system maintaining a UHS, like Hamilton, to a different transaction formats or scripting languages without needing to change the core execution engine. Fourth, it improves privacy as the transaction processor does not store balances or account information.

However, a UHS design also presents some challenges, stemming from its data minimization. The UHS, as described above, does not contain enough information to audit the total amount of unspent funds (the "full" UTXOs only reside in user wallets). However, UHS hashes could be augmented to store a value alongside hashes (making supply auditing trivial, at some privacy cost), or by converting UHS IDs to homomorphic commitments that can be maintained and tallied using additional cryptographic techniques [54, 60]. The sender also has to provide the recipient with the UHS ID preimage to further spend their funds, as described in §3.7. Finally, decoupling transaction-local validation and access to shared state means that future transaction programmability is restricted to only referencing transaction-local data.

3.4 Transaction format

To build Hamilton we designed a new, extensible transaction format in the UHS model. As we will see later, Hamilton's transactions can be split into transaction-local validation and existence checks.

Unspent funds. We represent unspent funds as triples

$\mathsf{utxo} := (v, P, \mathsf{sn})$. Here v and P are value and encumbrance. Currently we only support encumbrances of public keys, and thus represent predicate P by the 32-byte public key pk itself. Our model supports future encumbrances, such as requiring a subset of signatures from multiple public keys.

The third component, sn is a globally unique *serial number*. The serial number, enables us to reference and distinguish funds that share the same encumbrance and value (e.g., Alice having received same $5.00 value in two different transactions encumbered with the same public key $\mathsf{pk}_{\mathsf{Alice}}$); it also implies that UHS is a set, not a multiset. Our transaction format will ensure that serial numbers do not repeat across time: a serial number associated with a spent UTXO cannot "reappear" as a serial number for a new unspent UTXO. Global uniqueness of serial numbers is not a mere technicality: they express the intent of singling out a particular UTXO and prevent *replay attacks* (see §3.6 for discussion).

UTXO hash set. Instead of storing a set of entire UTXOs $\mathsf{utxo} = (v, P, \mathsf{sn})$, we store cryptographic commitments $h := \mathscr{H}(v, P, \mathsf{sn})$ to UTXOs. In Hamilton we set $\mathscr{H}$ to be SHA-256 to derive these hash commitments.

Mint **transactions.** New funds enter the system by system operator creating new utxo's and adding their hashes to the UHS. The issuer must choose unique serial numbers for newly minted UTXOs. It suffices to set these as uniformly random nonces.

Transfer **transactions.** A k-input, l-output Transfer transaction seeks to fully consume k UTXOs currently present in the system, and create l new UTXOs specified by encumbrances and values. Such a transaction $\mathsf{tx}_{\mathsf{Transfer}} = (\vec{\mathsf{utxo}}_{\mathsf{inp}}, \vec{v}_{\mathsf{out}}, \vec{P}_{\mathsf{out}}; \vec{\mathsf{wit}})$ is comprised of (a) a size-k list $\vec{\mathsf{utxo}}_{\mathsf{inp}}$ of input UTXOs to be spent; (b) two size-l lists $\vec{v}_{\mathsf{out}}$ and $\vec{P}_{\mathsf{out}}$ of output values and encumbrances specifying output UTXOs to be created; and (c) a size-k list of witnesses $\vec{\mathsf{wit}}$, one for each input.

Such $\mathsf{tx}_{\mathsf{Transfer}}$ creates l UTXOs with value/encumbrance pairs $(v_{\mathsf{out},i}, P_{\mathsf{out},i})$. We make UTXOs unique by deriving the serial numbers sn as pairs $\mathsf{sn} := (\mathsf{txid}, \mathsf{idx})$ as follows. The first component, txid is the unique transaction identifier, the cryptographic hash of the transaction that created this UTXO. This hash covers all input UTXOs, output encumbrances and values: $\mathsf{txid}(\mathsf{tx}_{\mathsf{Transfer}}) := \mathscr{H}((\vec{\mathsf{utxo}}_{\mathsf{inp}}, \vec{v}_{\mathsf{out}}, \vec{P}_{\mathsf{out}}))$. The second component, idx, is the particular output index, i.e., first, second, etc, output of the transaction.

Note that our transaction format includes input UTXOs in the Transfer itself. In contrast, a Bitcoin transaction does not: it references UTXOs via txid and idx instead, and requires UTXO look-ups in transaction processing.

Transaction creation. To create a Transfer transaction, users digitally sign txid with private keys corresponding to inputs they are spending. Each of these signatures serves as the witness authorizing the transaction to spend the given input. Witnesses are not included in the transaction identifier so signing can be deferred by the sender to after the transaction has been shared with the recipient. This is useful to support future smart contract functionality where unsigned transactions could be shared between parties to be signed and broadcast later under certain conditions. Recall that encumbrances are applied to individual outputs rather than whole transactions, and transactions can have multiple inputs, which means that funds can be spent atomically from multiple public keys in a single transaction. Once a transaction is finalized, the users will deterministically derive serial numbers of each of the output UTXOs from the transaction contents. Users store this outpoint information in their wallets.

3.5 Transaction execution

Processing a Transfer transaction involves confirming that it is valid and then applying it to the state. Validation involves checking the following:

1. *Syntactical correctness.* Check that the transaction has at least one input and output, and that the transaction supplies exactly one witness per input.
2. *Balance.* Check that transaction's input values tally up to exactly the same value as outputs to be created.
3. *Authorization.* Check that each input UTXO is accompanied by a valid signature, relative to the input's public key, on a message comprised of the transaction's identifier txid.
4. *Authenticity.* Check that transaction's input hashes exist in the UHS.

To apply a valid transaction to the UHS we atomically remove the spent input hashes and create the new output hashes under the control of the recipient(s); this in combination with the other checks provides durability.

Performing local-validation. Hamilton has dedicated components, which we call *sentinels*, that receive transactions from users and perform transaction-local validation. This local validation performs the above syntactical correctness, balance, and authorization checks.

Compaction. We further observe that while the transaction-local validation does not reference any data from the state and only uses transaction-local data, the UHS, in turn, does not reference a transaction's contents and only operates on the hash values. Thus, once locally validated, a transaction is *compacted*. First, the sentinel derives the output UTXO serial numbers; together with output encumbrances and values they fully specify output UTXOs to be created. Next, the sentinel hashes the input and output UTXOs and obtains two lists of hashes which we call a *compact transaction*. Finally, sentinels forward

compact transactions to the execution engine, which maintains the UHS, for existence checks and to update the UHS state.

Checking existence, execution and the `swap` abstraction. Now, given a compact transaction, our system does the following: First, check if all input hashes exist in the UHS; if so (a) remove the input hashes from the UHS and (b) add the output hashes of newly created UTXOs to the UHS. We call this UHS primitive `swap`. Processing Hamilton transactions at scale reduces to the challenge of implementing a fast, scalable, and durable backend for executing `swap`, which we describe in §4. Such a backend abstraction maintains a set of hashes, and exposes `swap` as the only operation. The inputs to `swap` are two lists of hashes: one for existence checks and removal (called input hashes), and one for insertion (called output hashes). To execute a `swap`, the system atomically checks that all input hashes are present. If an input hash is missing, `swap` aborts. Otherwise, it obtains an updated set of hashes by erasing all input hashes and inserting all output hashes. All other hashes in the set remain unchanged.

We note that separating transaction-local validation and execution means that `swap` supports multiple transaction formats concurrently without affecting UHS performance.

3.6 Transaction format security

Using our transaction format Hamilton maintains an authentic, authorized and durable set of unspent funds (§2.5), eliminates the possibility of double spends, and also achieves additional security goals related to its use. In particular, transactions in Hamilton are not *replayable* and digital signature authorizations are not *reusable*.

These properties are a consequence of the fact that each UTXO created by a Mint or Transfer transaction is unique and guaranteed to not equal any other UTXO in the past or in the future, as we now explain. In Hamilton, each Transfer's UHS IDs are derived by hashing all the corresponding transaction's inputs, as well as details pertinent to the particular output itself (see §3.4). In particular, sn references previous unique serial numbers and recursively incorporates the entire transaction history up to distinct (due to presence of the uniformly random nonce) Mint's. Collision resistance of $\mathscr{H}$ guarantees that these serial numbers are unique. Because UHS hashes commit to the same UTXO data which must be provided in the transaction, an attacker can not fit a different UTXO preimage into the same UHS hash without violating the collision-resistance of $\mathscr{H}$.

No double-spends. Transfer operations permanently delete input hashes from the UHS. Therefore, as serial numbers are unique, no UHS ID can be spent more than once or recreated after having been spent.

No replay attacks. In a basic replay attack the victim has signed a single transaction to authorize a single value transfer. The attacker, however, submits this transaction twice in the hope of effectuating two value transfers. For example, Alice, who has two unspent $5.00 "bills", might give Bob a transaction that spends one of her $5.00 bills to pay for ice cream, which Bob then submits twice to take possession of both.

Hamilton's transaction format prevents replay attacks as each transaction references globally unique input hashes, and each signature covers the entire transaction, including all its inputs and outputs. Thus, signatures are not valid for spending any other UHS ID, including those created in the future, and it is not possible to copy a Hamilton transaction and apply it multiple times to spend additional funds.

Transactions are non-malleable. In a system with malleable transactions, an attacker can change some details about the transaction (e.g., the witnesses used to satisfy input encumbrances or output UTXO serial numbers) without otherwise changing the input UTXOs or modifying output UTXO values or encumbrances. For example, if the transaction format included an auxiliary field not covered by the signatures but used in serial number computation, an attacker could change this field. This would change output UTXO serial numbers and make it unsafe to accept a chain of unconfirmed transactions, thus preventing certain higher level protocols like the Lightning Network. In 2014, the largest Bitcoin exchange Mt. Gox closed after claiming to be a victim of malleability attacks [32]. In our implementation, we require signatures to cover all fields of uniquely-encoded transaction and derive UTXO serial numbers from the same fields (plus, output indexes).

3.7 Transaction protocol

Our choice of transaction data model and format directly impacts potential transaction protocols. For example, transaction compaction for the UHS adds a new communication step requirement between sender and recipient. The recipient should not consider a payment "complete" until they have received both a confirmation from the transaction processor *and* the full preimage data for their new outputs. If the recipient does not receive these, the sender has essentially destroyed the funds.

In theory, cryptocurrencies in which the recipient's address is obfuscated also have this problem. In practice, because the entire blockchain is public and standard address formats are used, recipients can scan *every* transaction to detect if they have been paid and, if so, construct new transactions to spend those funds. Even if the UHS were public, recipients would still not be able to unilaterally detect payments. The hash preimage for a UHS ID depends on data from the transaction that creates the UTXO which is unavailable to the recipient. As there is no public ledger, recipients must rely on the transaction processor to learn about the status of outstanding transactions.

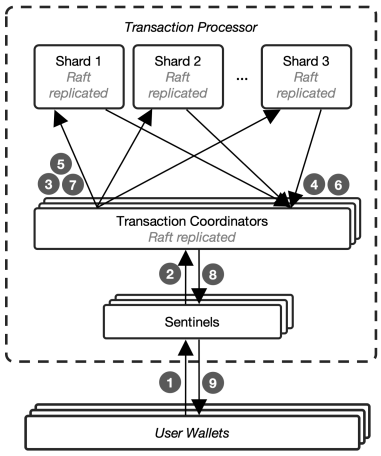


Figure 3: System diagram for the 2PC architecture and inter-component data flow

4 Processing transactions at scale

As described in §3, transaction processing can be split into transaction-local validation, existence validation, and execution, which creates opportunities for improving performance. To process more transactions, we *partition* the set of unspent funds across multiple computers. Transactions might reference unspent funds stored on different machines, requiring a coordination protocol to check existence of inputs and execute transactions atomically.

One way to achieve this is to first explicitly order all valid transactions and subsequently apply them to the partitioned state in the same order, if the inputs exist and have not already been spent. We investigated this architecture and found performance was limited by the ordering server (§4.3). However, we note that payment semantics do not *require* materializing a linear transaction history. Thus, we also built an architecture which executes transactions in parallel to achieve high performance, described next.

4.1 Applying transactions to the UHS

We use variants of two-phase commit and conservative two-phase locking[36, 44] to atomically apply transactions to the UHS, partitioned across *shards* which are each responsible for a subset of the UHS IDs which are unspent within the system.

Figure 3 shows a diagram of the components in the 2PC architecture and the data flow between components. As described in §3.5, a wallet submits a transaction to a sentinel (1), which validates everything *except* the existence of inputs. Upon success, sentinels convert transactions to compact transactions and send compact transactions to a *transaction coordinator* (2).

Each coordinator has a thread pool to execute transactions in parallel, and adds incoming compact transactions from sentinels to a queue. Once a thread from the pool becomes available, it drains the queue and creates a new distributed transaction (*dtxn*) containing the pending compact transactions (up to a maximum size). The thread then performs the 2PC protocol to commit the dtxn. If a thread is available, it will begin a new dtxn even if there is only one compact transaction waiting in the queue, it will not wait for the queue to grow. Due to application choices described in §3, we know the read/write sets ahead of time and can execute the dtxn entirely inside the 2PC protocol, without any extra roundtrips. This is the same technique introduced in Sinfonia [1].

There are three steps to commit a dtxn:

1. **Prepare (3).** The coordinator contacts each shard responsible for a UHS ID included in the dtxn and requests that it durably lock the input UHS IDs (a *prepare* request). (Note that by design of our transaction format (see §3.4), valid output UHS IDs are guaranteed to be unique across transactions, so reserving outputs is not necessary.) Each shard responds to the *prepare* request indicating which compact transactions in the dtxn had their IDs successfully locked, and which no longer exist, or were already locked by a different dtxn (4).
2. **Commit (5).** The coordinator uses the shards' responses to determine which compact transactions in the dtxn can be completed, and which cannot complete because some of the inputs are unavailable or already locked. The coordinator makes this decision durable and then contacts each shard again to indicate which compact transactions in the dtxn to complete and which to cancel (a *commit* request). Each shard then atomically unlocks the input UHS IDs belonging to a canceled transaction, deletes input UHS IDs and creates the output UHS IDs for successful transactions, and updates local dtxn state about the status of the dtxn. The shard then responds to the coordinator to indicate that the *commit* was successful (6).
3. **Discard (7).** The coordinator issues a *discard* to each shard informing them that the dtxn is now complete and it can forget the relevant dtxn state.

Once every shard participating in the dtxn has completed the *commit*, the coordinator informs each sentinel whether its transactions were successfully executed or rejected by the shards (8). The sentinels in turn forward these responses to the users who submitted the transactions (9).

It is possible that if two concurrent transactions by different transaction coordinators spend the same inputs, neither will succeed, because both will be canceled due to observing the other's lock conflicts. This means that at least one will need to be retried, which is left to the user's wallet. An adversary could try to continually conflict a user's transaction by spending the same input. However,

this requires the adversary to have the authorization to spend the input in order to pass sentinel validation. Investigating methods to fairly resolve concurrency conflicts is left to future work.

Combining many compact transactions into a larger distributed transaction amortizes the costs of messaging and making the result of each phase of the protocol durable on each shard, whether by flushing to persistent storage or replicating as part of a distributed state machine. Because our application semantics are constrained, this is slightly different from traditional two-phase commit in that dtxns always complete successfully, and individual compact transactions are executed (or not) deterministically: if all of a compact transaction's input UHS IDs are locked and output UHS IDs are reserved, the compact transaction will succeed. The transaction coordinator always completes both phases of dtxns, even if some of the compact transactions within do not succeed. General 2PC designs need to support transaction coordinators that might make arbitrary decisions about whether to commit or abort transactions.

4.2 Fault Tolerance

Each transaction coordinator and shard is made fault tolerant using Raft [58], a distributed consensus algorithm. Sentinels only maintain state during the duration of the user wallet request to return transaction status to the user; if one fails, the user's wallet will need to retry its transaction or ask about its status with another sentinel.

Only the leader node in the transaction coordinator Raft cluster actively processes dtxns; followers simply replicate the inputs to each phase of the dtxn. This is a technique used in deterministic scalable database systems [68]. Before initiating each phase of the distributed transaction, the coordinator replicates the inputs to both the *prepare* and *commit* requests to each shard. Shards remember which phase each dtxn has last executed and the response to the coordinator. If the coordinator leader changes mid-dtxn, the new leader reads the list of active dtxns from the coordinator state machine and continues each dtxn from the start of its most recent phase. Shards that have already completed the requests will return the stored response to the new coordinator leader. To ensure proper completion of the *commit* across all shards, shards will remember the response for the *commit* until the coordinator has received responses from all shards in the dtxn and issued a *discard* to inform shards the dtxn is complete and can be forgotten. Note that these can be applied lazily and the transaction coordinators can inform the sentinels the transactions were successful before issuing *discard*.

Similar to coordinators, in each shard cluster only the leader processes dtxns and responds to sentinels. Although followers do not handle RPCs, they maintain the same UHS as the shard leader, so they are prepared to take over processing RPCs if the leader fails without a specific recovery procedure beyond that provided by Raft. Once a dtxn has entered the *prepare* phase and has been replicated by the coordinator cluster, the dtxn will always run to completion. If a shard leader fails mid-transaction, the coordinator leader will retry requests until a new shard leader processes and responds to the request.

4.3 Comparison to blockchain architectures

Many have suggested using blockchain technology to design a central bank digital currency. We found that using a blockchain-based system in its entirety was not a good match for our requirements. First, there was no requirement to distribute governance amongst a set of distrusting participants. The transaction processing platform is controlled and governed by a central administrator. Blockchains use relatively new distributed consensus protocols which are designed to operate in a permissionless, adversarial environment. This introduces software and operational complexity as well as new cryptographic assumptions. A CBDC should rely on the simplest, most well-understood, well-tested protocols to achieve its goals.

Second, we anticipated the complexity of a blockchain architecture would limit performance. To evaluate this we implemented a streamlined permissioned-blockchain-inspired design, the *atomizer*. Instead of using transaction coordinators and two-phase commit, the atomizer orders compact transactions into blocks through a replicated state machine. To reduce load on this ordering server, the design outsources storage of the UHS to shards, which hold the partitioned UHS ID state as in the 2PC design. However, a shard's UHS ID state is only correct up to a specific block height. Sentinels send compact transactions to shards, and shards then pass attestations for input IDs that exist to the atomizer, which collects complete compact transactions into blocks. The atomizer broadcasts confirmed blocks so shards can update their state, deleting spent UHS IDs and creating new ones. Shards do not require consensus or even primary/backup for correctness, they are merely replicated. For the specifics of the atomizer design see a related technical report [51]. As might be anticipated, we found the atomizer architecture's throughput is limited by the resource constraints (network bandwidth and CPU) of a single server, the atomizer leader, and cannot benefit beyond a limited point from additional shard resources (§6).

OmniLedger [49] is a sharded blockchain design which, like Hamilton, operates in the UTXO model, but uses a client-driven commit protocol to commit cross-shard transactions without going through a single server. Based on reported results, OmniLedger can achieve much higher performance than our atomizer prototype (with a replication factor of four per shard and 1% adversarial power, approximately 400K txns/sec). However, in Hamilton we

did not want to rely on clients, which might fail or disappear, to complete transactions. Hamilton instead uses replicated transaction coordinators, so incomplete transactions will never result in stuck or frozen funds.

5 Implementation

We implemented Hamilton in C++17, released at [https://github.com/mit-dci/opencbdc-tx], and tested on Linux and macOS, though it should be portable to any UNIX-like system. The primary dependence on a UNIX-compatible API is our use of UNIX sockets for network communication. Clients communicate via a custom serialization protocol, via single, short-lived TCP connections.

We use LevelDB [43], NuRaft [35], libsecp256k1 [13] and vendored components from Bitcoin Core [12]. We use BIP-340 compatible Schnorr signatures [76] as our digital signature scheme. We also use the cryptography components of Bitcoin Core to provide optimized implementations of SHA256 [56], used as our cryptographic hash function, SipHash [4] used for hashmaps and bech32 [74, 75] used for error-correcting public key encoding.

6 Evaluation

Our evaluation answers the following questions:

- How does Hamilton's performance compare to other database and blockchain-based designs?
- How does Hamilton perform with multiple regional failures?
- How well does Hamilton tolerate different transactional workloads, whether with larger transactions or double spends?

Setup. Unless otherwise specified, for benchmarking we deployed on Amazon Web Services (AWS) using EC2 virtual servers running Ubuntu 20.04 (c5n.2xlarge instances with 8 vCPUs and 21GB RAM). We ran the system components across three regions within the United States: Virginia, Ohio and Oregon. Round-trip time between Virginia and Ohio was $\approx$ 12ms, Virginia to Oregon $\approx$ 62ms and Ohio to Oregon $\approx$ 51ms. Unless otherwise stated, there were 1B outputs, 8 logical shards, and shard and coordinator clusters were replicated by a factor of three. In particular, each shard and each coordinator has a Raft node in each of these three regions. Non-replicated components such as sentinels and load generators were distributed between regions to simulate load from across the United States. Load generators (c5n.large instances with 2 vCPUs and 5.25GB RAM) were simulated wallets that produced valid, signed transactions with two inputs and two outputs unless otherwise stated. We limited dtxns to a maximum size of 2000 compact transactions, and each coordinator had a thread pool containing 75 threads.

We do not consider data from a benchmark for a configuration if any Raft cluster was unable to reliably replicate data between *all* regions during the experiment. Take, for example, a three node cluster: if one follower is reliably lagging behind due to data replication issues, though the leader and other follower still form a quorum, this configuration can't tolerate an additional failure of either node without potentially suffering a delay and throughput reduction. We took this to imply the system was oversaturated.

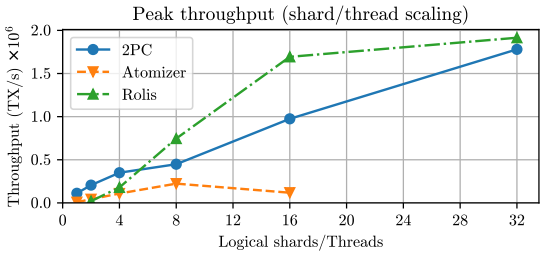


Figure 4: Compares the peak throughput of 2PC, the atomizer, and Rolis, when varying logical shard count (2PC and atomizer) or the number of threads (Rolis).

6.1 Comparison

Figure 4 shows the peak throughput for Hamilton and the atomizer when varying the number of load generators for different shard counts. Our 2PC architecture scales linearly as the number of logical shards increases, up to 1.7M txns/sec with less than one second 99% tail latency and under 500ms 50% latency, though we expect peak throughput would continue to increase with more shards and this would not negatively affect latency. Additionally, if a lower tail latency is desired for a particular transaction throughput, increasing the number of shards can decrease tail latency for the same offered load. This makes sense because, in the worst case, each transaction requires the participation of a subset of shards equal to the number of inputs and outputs in the transaction. Since transactions in the test load have an upper bounded number of inputs and outputs, increasing the number of shards results in each transaction requiring the participation of a smaller proportion of the total shards in the system.

The atomizer achieves 170K txns/sec with under two seconds 99% tail latency and 700ms 50% latency, the bottleneck being network bandwidth limitations between the replicas in different regions. In other experiments we found if bandwidth constraints are relaxed, computation in the lead atomizer replica to manage Raft replication and execute the state machine becomes the bottleneck.

We compare Hamilton's performance to three centralized databases: PostgreSQL, Rolis [64], and CockroachDB. In all cases the workload was the `swap` function with 2-in/2-out compact transactions; we did not run sentinels or do signature checking, which improved latency for these measurements.

PostgreSQL. We chose to compare against PostgreSQL

since it is a widely used, fully-featured commercial database. We ran PostgreSQL on c5a.8xlarge EC2 instances, using benchmarking recommendations [50] and implementing `swap` as a stored function. We were able to obtain a max throughput on PostgreSQL v13.8 of 63.4K txns/sec and an average throughput over a 45 second run of 61K txns/sec, with average and 99% latency (as measured on the client) of 10ms, using hundreds of load generating clients. The database had 38.4M UHS IDs. Our experiments indicated that PostgreSQL is limited by throughput to the write-ahead log. Note that this was a single-machine benchmark with no replication.

Rolis. Rolis is a very high-performance in-memory research database. To evaluate Rolis we implemented the `swap` function in the benchmarking experiment software that accompanies Rolis. We used three replicas in the regions specified above and, as Rolis uses significant amount of RAM, used c5n.18xlarge instances (72 vCPUs, 192 GB RAM), largest among the c5n family of instances that we used to evaluate Hamilton. We did our experiments using Ubuntu 18.04.

We used a database containing 200M UHS IDs, unlike the 1B used for benchmarking Hamilton as 250M and larger data base sizes reliably ran out of memory. Figure 4 shows the performance as we increase the number of threads; the thread count there includes the additional thread [64, §6.1] that Rolis uses to advance the watermark and perform leader election tasks. To maximize Rolis's performance, we (a) implemented load generators inside each Rolis thread (so unlike when evaluating Hamilton, PostgreSQL, or CockroachDB, the load generators were not networked) and (b) used sequential UHS IDs. The latter avoids calling a hash function to generate UHS IDs in the critical path of the load generator, which halved the throughput when we tried it, but would be parallelizable in a different load generator implementation.

Rolis achieves a max throughput of 1.91M txns/sec on our workload. When replicating the YCSB++ benchmark on the same EC2 instances, the max throughput using 32 threads was 10.2M txns/sec, comparable to 10.3M reported in the Rolis paper. Our keys are 32 bytes instead of 8 bytes, and unlike YCSB++, which is 50% read-only, our workload is 100% read/modify/write, thus it requires more logic per transaction and more bandwidth for replication.

The almost-linear scalability when using 1, 2, 4, 8, and 16 threads did not continue when thread count increased from 16 to 31. We attribute this to hyper-threading: Rolis's implementation only supports a single socket (as it uses `rdtscp` counters for time-stamping), whereas our 72 vCPU instance had 16 cores (32 hyper-threads) per socket. This made our 31 thread benchmarks use hyper-threading and we would expect Rolis to perform better with an increased number of dedicated cores.

When comparing performance, it is important to note that Hamilton durably commits every transaction to disk, whereas Rolis runs wholly in-memory persisting nothing to disk. As our setting calls for ability to cold-start a system after potentially correlated failures, we see the excellent Rolis in-memory performance as establishing an upper, rather than a lower, bound for a backend.

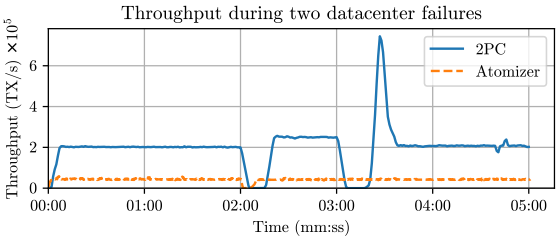


Figure 5: Throughput over time where the number of supported failures for both architectures was 2 and 2 whole data center failures were triggered at 120s and 180s. 5 sample moving average.

CockroachDB. We ran limited experiments against CockroachDB v22.1.9, a feature-complete scalable distributed database. CockroachDB automatically manages data partition assignments. We ran with a replication factor of three in the regions specified above, using c5d.4xlarge instances. We had to implement the `swap` function through client queries, requiring two roundtrips to commit a transaction, as CockroachDB does not yet support stored procedures or functions. It achieved throughput of 3.4K, 5.7K, and 11K txns/sec partitioning data across 1, 2, and 4 logical shards, respectively. CockroachDB is slower because it implements many more features than Hamilton, whereas implementing `swap` as a primitive in our distributed backend reduces the number of roundtrips and transferred data required to commit.

In summary, Hamilton achieves higher throughput than PostgreSQL and CockroachDB by co-designing the application with the data model, and pushing the `swap` primitive into the commit protocol. It approaches the performance of Rolis, a very fast in-memory replicated database. Rolis might be a better choice for a backend if in-memory replication is sufficient for durability; in our application it was important to have data on disk to have a path to recovery from simultaneous crashes. Another reasonable choice is to use an existing commercial database if performance is not as much of a concern.

6.2 Fault Tolerance

We evaluate how our system handles up to two regional data center failures, and its scalability as the number of supported faults increases.

Figure 5 shows the overall system throughput over time when shards and coordinators have a replication factor of five (supporting up to two failures per cluster). To test continued system availability when up to two data centers fail

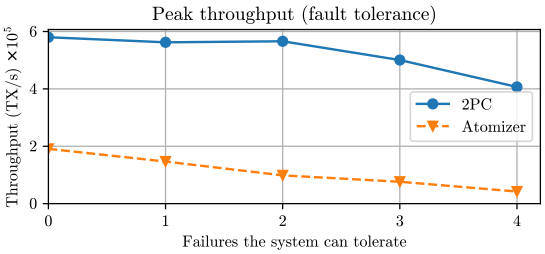


Figure 6: Peak throughput versus the number of tolerated failures per replicated service. Atomizer used 100M UHS IDs, 2PC used 1B.

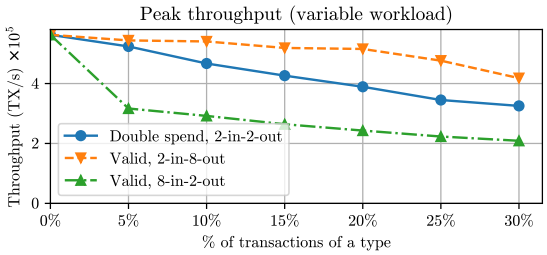


Figure 7: Peak throughput varying the proportion of valid, 2-in-2-out transactions.

completely, the Raft leaders for coordinators and shards were killed at 120 seconds into the test, and a subsequent set of nodes for each cluster were killed at 180 seconds into the test. The second group of nodes contained a mixture of leaders and followers depending on the result of the leader election from the previous failure. The plot shows that our system is successfully able to recover from the failure of two entire data centers with minimal downtime and no loss of system performance. For each failure, throughput was temporarily reduced for less than fifteen seconds, before automatically recovering to the baseline. There is no data loss and the system is not left in an inconsistent state as the replacement coordinators complete any distributed transactions that were in progress at the time of each failure.

Figure 6 compares how peak throughput is affected by the number of supported system failures between architectures by increasing the number of tolerated failures from 0 through 4. The plot shows that as replication factor increases, peak throughput for a given system configuration decreases. Since the performance of our Raft replicated services is limited by bandwidth constraints between the leader and follower nodes, more replicas require more leader bandwidth to provide the same throughput. In 2PC, we believe a higher replication factor can be supported without a loss in performance by increasing the number of shard and coordinator clusters. The atomizer architecture could not scale in this way, as the system throughput is limited by a single Raft service which provides the global order of transactions.

6.3 Workload Variability

We varied the proportion of transactions with a high number of inputs and outputs, and the proportion of double-spending transactions to see how Hamilton performs under different workloads from users.

Figure 7 shows how the proportion of double-spending transactions, or those with a large number of inputs and outputs affects peak throughput. In this test, the proportion of invalid or non-2-in-2-out transactions in the workload was varied from 0% through 30%. The load generators sent double-spending transactions by storing previously confirmed transactions and re-issuing them at a later time. We only plot the throughput of valid transactions because double-spending transactions never complete.

As the proportion of large or double-spending transactions increases, the peak throughput decreases. This behavior is similar to increasing the overall number of 2-in-2-out transactions. The system is limited by the overall number of UHS IDs being processed, regardless of how they are grouped into transactions. Shards and coordinators replicate all transactions, so double-spends exert the same load as valid transactions. Thus, increasing the number of shards and coordinators could absorb an increased proportion of large or double-spending transactions while executing the same number of valid transactions. Transactions with a large number of inputs most negatively affect peak throughput because the sentinels have to validate more signatures per transaction. This could be solved by increasing the number of sentinels per load generator.

7 Related Work

Central banks are experimenting with or launching CBDCs. Some [22, 23, 34, 65] use DLT [57, 63], but according to their reports do not achieve as high performance as Hamilton. Other CBDC work uses a parallelized architecture; China's e-CNY is currently in public trials [24, 46, 62] and is a scalable system based on the UTXO model, but does not support self-custody. The Eurosystem has tested a CBDC design based on tracking groups of bills using a set of parallelized blockchains [38]. While it achieves linear scalability, transactions involving multiple bills require external coordination. The Regulated Liability Network [29] presents a design which claims to achieve 1M transactions per second with multiple coordinated blockchains. However, they do not discuss deployment across multiple geographic regions which is vital for resiliency or provide latency measurements.

Chaumian eCash [26] and designs based on it [18, 20, 21] operate with a central trusted intermediary, but either require maintaining an ever-growing list of all spent coins for double spend prevention, or require users to manage expiring coins, which has significant policy implications. The Swiss National Bank [27] expands upon the Chau-

mian ecash model using this technique.

Several central banks already support real-time gross settlement (RTGS) and fast payment systems [5]. These systems are designed to settle transactions between only eligible financial institutions. In practice, these systems do not handle a volume of traffic representative of a national retail payment system, do not provide programmability, nor provide access to the general public [6, 39, 40, 59].

Contrary to decentralized cryptocurrencies [53, 73] or stablecoins [11, 25, 33, 67], Hamilton is designed to be administered directly by the central bank or a related entity, and transacts in central bank liabilities. However, Hamilton borrows ideas from cryptocurrency designs; it uses the UTXO transaction model, but only stores 32-byte hashes [41]. Users cannot verify transaction execution themselves since they cannot access the ledger or state of the system. Techniques like authenticated datastructures [66] or cryptographic proofs of transaction inclusion [55] might be able to help with this.

Newer consensus algorithms [30] achieve higher throughput for agreement on ordering, and Hamilton could benefit from these as a replacement for Raft. However, faster consensus does not address the scalability bottleneck of state machine execution and validation in non-sharded blockchain-based architectures. Our 2PC architecture outperforms both our straw-man sharded blockchain as well as existing sharded blockchains which aim to operate in a decentralized setting, and thus cannot rely on a trusted coordinator to drive the cross-shard commit protocol to completion [2, 31, 49, 77]. However, some of these blockchains provide more features than Hamilton, like general smart contracts.

Via careful choices in application transaction design and format, Hamilton is able to avoid the need for reads or any other transaction execution before commit time, and can apply good ideas in traditional distributed transaction commit protocols [1, 28, 68] in a simplified backend that does not need to handle general transactions.

8 Conclusion

This work presents a high-performance, resilient transaction processor for CBDCs. We support a range of potential policy choices and can minimize data stored in the transaction processor while supporting a variety of custodial models. Our experiments show that a blockchain-based design for CBDC has seriously scalability limitations, but by validating transactions in parallel we can achieve millions of transactions per second.

9 Acknowledgements

The authors express gratitude to Robert Bench and Jim Cunha for their leadership and direction in this work. We are also grateful to Tyler Frederick and David Urness who were instrumental in the early stages of this research. In addition, we thank Chris Berube, Alexander Chernyakhovsky, Nikhil George, Gert-Jaap Glasbergen, Alistair Hughes, Ben Kincaid, Weihai Shen, Bernard Snowden, Sam Stuewe, and our shepherd and reviewers for their helpful contributions, feedback, and comments. Cory Fields, Neha Narula, and Madars Virza were supported by the funders of the Digital Currency Initiative.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):159–174, 2007.

[2] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[3] R. Auer, J. Frost, M. Lee, A. Martin, and N. Narula. Why central bank digital currencies? Liberty Street Economics, 2021. https://libertystreeteconomics.newyorkfed.org/2021/12/why-central-bank-digital-currencies/.

[4] J. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. Cryptology ePrint Archive, Report 2012/351, 2012. https://eprint.iacr.org/2012/351.

[5] Bank For International Settlements. Fast payments - enhancing the speed and availability of retail payments. Committee on Payments and Market Infrastructures, 2016. https://www.bis.org/cpmi/publ/d154.pdf.

[6] Bank for International Settlements. BIS statistics explorer, 2019. https://stats.bis.org/statx/toc/CPMI.html.

[7] Bank of Canada et al. Central bank digital currencies: foundational principles and core features. BIS Working Group, 2020. https://www.bis.org/publ/othp33.pdf.

[8] Bank of England. Central bank digital currency: Opportunities, challenges and design, 2020. https://www.bankofengland.co.uk/-/media/boe/files/paper/2020/central-bank-digital-currency-opportunities-challenges-and-design.pdf.

[9] M. L. Bech and R. Garratt. Central bank digital currencies. BIS Quarterly Review, September 2017.

[10] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[11] Binance. Binance USD. https://www.binance.com/en/busd.

[12] Bitcoin Core Developers. Bitcoin Core. https://github.com/bitcoin/bitcoin.

[13] Bitcoin Core Developers. libsecp256k1. https://github.com/bitcoin-core/secp256k1.

[14] C. Boar and A. Wehrli. Ready, steady, go? – results of the third BIS survey on central bank digital currency. *BIS Papers No 114*, 2021. https://www.bis.org/publ/bppdf/bispap114.htm.

[15] Board of Governors of the Federal Reserve System. Money and payments: The U.S. dollar in the age of digital transformation, January 2022.

[16] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. Zexe: Enabling decentralized private computation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, S&P '20, 2020. ePrint: https://eprint.iacr.org/2018/962.

[17] L. Brainard. Update on digital currencies, stablecoins, and the challenges ahead, 2019. https://www.federalreserve.gov/newsevents/speech/brainard20191218a.htm.

[18] S. Brands. Untraceable off-line cash in wallet with observers. In *Annual international cryptology conference*, pages 302–318. Springer, 1993.

[19] N. Brewster and S. Bishop. Getting out the message. http://www.centralbank.org.bb/_economic-insightbb/getting-out-the-message.

[20] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 302–321. Springer, 2005.

[21] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Balancing accountability and privacy using e-cash. In *International Conference on Security and Cryptography for Networks*, pages 141–155. Springer, 2006.

[22] Central Bank of Nigeria. Design paper for the eNaira. https://enaira.gov.ng/download/eNaira_Design_Paper.pdf.

[23] Central Bank of The Bahamas. Sand dollar. https://www.sanddollar.bs.

[24] Central Banking Newsdesk, 2020. https://www.centralbanking.com/fintech/cbdc/7529621/pboc-confirms-digital-currency-pilot.

[25] Centre Foundation. USD-C. https://www.centre.io/usdc.

[26] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of Crypto 82*, pages 199–203. Springer, 1983.

[27] D. Chaum, C. Grothoff, and T. Moser. How to issue a central bank digital currency. *arXiv preprint arXiv:2103.00254*, 2021.

[28] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[29] A. Culligan, N. Pennington, M. Delatine, P. Morel, E. M. Salinas, G. Vargas, N. Dusane, J. Iu, S. Sheikh, N. Kerigan, T. McLaughlin, P. D. Courcy, M. Low, and K. H. Park. The regulated liability network, 12 2021. https://setldevelopmentltd.box.com/shared/static/18mff2m990qabgzseiex3h7itq7qdnls.pdf.

[30] G. Danezis, E. K. Kogias, A. Sonnino, and A. Spiegelman. Narwal and Tusk: A DAG-based mempool and efficient BFT consensus, 2021. https://arxiv.org/pdf/2105.11827.pdf.

[31] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[32] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and MtGox. In *Proceedings of the 19th European Symposium on Research in Computer Security*, pages 313–326, 2014.

[33] Diem Foundation. Diem. https://www.diem.com/en-us/white-paper/.

[34] Eastern Caribbean Central Bank. ECCB digital EC currency pilot, 2021. https://www.eccb-centralbank.org/p/about-the-project.

[35] eBay. NuRaft. https://github.com/eBay/NuRaft.

[36] K. Eswaran, J. Gray, and L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, november 1976.

[37] European Central Bank. ECB publishes the results of the public consultation on a digital euro, 2021. https://www.ecb.europa.eu/press/pr/date/2021/html/ecb.pr210414~ca3013c852.en.html.

[38] European Central Bank. Work stream 3: A new solution – blockchain & eID, 2021. https://haldus.eestipank.ee/sites/default/files/2021-07/Work%20stream%203%20-%20A%20New%20Solution%20-%20Blockchain%20and%20eID_1.pdf.

[39] Eurosystem. TARGET Instant Payments Settlement user requirements, 2017. https://www.ecb.europa.eu/paym/target/tips/profuse/shared/pdf/tips_crdm_uhb_v1.0.0.pdf.

[40] Eurosystem. T2-T2S consolidation user requirements document for T2-RTGS component, 2018. https://www.ecb.europa.eu/paym/pdf/consultations/T2-T2S_Consolidation_User_Requirements_Document_T2_RTGS_v1.2_CLEAN.pdf.

[41] C. Fields. UHS: Full-node security without maintaining a full UTXO set. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-May/015967.html.

[42] R. Garratt, M. J. Lee, et al. Monetizing privacy with central bank digital currencies. Technical report, Federal Reserve Bank of New York, 2020.

[43] Google. LevelDB. https://github.com/google/leveldb.

[44] J. N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, pages 394–481. Springer, 1978.

[45] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specifiation, 2021. https://zips.z.cash/protocol/protocol.pdf.

[46] J. C. Jiang and K. Lucero. Background and implications of China's central bank digital currency: E-CNY. *Available at SSRN 3774479*, 2021.

[47] J. Kiff, J. Alwazir, S. Davidovic, A. Farias, A. Khan, T. Khiaonarong, M. Malaika, H. Monroe, N. Sugimoto, H. Tourpe, and P. Zhou. A survey of research on retail central bank digital currency, 2020. https://www.elibrary.imf.org/view/journals/001/2020/104/001.2020.issue-104-en.xml.

[48] koe, K. M. Alonso, and S. Noether. Zero to Monero: Second edition, 2020. https://www.getmonero.org/library/Zero-to-Monero-2-0-0.pdf.

[49] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[50] V. Kumar. Pgbench: Performance benchmark of postgresql 12 and edb advanced server 12, 2020. https://www.enterprisedb.com/blog/pgbench-performance-benchmark-postgresql-12-and-edb-advanced-server-12.

[51] J. Lovejoy, C. Fields, M. Virza, T. Frederick, D. Urness, K. Karwaski, A. Brownworth, and N. Narula. A high performance payment processing system designed for central bank digital currencies. *Cryptology ePrint Archive*, 2022.

[52] G. Maxwell. Confidential transactions – investigation. https://elementsproject.org/features/confidential-transactions/investigation.

[53] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 10 2008. https://bitcoin.org/bitcoin.pdf.

[54] N. Narula, W. Vasquez, and M. Virza. zkLedger: Privacy-preserving auditing for distributed ledgers. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, 2018. ePrint: https://eprint.iacr.org/2018/241.

[55] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security '17)*, pages 1271–1287, 2017.

[56] NIST. Secure Hash Standard, 2002. https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf.

[57] NZIA. Nzia cortex dlt. https://nzia.io.

[58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 305–319, 2014.

[59] Pay.UK. Pay.UK 2020 annual self-assessment against the principles for financial market infrastructure, 2020. https://www.wearepay.uk/wp-content/uploads/Pay.UK-PFMI-Self-Assessment-Jun-20.pdf.

[60] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '91, pages 129–140, 1992.

[61] A. Pertsev, R. Semenov, and R. Storm. Tornado cash privacy solution: Version 1.4, 2019. https://tornado.cash/Tornado.cash_whitepaper_v1.4.pdf.

[62] Y. Qian. Technical aspects of CBDC in a two-tiered system, 2018. https://www.itu.int/en/ITU-T/Workshops-and-Seminars/20180718/Documents/Yao%20Qian.pdf.

[63] R3. Corda. https://www.corda.net.

[64] W. Shen, A. Khanna, S. Angel, S. Sen, and S. Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 69–84, 2022.

[65] Sveriges Riksbank. E-krona pilot phase 1. *Sveriges Riksbank Report*, 2021. https://www.riksbank.se/globalassets/media/rapporter/e-krona/2021/e-krona-pilot-phase-1.pdf.

[66] R. Tamassia. Authenticated data structures. In *European symposium on algorithms*, pages 2–5. Springer, 2003.

[67] Tether Operations Ltd. Tether. https://tether.to/.

[68] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[69] UkoeHB. Mechanics of MobileCoin. https://github.com/UkoeHB/Mechanics-of-MobileCoin.

[70] A. Usher, E. Reshidi, F. Rivadeneyra, S. Hendry, et al. The positive case for a CBDC. *Bank of Canada Staff Discussion Paper*, 2021.

[71] N. van Saberhagen. CryptoNote v 2.0. https://web.archive.org/web/20201028121818/https://cryptonote.org/whitepaper.pdf.

[72] T. Walton-Pocock. Why hashes dominate in SNARKs: A primer by AZTEC, 2019. https://medium.com/aztec-protocol/why-hashes-dominate-in-snarks-b20a555f074c.

[73] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[74] P. Wuille. Bech32m format for v1+ witness addresses, 2020. https://github.com/bitcoin/bips/blob/master/bip-0350.mediawiki.

[75] P. Wuille and G. Maxwell. Base32 address format for native v0-16 witness outputs, 2017. https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki.

[76] P. Wuille, J. Nick, and T. Ruffing. Schnorr signatures for secp256k1, 2020. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki.

[77] M. Zamani, M. Movahedi, and M. Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics

Mehrdad Khani[1,2], Ganesh Ananthanarayanan[2], Kevin Hsieh[2], Junchen Jiang[3], Ravi Netravali[4], Yuanchao Shu[5], Mohammad Alizadeh[1], Victor Bahl[2]

[1]*MIT CSAIL,* [2]*Microsoft,* [3]*University of Chicago,* [4]*Princeton University,* [5]*Zhejiang University*

Abstract

Continuous learning has recently shown promising results for video analytics by adapting a lightweight "expert" DNN model for each specific video scene to cope with the data drift in real time. However, current adaptation approaches either rely on periodic retraining and suffer its delay and significant compute costs or rely on selecting historical models and incur accuracy loss by not fully leveraging the potential of persistent retraining. Without dynamically optimizing the resource sharing among model selection and retraining, both approaches have a diminishing return at scale. RECL is a new video-analytics framework that carefully integrates model reusing and online model retraining, allowing it to quickly adapt the expert model given any video frame samples. To do this, RECL *(i)* shares across edge devices a (potentially growing) "model zoo" that comprises expert models previously trained for all edge devices, enabling history model reuse across video sessions, *(ii)* uses a fast procedure to online select a highly accurate expert model from this shared model zoo, and *(iii)* dynamically optimizes GPU allocation among model retraining, model selection, and timely updates of the model zoo. Our evaluation of RECL over 70 hours of real-world videos across two vision tasks (object detection and classification) shows substantial performance gains compared to prior work, further amplifying over the system lifetime.

1 Introduction

Video analytics with deep neural networks (DNNs) is a promising technology adopted in a wide range of applications such as enterprise security, retail, traffic management, and transportation [1, 2]. Across these applications, it is often imperative to run analytics tasks directly on edge devices (e.g., using on-premises edge servers [3, 4]) to ensure that the system can deliver real-time results with low latency and in compliance with data privacy constraints [5–8]. However, the edge has limited compute resources, which cannot match the unrelenting growth of video analytics workloads, DNN models, and video streams [9, 10]. Even for applications that can be deployed in resourceful environments such as public clouds, the cost of running video analytics remains exorbitant despite recent advancements in DNN resource efficiency [11–13]. For example, a high-end NVIDIA V100 GPU can only support two video streams running the state-of-the-art YOLOv5-L model [13] at 30 FPS, which translates to a steep cost of $1,100/month/stream on public clouds [14].

One common approach to reducing the resource requirements for video analytics is to use specialized and compressed DNNs [15–18]. However, owing to their inherent limits on the number of object appearances and scenes they can learn in their condensed structures, such specialized DNNs require *continuous retraining* to cope with dynamic scenes (data drifts) in order to maintain high inference accuracy. Recent work in the computer vision and systems communities [19–21] has shown the effectiveness of this approach for edge video analytics, delivering both high resource efficiency and accuracy in results.

Though promising, continuous retraining and deploying specialized DNNs has two fundamental limitations. First, continuous retraining consumes the vast majority of *compute resources* in these video analytics systems (70%–90% in our study) [20, 21], making model retraining the key bottleneck in scaling video analytics to more video streams with limited compute resources. Our study (Fig. 2) shows that accuracy drops sharply (by 40% in object detection) as 4× more cameras share the GPU cycles to retrain their models (§2.2). Second, it takes *time* to retrain specialized DNNs, and abrupt video scene changes inevitably lead to drastic accuracy drops until the retraining is completed (see Fig. 3 for an example). Hence, it is fundamentally challenging to uphold the accuracy lower tail during the retraining.

Our goal in this work is to address the above two fundamental limitations so that video analytics are scalable with more consistent accuracy. As retraining specialized DNNs requires resources and takes time, we aim to minimize the necessity of retraining by judiciously *reusing* historical specialized DNNs that are trained with past video segments. The intuition behind our approach is that video streams typically exhibit spatio-temporal correlations (e.g., a car drives back on the same street or another car has been on the same street before) [22]. Thus, it is likely that the current video segment bears some resemblance to historical video segments, and the corresponding historical specialized DNNs can be *reused* for the current scene. Indeed, our study in §2 shows that an idealized model reusing scheme can consistently deliver high accuracy (35% mAP) with limited compute resources. In comparison, existing continuous retraining systems (e.g., [20]) cannot keep up with the compute demand of more cameras, with their accuracy dropping to a low 24% mAP.

Technical challenges: Harnessing the potential of model reuse for video analytics faces two challenges. First, we need to quickly and accurately find the specialized DNN that works well for the current video segment so that we can reuse the DNN in real-time. This is difficult because it is unclear how to compare the similarity of high-dimensional and unstructured data such as video segments [23], and comparing the current

video segment with all the historical video segments is not practically feasible. Second, we need to keep the cost of enabling model reuse much lower than the cost of model retraining. This is challenging as the cost of seeking through historical models grows with the size of history, while model retraining only requires fixed expenditure for each video segment. Recent video analytics solutions that reuse historical models (e.g., ODIN [23]) cannot address these challenges because they are not designed for resource efficiency.

Solution: We present RECL, a new video analytics solution that leverages historical specialized DNNs to improve scalability, responsiveness, and accuracy consistency in a resource-constrained environment. RECL is the first end-to-end system that integrates model reusing with model retraining for resource-efficient video analytics, entailing three main ideas:

- We design a *fast and robust model selection* procedure to quickly select a suitable model from the *model zoo*, a large collection of historical specialized models (§3.1). Our model selector is inspired by sparse gating networks in the mixture of experts (MoEs) approach [24–26], and we make it resource-efficient by decoupling the training of the gating network from the training of underlying experts. This allows RECL to select a model based on the characteristic of video analytic tasks and video scenes (e.g., detecting cars on a sunny day), which is superior to existing solutions that only consider the similarity of video frames (e.g., rainy or sunny days) [23].
- RECL *shares* the model zoo across different edge devices to enable more *model reusing* and dynamically adds new experts to the model zoo with a lightweight process to update the model selector (§3.3).
- RECL shares GPU resources across the retraining jobs using an *iterative training scheduler* that dynamically prioritizes retraining jobs that progress faster (§3.2). As a result, it spends little retraining resources on expert models that are already a good match with the current video segments.

We implement and evaluate RECL on two computer vision tasks: object detection and object classification. We compare RECL against three state-of-the-art video analytics systems (Ekya [21], AMS [20], and ODIN [23]) over a total of 71 hours of driving videos. Given the same compute resource, our evaluation shows that RECL improves the object detection mAP and image classification accuracy over the state-of-the-art solutions by up to 9.0% and 7.4%, respectively. To put these accuracy gains in perspective, the state-of-the-art mAP score for the object detection task on the PASCAL dataset has only improved by less than 8 percent in the past 6 years [27]. Moreover, the baseline systems need at least 3.2× more compute resources to match RECL's accuracy. Our ablation study shows that RECL's superior performance mostly comes from effective integration of model reuse in our design. Compared to Ekya as a prior continuous training approach, RECL achieves the same accuracy up to 91 seconds faster on average. We also show that the compute overhead of RECL declines gracefully over time as more expert models are learned and added to the model zoo.

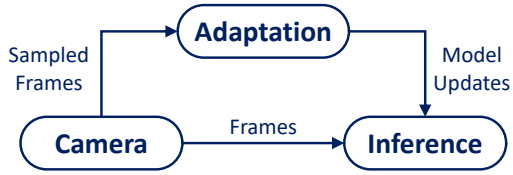


Figure 1: Overview of a video analytics system utilizing continuous learning. A typical adaptation module continuously retrains expert models or selects them from an existing collection of models trained in the past.

2 Background and Motivation

We first introduce the background of continuous retraining and deploying specialized DNNs for video analytics (§2.1). We then discuss the fundamental limitations of this approach and how reusing historical specialized DNNs can address these limitations effectively (§2.2).

2.1 Continuous Retraining for Video Analytics

State-of-the-art generic DNNs are often too expensive to run for video analytics all the time in resource-constrained environments such as a mobile edge computing (MEC) network [28]. A common approach is to deploy specialized and compressed DNNs (or "expert" models) that are trained using the knowledge of the generic and expensive DNNs (or the "teacher" model). The idea is to use knowledge distillation [29] to transfer the knowledge from a large teacher model to a small expert model for a specific video segment or video stream. On a matching video segment, an expert model can save compute resources by orders of magnitude while achieving similar model accuracy as the large teacher model [15, 16, 30]. This approach has been widely adopted in modern systems such as Microsoft's Rocket [17] and Google's Learn2Compress [18].

As an expert model only recognizes a limited set of object appearances and video scenes, a static expert model cannot achieve high accuracy on dynamic live videos where objects and scenes inevitably change over time (e.g., different locations, lighting conditions, object classes, etc.) [21]. A promising approach to employing expert models on dynamic live videos is to *continuously retrain* the expert model with the most recent video frames. Recent work [19–21, 31] has established that continuous retraining and deploying small expert models can simultaneously achieve high accuracy and resource efficiency on dynamic video content. Furthermore, continuous retraining has shown superior performance compared to running the large teacher model on a subset of frames and interpolating the labels (e.g., using optical flow tracking methods) [20].

Fig. 1 can be used to illustrate the high-level components of a video analytics system that continuously retrains and deploys expert models. They include: (i) *camera service:* periodically sends new sample video frames to the adaptation service; (ii) *adaptation service:* uses the recently sampled frames to fine-tune (a copy of) the camera's expert model to mimic a larger teacher model for the current scene, and sends (or "streams") the updated expert model to the inference service; and (iii) *in-*

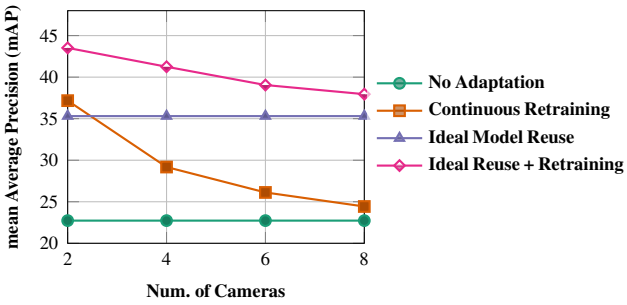


Figure 2: Object detection accuracy (mAP) of different designs under different numbers of cameras. Model reuse has the potential to significantly improve the accuracy in resource-constrained regimes (4, 6, and 8 cameras), and when combining model reuse and model retraining, performance *could* be greatly improved.

ference service: uses the received lightweight expert model for real-time inference on video frames from the camera service.

This paper focuses on the adaptation service. As retraining an expert model takes significant compute resources and time (§1), the adaptation service becomes a key bottleneck in resource efficiency and accuracy consistency. We observe that these systems [20, 21] need to spend 70%–90% of the overall compute resource on retraining their expert models. This is because model training is much more expensive than model inference. Besides, knowledge distillation needs to run the large teacher model to generate data labels on the sampled frames. In order to address this fundamental challenge, we need an effective approach to minimize the necessity of invoking expert retraining.

2.2 The Case of Reusing Historical Expert Models

It is well known that a video deployment usually exhibits temporally and spatially recurrent patterns [22,23,32]. Similar video scenes reoccur on the same camera at a similar time of day (e.g., morning or night), weather (e.g., sunny or raining), and location (e.g., a drone revisits the same street). More importantly, a video scene from one camera can also appear on *other* cameras, especially those in the same geographical vicinity, such as a self-driving car visiting a place that other cars in the same fleet have seen. These temporal and spatial correlations imply that some expert models trained on video scenes in the past could perform reasonably well on the current video scene, and we can potentially leverage these historical expert models to minimize the necessity of retraining.

To empirically show the potential of reusing historical expert models, we use a total of 71 hours of driving videos collected from YouTube (more details in §5.1). The large teacher model is a state-of-the-art object detector DNN, YOLOX-X (282 GFLOPs), and the expert model is a much smaller variant YOLOX-Nano (1 GFLOPs) [13]. Similar to existing continuous retraining solutions, we train one expert model for each 30-second video segment. We create a model zoo using all the expert models trained on the first 30 hours of the videos ("training data"), and we use the remaining 41 hours of the videos ("test data") to report the object detection accuracy.

We evaluate four designs:

1. **No Adaptation:** trains a single expert model based on all training data and deploys this expert on the test data.
2. **Continuous Retraining:** periodically retrains an expert model for each camera using the most recent video segments. This serves as a reference point of recent model-retraining systems, such as AMS [20] and Ekya [21].
3. **Ideal Model Reuse:** deploys the *best* expert model from a given model zoo created based on video segments in the first 30 hours (ignoring the model-selection overhead). This can be seen as a strictly better version of ODIN [23], recent model reusing baseline.
4. **Ideal Reuse with Retraining:** combines 2 & 3 (retraining the reused model selected by 3.) This shows how much an ideal model reusing scheme can improve in a continuous retraining framework.

All designs are given the same amount of GPU resources to continuously retrain expert models, while No Adaptation (Design 1) and Ideal Model Reuse (Design 3) do not use this resource for retraining.

Benefits in resource efficiency: Fig. 2 shows the mean Average Precision (mAP) score on the test data while varying the number of cameras. The observations are two-fold.

First, model reuse is a promising direction in minimizing retraining. The benefits of model reuse become more evident when the compute resource is not enough to retrain the expert models for more cameras (4, 6, and 8 cameras). Even when the compute resource *is* enough for model retraining (2 cameras), Ideal Model Reuse can still achieve a similar mAP as Continuous Retraining. This observation is encouraging because reusing history models does not require the resources (not shown here) to retrain any new expert models, and at the same time, the best expert model in the past already achieves comparable accuracy with the expert models trained on the most recent video data.

Second, model reuse has a promising synergy with continuous retraining—Ideal Reuse with Retraining achieves the highest mAP across the board. This is because the reused model provides a strong starting point for retraining, which reduces the compute resource needed by retraining (i.e., faster convergence) *and* improves the inference accuracy of the resultant expert models.

Benefits in accuracy consistency: Another key benefit of model reuse is that we do not need to wait for an expert model to finish retraining. This is particularly important when a camera has experienced a sudden scene change and is in urgent need of a new model. For example, when a car drives into a tunnel, we can select and change the expert model quickly without the latency of training a new expert (Fig. 3 shows a concrete example). We demonstrate this benefit with the CDF of mAP across all video segments for the 8 camera setting (Fig. 4). As the figure shows, Ideal Model Reuse has a much better

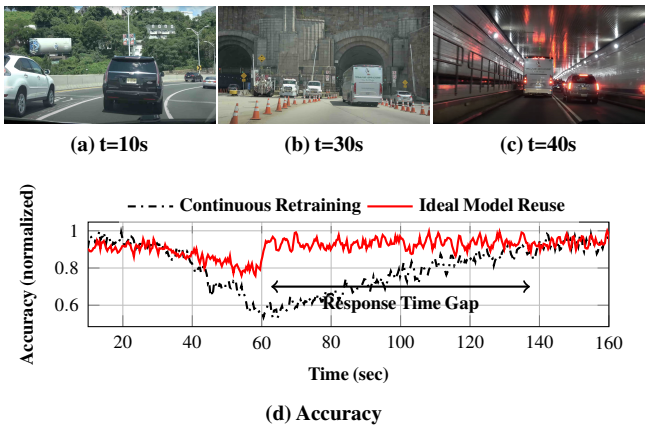


Figure 3: Example of a scene change when a car camera enters a tunnel and how fast Ideal Model Reuse and Continuous Retraining respond: Model updates arrive every 30 seconds. At t=60 sec, both schemes can access sample frames from the tunnel scene. Ideal Model Reuse switches to a good model for the new scene immediately at $t = 60$ sec, whereas Continuous Retraining takes about 80 sec to retrain the model till the accuracy bounces back.

tail mAP than Continuous Retraining. For instance, at the 1st percentile, Ideal Model Reuse retains 24% mAP while Continuous Retraining drops to an unacceptable 7% mAP. Fig. 3 illustrates a concrete example. As the car drives into the tunnel ($t = 40s$), Ideal Model Reuse switches to a matching expert much faster ($t = 60s$) than Continuous Retraining ($t = 120s$), which leads to a much more moderate drop in model accuracy.

Challenges of model reuse: Several technical challenges need to be addressed to fully realize the benefits of model reuse. Ideal Model Reuse assumes that it always selects the best expert model with no compute cost or delay in searching through all experts in the model zoo, which is not practical. Recent model reuse solutions in the database community (e.g., ODIN [23]) cannot address these challenges either, because they are not designed for resource efficiency, when sharing the compute resource among the functions of model selection and model retraining for many edge devices. To unleash the potential of model reuse in practice, we need a mechanism to find the best expert model quickly and accurately. We also need to rein in the cost and latency of model selection, so that it does not grow indefinitely with the number of videos or cameras.

In summary, reusing historical expert models is a promising complement to model retraining, and when used jointly, it leads to better resource efficiency and more stable and accurate model adaptation. That said, to make model reuse practical, several technical challenges remain, which we will tackle in the next section.

3 Design of RECL

This paper presents RECL, a new end-to-end design of model adaptation for continuous learning on edge devices. At a high level, RECL is given an accurate-yet-expensive model (the "teacher") and a set of edge devices, and it automatically adapts

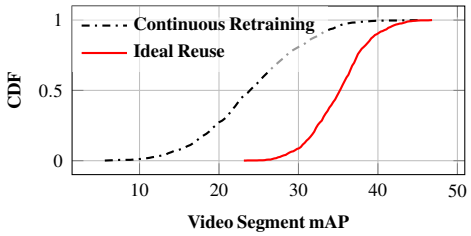


Figure 4: Ideal Reuse improves both average and tail accuracy (mAP) across video segments.

the deployed lightweight ("expert") models, each dynamically tailored to an edge device's particular distribution of video frames at any point in time, allowing each edge device to obtain results similar to running the teacher model.

Overall architecture (Fig. 5): RECL launches a *model-adaptation controller* on a server machine (e.g., in the cloud, edge compute cluster, etc.), which manages a set of daemons running on edge devices. The controller selects and deploys lightweight models on edge devices, which run local fast inference using the lightweight model. This work focuses on the adaptation controller, and the optimizations inside the edge devices or on the communication between the controller and the edge device are orthogonal to RECL. Furthermore, we assume the interactions between the server and edge do not interfere with any other processes running on the edge device (including the local inference).

In each model-update window (by default, every 30 seconds),[1] each edge device sends sampled frames to the controller to query if a new model should be used. (Note that the RECL controller only updates models for edge devices, which then use models to run inference on video streams.) The frame sampling rate is set dynamically based on the extent of scene change (similar to the technique used in AMS [20]). AMS takes the drift rate of the labels measured at the server as a signal for setting the frame sampling rate. As labels are usually in a lower dimension than input images, their variation rate is a less noisy proxy for detecting the scene change pace.

Based on the sampled frames, the controller performs two basic functions—*model selection* (§3.1), which selects a suitable expert model from a collection of history expert models to quickly respond to the edge device's query, and *model retraining* (§3.2), which fine-tunes the selected model based on the sampled frames and manages GPU resources to many edge devices to retrain their models. Furthermore, retrained models are periodically added to the model zoo shared with other edge devices (§3.3). The rest of the section will present their designs and rationales.

[1] We use fixed update windows, similar to Ekya [21]. Dynamic window size is orthogonal to RECL. In general, an update window can be triggered by an edge device when it detects substantial changes in its video stream, and there are several prior efforts on scene change detection.

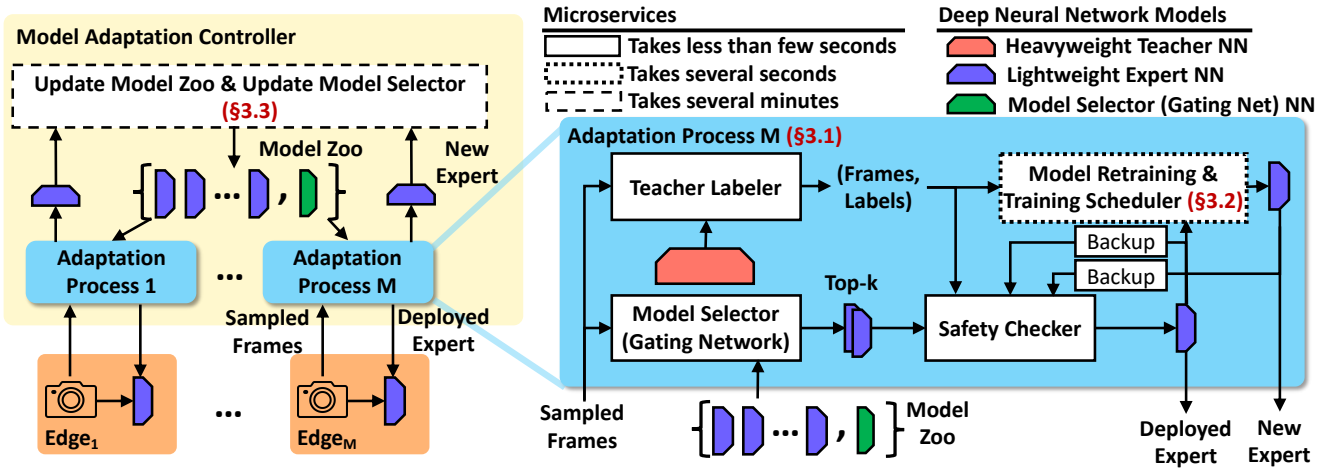


Figure 5: RECL system architecture. Edge devices (cameras) run real-time inference using lightweight models, and the model-adaptation controller manages a model zoo of expert models trained on history frames from edge devices (cameras) and, on receiving a model-update query, quickly selects a suitable model from the model zoo and recently trained models (the light-blue box). New models are also continuously retrained (optimized by a custom training scheduler) and then incrementally added to the model zoo over time. (The figure does not show optimizations to speedup inference on edge devices or the controller-device communication, as they are orthogonal to RECL.)

3.1 Model Selection

RECL's model selection module, on receiving a query from an edge device, should *quickly* select a high-quality (accuracy) expert model from a collection of models. RECL achieves this goal by: (*i*) maintaining a large (potentially growing) *model zoo* of history expert models that are previously trained for any edge device; and (*ii*) using a fast and robust selection procedure to navigate the large model zoo.

Sharing model zoo across video sessions: RECL's model zoo consists of a set of lightweight expert models, each trained for a specific scene distribution previously seen by some edge device managed by the controller. For example, if the controller manages several driving video sensors in an area, the model zoo might contain experts for different streets/neighborhoods, different weather conditions, etc. It is crucial to note that RECL does *not* directly rely on any priors about the features (e.g., weather conditions) of video content as a signal for creating new models; rather, an expert model is created based on frames of an edge device in an update time window, and then added to the model zoo if it improves performance (see §3.3).

An important design choice of RECL is that rather than caching the history models of different devices separately, RECL *shares* the model zoo and its gating network across devices, enabling model *reuse* across similar video sessions of different devices that might share similar temporal-spatial correlations (e.g., in the same geographical vicinity) [33]. This reduces the need for online model retraining and improves system responsiveness when an edge device experiences a sudden scene change for which a previously trained model (probably of another device) with good accuracy is available. For example, cars in the same city would observe the same scenery over time, even though the frames observed throughout one driving session may vary significantly. In such an application, the model zoo would eventually include an expert for most scene distributions encountered, significantly reducing the need for per-session model training.

Fast, robust online model selection: Figure 5 (right-hand side) describes RECL's online procedure to select a model from the model zoo. One strawman solution to the model selection problem is running an exhaustive search over all experts in the zoo. However, the number of models in the zoo can grow large over time, and it would become prohibitively expensive to select models by testing all of them on the sampled frames in each update window. To scale model selection to a large model zoo, RECL uses a *gating network* [26] to directly infer which models in the zoo better fit a given video content. The gating network is a lightweight DNN that given an image, assigns a score to each model in the model zoo. Logically, the gating network is similar to an image classifier, except that the labels are not object classes but models in the model zoo. A higher score indicates the model likely has higher accuracy on the image. (§3.3 will explain how to update the gating network to handle the changing model zoo.)

An alternative approach [23,34] to model selection is to map video content to an embedding space (via an autoencoder), partition the embedding space, and map each partition to a specific expert model. We found that this approach works poorly in practice (§5.2). The intuitive reason is that auto-encoders are trained to learn the distributions of only *input* data (e.g., which video frames look similar), rather than simply learning which frames can share a good expert model. The former task is too generic, and therefore, it is significantly hard to learn an efficient embedder to deploy in practice. We refer readers to [35] for further details. In contrast, RECL's gating network directly predicts the quality (accuracy) of each

expert model and avoids the need to have a good auto-encoder.

That said, it is hard to train a gating network that always picks the best model from the model zoo. Instead, RECL runs the gating network on the edge device's latest sampled frames and selects the top-K models (e.g., $K = 10$) with the highest average scores. The intuition is that the performance of the best of the top-K models improves quickly with larger K (see §5.3). In short, the top-K filtering approach strikes a decent balance between leveraging a large model zoo and fast model selection.

The *safety checker* then tests the accuracy of these top K models, along with the current model of the edge device and the last model retrained on video frames from the same edge device (explained in §3.2). The testing is based on the sampled frames and their "ground truth" labeled by the more accurate and more expensive "teacher" model. Finally, among these models (top-K from the model zoo, current model, and the last retrained model), RECL selects the one with the highest empirical accuracy on the labeled images and sends it to the device. This online model selection process is fully automatic and has a low compute cost. For the object detection task, for example, we use YOLOX-nano for the lightweight experts and ResNet18 as the gating network. These two models have a close inference cost per sample (1.1 vs. 1.8 GFLOPs). However, the gating network only runs on a significantly smaller subset of frames (e.g., 1/30th of frames).

3.2 Model Retraining

So far, we discussed how to reuse the previously trained models. Like other continuous learning frameworks [20, 21, 23], RECL also retrains models online for each edge device. The edge device periodically queries the controller in every model-update window. For each query, RECL will initiate a retraining job using the sampled frames sent by the device (similar to [21]), after the model selection process described above is finished.

However, to scale to more edge devices, many of which need new models, RECL must carefully allocate its GPU resource to model retraining jobs. The basic idea of RECL is to closely monitor how accuracy improves on each training job and dynamically share more GPU resources to the jobs that benefit more from additional GPU cycles.

RECL time-shares the GPU among multiple retraining jobs by micro-windows—in a micro-window, we let one of the retraining jobs use all GPU cycles and may switch to a different job at the boundary of micro-windows based on the logic described next. Each micro-window is long enough for one retraining job to complete one epoch (i.e., going through all sampled frames once). A typical micro-window size is about one second. (We will explain the reason for timesharing GPU shortly.)

Retraining scheduling algorithm: Targeting a fixed maximum accuracy gap with the teacher model for each video scene can become quickly intractable as it can be pretty challenging for the student model to track the same target performance for all real-world scenes. However, as our results show later, we can still target a fixed maximum gap on the average accuracy. Hence, having a system that uses the resources efficiently, we can always add more resources as the number of cameras grows till we are happy with the overall accuracy.

Algorithm 1 RECL GPU Sharing Algorithm

```
1:  Input: training requests R, micro-window number of seconds μ, window size of T sec
2:  budget ← T                                   ▷ Total time budget
3:  procedure ProcessRequest(r)
4:      acc_i ← r.Eval()
5:      Train the model for request r for μ seconds
6:      acc_f ← r.Eval()
7:      budget ← budget − μ
8:      return (acc_f - acc_i)/μ                 ▷ Returns the accuracy gain
9:  end procedure
10: for r in R do                                ▷ Initialize the gain estimates
11:     gain[r] ← ProcessRequest(r)
12: end for
13: while budget > μ do                          ▷ Schedule the most promising
14:     r ← argmax gain                          ▷ Find the request with max gain
15:     gain[r] ← ProcessRequest(r)
16: end while
```

Consider C concurrent training jobs (one for each edge device). We define $I_c(\tau)$ as the improvement achieved from training the model corresponding to camera c for τ seconds. Our objective is:

$$\max_{\tau_1,\tau_2,\ldots,\tau_C} \sum_{c=1}^{C} I_c(\tau_c) \quad s.t. \sum_{c=1}^{C} \tau_c = T \tag{1}$$

That is, given a time budget T (e.g., the update window duration), we want to time-share GPU resources to maximize the total improvement of accuracy across all models.

To solve this optimization problem, RECL uses the following *iterative* scheduler (Algorithm 1). At the beginning of each update window of size T, the scheduler receives a set of training requests, $\mathcal{R}$. Each training request corresponds to a set of labeled frames (already labeled by the teacher model as part of safety checking), and an expert model checkpoint (selected by the safety checker at the beginning of the window). In each micro-window of μ seconds, the ProcessRequest procedure (Lines 3-9) takes one of the requests r as input and evaluates *how much its accuracy improves between before and after a micro-window*. Notes that the cost of these accuracy evaluations is ignored as they only require a lightweight forward pass on the test subset of the data.

The main loop of the algorithm first spends one micro-window to process each request and initialize its accuracy improvement (Lines 10-12). Then it iteratively picks the

training request with the largest accuracy improvement as our next model to train till we run out of time (Lines 13-16).

Since DNN training curves $I_c(\tau)$ are usually concave (i.e., accuracy improves quickly and then slows down), then this iterative algorithm effectively minimizes the maximum speed of these models' training curves ($\frac{\partial I_i}{\partial \tau_i}$). It can be shown that this iterative process converges to a near-optimal partition of the total time budget that maximizes the total accuracy improvement across the training jobs [36].

Design choices: We highlight two design choices behind the retraining scheduler.

To find the best GPU allocation, both Ekya and RECL predict each retraining job's training speed (accuracy improvement vs. epochs) but with different approaches. Ekya periodically runs extra ("out-of-band") micro-profiling on each camera: running a few epochs of training on a subset of history images to build a profile of the training curve of each camera. Such upfront micro-profiling has extra compute overhead and fails when the training curve changes over time. In other words, they inherently trade off between the profile accuracy and their overhead. In contrast, RECL uses an "in-band" profiler—it measures the *actual* learning progress (accuracy improvement) of each job on the fly and dynamically determines which one progresses faster. This scheme avoids the micro-profiling overhead of Ekya without losing accuracy. Note that RECL requires fast switching between models, which will be discussed next. Our scheduler's iterative algorithm is similar to [37] which is designed to achieve fairness among the cluster-level training jobs which compete at a significantly longer time scale.

Instead of splitting GPU cycles spatially across concurrent training jobs, RECL time-shares the GPU cycles by switching among concurrent retraining jobs every micro-window. While it is logically equivalent to spatially sharing, RECL's GPU timesharing is based on three practical considerations. (1) The delay to context switch between GPU-loaded models (usually less than tens of milliseconds) is negligible compared to a micro-window. Since a lightweight model in RECL has a small memory footprint, we can load it to GPU memory and not swap it out (even when switching between retraining jobs) until the model retraining completes. (2) Unlike Ekya, RECL does not have to finish model training very quickly (it responds to each edge device by first selecting a good model from the model zoo or the most recently trained model has been good enough). (3) It does not rely on any GPU library to dynamically reallocate GPU across different jobs.

3.3 Updating the Model Zoo & Selector

Admission of new models to model zoo: RECL does not add every retrained model to the model zoo. A recently trained model is considered *promising* if the safety checker finds this retrained model's accuracy is α higher than the rest of the candidate models (the top-K experts selected by gating network and edge device's current model). These promising models are put in a queue. When the promising model's queue grows larger than a fixed threshold, γ, we empty the queue by adding them to the model zoo and update the gating network (explained next) to consider the recently added models. Hence, α and γ control the frequency of model selector updates. We later study the impact of the zoo admission rate on the system performance (§5.3).

Incrementally update of gating network: Recall that the gating network predicts the accuracy of each expert in the model zoo on the input frame. Hence, when updating the gating network to handle new expert models, we need to first label the accuracy of all experts on both the new frames that were used to train the new expert models as well as a sub-sampled set of history frames (those used to update gating network before). To this end, we label the new samples with existing experts in the zoo and label the existing samples with the new experts added to the zoo. This way, we track the performance of all experts on a sampled set of frames so far. Note that when we create the training frame set of the gating network, we sub-sample frames used before and mix them with the new frames in order to keep the same training size over time.

As the zoo size increases, the output size of our gating network must change as well. Since the accuracy prediction logic does not change for most of the models in the zoo, we only need to add corresponding neurons for the new models to the final layer without changing the connectivity weights for existing expert models. This way, we transfer as much knowledge as possible from one gating network to the next. To further speed up the training of the gating network, we use the mean and variance of the most recent model selector in order to initialize the connectivity weights corresponding to the new experts in the final layer.

Pruning the model zoo: Though updating the gating network is usually fast, the overhead of retraining for updating the gating network grows proportionally with the size of the model zoo. To prevent the model zoo to grow indefinitely, we deem an expert in the zoo ineffective if other experts always have a preferred accuracy. In particular, we remove the experts that are chosen less than η times in the last q model selection calls. We set $q = 3000$, which is about one day's worth of video streaming in our system and study the impact of the η parameter later in §5.2.

4 Implementation

We have implemented RECL in Python and used Pytorch [38] for inference and training of ML tasks. For communication between the services, we use the gRPC [39] framework for remote procedure calls.

Microservices: We implement several microservices to prototype RECL. These microservices are designed to generalize to different continuous adaptation design choices in prior work. RECL runs a camera streaming service on each camera device to send the subsampled video frames to the teacher labeler service running on the adaptation server. We use TensorRT [40] and half-precision computation to further

speed up the inference processes. One runner microservice manages the coordination of different components across all video streams that share resources in the server.

Hooks: Each microservice can register a hook in other microservices. These hooks are specific functions to run at predefined time events in the system. Our time events are a combination of before/after, window/microwindow, and first/last time. For example, if a model zoo update strategy requires to have information about the training gain of each model at the microwindow level, it registers accuracy evaluation hooks in the training scheduler before and after each window.

Adaptation state: There is an adaptation state shared across all microservices that register to the same runner. All microservices have read and write access to the adaptation state to optimize their decisions, possibly share the hooks results, and keep track of possible global events like the beginning of a new window.

Training strategy: Our training scheduler service relies on a sharing strategy abstraction. Each strategy has access to the adaptation state, can register or subscribe to a hook, and has a run method to decide which camera model should train next in each microwindow. If an adaptation scheme is not microwindow-based, it only has to register hooks for the first and last microwindow.

Performance monitoring: For tracking the system performance metrics, we implement logging hooks to track system-level metrics like compute times and resource utilization in addition to RECL-specific performance metrics like zoo admission rate and model reuse rates.

5 Evaluations

Finally, we evaluate RECL on two video-analytics tasks using real-world driving videos. Our key findings include:

- Given the same compute resource, RECL improves the object detection mAP and image classification accuracy over state-of-the-art baselines by up to 9.0% and 7.4%, respectively. The baselines need to use at least 3.2× more compute resources to match RECL's accuracy.
- The superior performance of RECL comes primarily from our distinctive design of model reuse. RECL 's fast gating network and safety checker outperform the state-of-the-art model selection mechanism in terms of accuracy and efficiency by a large margin.
- RECL is highly responsive to a model-update request. On average, the time RECL needs to adapt models to the same accuracy is 11–91 seconds faster than that of the baselines, with the gap growing both at the tail of the distribution (by almost 2×) and with the total number of cameras.
- RECL's retaining scheduler also makes better use of GPU. In contrast to round-robin and out-of-band profiling used in several recent continuous learning systems, RECL's in-band profiling provides a 2.0% higher mAP at up to 6.1× lower overhead.
- Compute overhead of RECL decreases gracefully over time as more models are trained and added to the zoo.

Model	Params	FLOPs	Throughput (FPS)
MobileNetV2	3.5M	0.32G	1.5K
ResNet50	25.6M	4.12G	153
YOLOX-Nano	0.91M	1.1G	312
YOLO-X	99.1M	282G	58
ShuffleNetV2	2.28M	0.15G	3.7K
ResNet18	11.7M	1.8G	490

Table 1: Specifications of the models used for the evaluation. Throughputs are reported for NVIDIA V100 GPU with a batch size of 1.

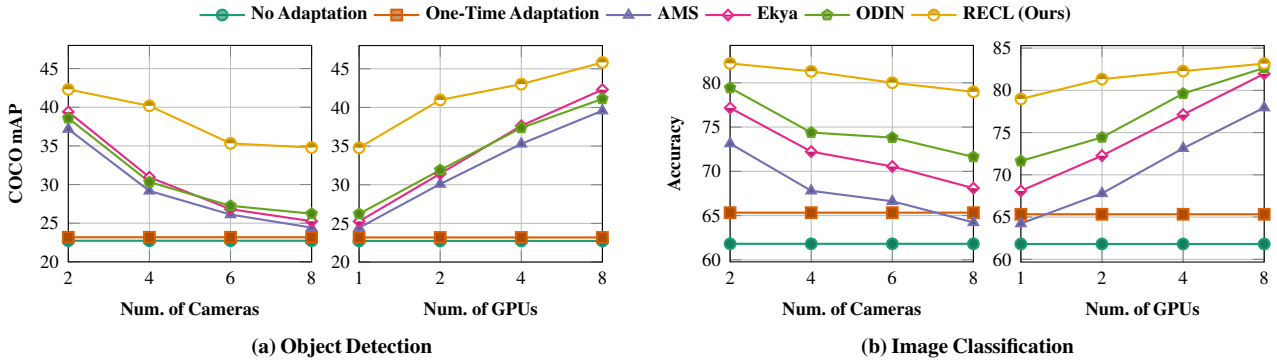
5.1 Methodology & Setup

Dataset: We evaluate RECL on two computer-vision tasks—image classification and object detection—using 151 driving videos collected from YouTube. Since we would like our video sessions to include meaningful data drifts, we adopt videos that have a length of at least a few minutes (up to a couple of hours)[2] with a total length of 71 hours. Furthermore, our dataset covers a wide range of cities and driving situations in North America, including weather conditions, time of day, and driving speed. Note that in each experiment, we do *not* play the exact same video segment twice on *any* edge devices, since it might artificially amplify the gain from model reusing. Driving video is a remarkably challenging workload for evaluating our system as the scenes change more widely and frequently. This workload brings a variety of situations where exact matching is impossible and requires more than a few models to cover the wide range of possible scenarios. Responsiveness is also more challenging for driving cameras compared to fixed cameras. For example, traffic light cameras mostly need only to update every few hours when the lighting/weather change, significantly stressing the compute power at the adaptation server.

Models: For object detection, we use YOLOX-Nano and YOLOX-X [13] for the student and teacher models, respectively. For image classification, we use MobileNetV2 [42] and ResNet50 [43] for the student and teacher models. Details of these models are shown in Table 1. Our models are pre-trained on ImageNet [44] and COCO [45] datasets for classification and detection, respectively. For fast model selection, we use ResNet18 as the gating network architecture by default, unless otherwise stated.

Metrics: To evaluate the accuracy of different schemes, we compare the inference results on the edge device with labels extracted for the same video frames using the teacher model (similar to prior work [20, 21]). We use *mean Average Precision (mAP)* for the detection task, while for classification, we report *accuracy* by the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined. We calculate these metrics across *all* 80 and 1000 classes of MS COCO and ImageNet datasets for

[2] Video sessions in other similar video datasets like Berkeley Driving Dataset (BDD) [41] were not long enough for our purpose. For example, each driving episode in BDD is only 40 seconds.



(a) Object Detection (b) Image Classification

Figure 6: End-to-end scaling of the average accuracy across different schemes for two typical vision tasks.

detection and classification, respectively.

Setup: In our setting, model selection and training of the adaptation controller happen in the cloud, and each edge device only runs inference by the lightweight expert model on the local video stream. All experts can run at a real-time inference speed (30 frames-per-second) even on lower compute power edge devices such as NVIDIA Jetson Nano [46] and Coral Edge TPU [47], and we do not evaluate any optimization on the edge device, as it is orthogonal to RECL. We use NVIDIA V100 GPUs for the adaptation server. The adaptation processes of different edge devices share the same pool of GPU resources.

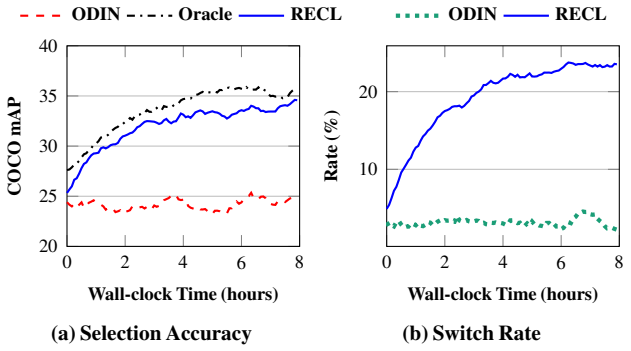
Baselines. We compare RECL against the following continuous learning methods:

- **No Adaptation:** We run the pre-trained model on the edge device without any adaptation.
- **One-Time Adaptation:** We fine-tune the entire model on the first half of the videos and test on the rest. This adaptation happens only once. Comparing RECL with this scheme will show the benefit of having a continuous adaptation system in place.
- **AMS:** We implement Adaptive Model Streaming (AMS) as in [20], which uses a remote server to continually adapt lightweight expert models running on edge devices. As the update intervals are longer and our lightweight models are smaller than AMS, network bandwidth consumption is less of a concern in our setup. As such, we relax the bandwidth constraint of AMS and allow for full model parameter updates in this scheme. AMS uses a simple round-robin mechanism for GPU sharing. Comparison with AMS mainly highlights the gains of model reuse and optimized GPU sharing. As AMS reasonably outperforms Just-In-Time [19] and remote server inference in prior work [20], we no longer compare with these schemes.
- **Ekya:** Ekya enables both retraining and inference to co-exist on the edge node without any model reuse. Since RECL shares the server GPU resource only among model retraining and selection jobs (inference is on edge devices), for a fair comparison, we compare RECL with applying Ekya's microprofiler and thief scheduler (released in Ekya [21]) to model retraining jobs. Despite the more sophisticated resource-sharing mechanisms compared to AMS, Ekya, however, incurs the out-of-band profiling overhead and cannot reuse models compared to RECL. Moreover, since Ekya shows how continuous retraining significantly outperforms naive model reuse methods (e.g., reuse models from the same time of the day) [21, §6.4], we do not compare RECL with these naive reuse heuristics.
- **ODIN:** ODIN [23] is a video analytics system that can detect and recover from data drift by building expert models based on the similarity of video scenes. We use the autoencoder-based method proposed in ODIN for model selection. Specifically, the average of embedding vectors of the sampled frames in a window is used as the embedding vector of that window. Also, each trained expert is assigned an embedding vector the same as its training data. We use the L2 distance between the embedding vector of a window and the models in the zoo as a measure of similarity, and the model selector returns the model with the least distance from the samples in each window as in the ODIN paper [23].

5.2 Results

End-to-end performance: We first compare the end-to-end accuracy of RECL with the baselines over a range of provisioned GPUs and a varying number of concurrent cameras replaying videos from our dataset. Whenever a video ends, we continue with another video from our dataset. Note that we never repeat the same video twice as it favorably impacts the accuracy gain of model reuse. We use NVIDIA V100 GPU as the adaptation server. In measuring the impact of the number of cameras on the accuracy, we fix the number of GPUs to 1. For the varying number of GPUs experiment, we run a workload consisting of 8 cameras. As shown in Fig. 6:

1. Continuous adaptation significantly improves mAP and accuracy. Gains from continuous learning grow with more resources provisioned per camera.
2. Overall, RECL outperforms all baselines by a large margin. In object detection, for instance, RECL improves mAP by up to 9.0% (8 cameras, 1 GPU) compared to the second best approach. In terms of resource consumption, RECL supports 2.6× more cameras on one GPU, and requires



(a) Selection Accuracy (b) Switch Rate

Figure 7: An example of RECL model selection performance over time. As the model zoo grows, (a) accuracy of the RECL-selected models gradually improves, and (b) the model selected by RECL's gating network has higher accuracy than models selected by ODIN (as evidenced by the fact that the safety checker would more frequently pick the model by the gating network than it would pick the model selected by ODIN).

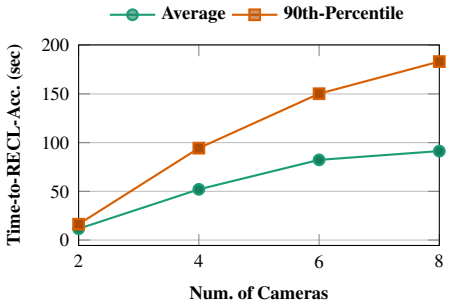
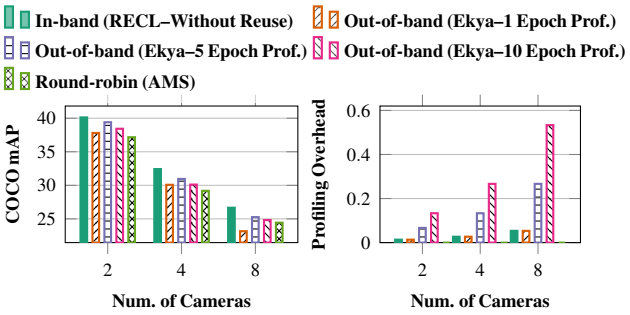


Figure 8: Model reuse impact on improving the response time.

3.2× fewer GPU cycles to maintain an mAP of 35%.

3. mAP/accuracy improvements from model reuse are significant. Compared with Ekya and AMS which do not reuse historical models, RECL brings up to 9.8% and 10.7% improvement in mAP and accuracy for object detection and image classification, respectively.
4. Without model reuse, RECL's scheduler provides about 1.5% mAP improvement compared to Ekya.
5. In image classification, ODIN performs the best among the baselines due to its model reuse and specialization design. Nonetheless, ODIN's auto-encoder-based model selector (and the lack of optimized resource sharing) performs poorly on relatively complicated tasks like object detection. In contrast, we see better performance of RECL across all settings in both tasks due to our unique model selector and retraining scheduler design.

Model selection performance: To directly examine the model selector performance, in Fig. 7a, we plot the accuracy of the selected models vs. the system's wall-clock for 8 GPUs, i.e., within each hour on the wall-clock, the system ingests 8 hours of video. In the figure, we also include the performance of ODIN (a recent model selector) over the same zoo created by RECL, as well as the accuracy of an oracle model that exhaustively searches over all models in the zoo at each point in time (while ignoring the oracle's compute overhead). It



(a) Scheduling Perf. (b) Profiling Overhead

Figure 9: Impact of profiling on retraining performance: RECL's retraining scheduler (which uses a low-overhead in-band profiling) outperforms Ekya (which relies on out-of-band profiling on each job) and AMS (whic uses a round-robin scheduler).

is not surprising that the accuracy of the model selected by RECL improves over time as more models are being added to the zoo. Furthermore, we observe that RECL performs closely to the oracle selector, while ODIN struggles to select a good model from the same zoo. Notice that the cost of running the oracle model is prohibitively expensive as, after a couple of hours, it requires testing the accuracy of thousands of experts in the zoo for each sample frame. On the contrary, RECL uses ResNet18 as the underlying gating architecture that runs at 490 frames per second (see Table 1).

We further notice that, in Fig. 7a, the model zoo roughly converges to a desirable accuracy after four hours, totaling 32 hours of video stream ingestion. This observation shows an opportunity to reduce model zoo update frequency (and thus its cost) after enough representative experts are collected in the system. With the growing model zoo, model reuse becomes more favorable over time as well. Fig. 7b depicts the percentage of the time that the safety checker prefers the selected model over the rest (e.g., a recently trained model). For a fair comparison of the effectiveness of model reuse, we run RECL and ODIN end-to-end independently (i.e., they are not sharing the same model zoo). As can be seen in Fig. 7b, RECL's model hit ratio increases with a larger zoo, making our system both more accurate and efficient than ODIN. Moreover, notice that the safety checker picks the offered historical model 25% of the time in the case that the most recent trained model is also coming from RECL. To better understand the model reuse impact here, we design the following experiment.

Impact of model reuse on responsiveness: Model reuse improves the response time of the adaptation server by not needing to retrain a new expert model first. To directly evaluate this effect, we first profile the accuracy of 102 models generated in Ekya (in a video of 51 minutes long) against 2 minutes of offline training on a single V100 GPU. Using these profiles, we then measure the time it would take Ekya, as a continuous retraining approach, to adapt each model to the same accuracy level of the RECL's selected model for reuse on the same window. We refer to this metric as Time-to-RECL-Accuracy. Figure 8 shows the

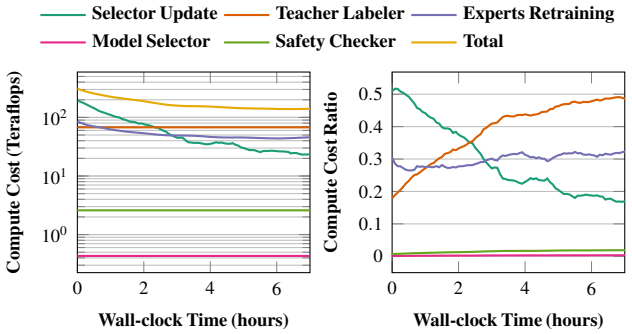


Figure 10: Breakdown of compute cost by the components of RECL controller. The total cost drops over time as the extra-cost of maintaining the model zoo significantly reduces, allowing RECL to enjoy the benefit of model reuse without much additional overhead.

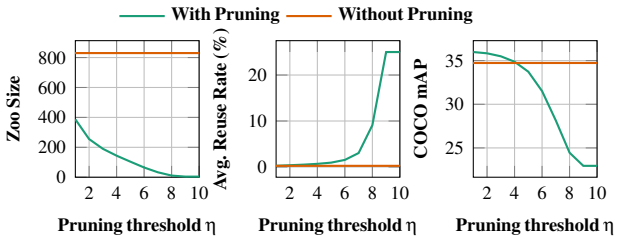


Figure 11: Pruning policy impact on RECL accuracy.

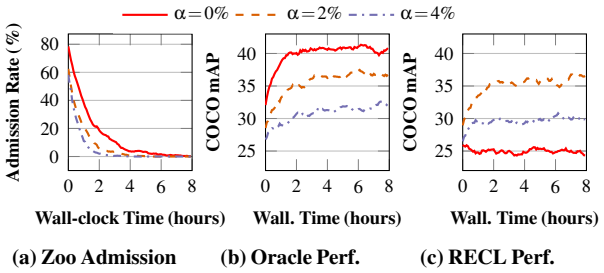


Figure 12: Impact of changing the admission rate through the α-promise threshold on RECL model selection performance.

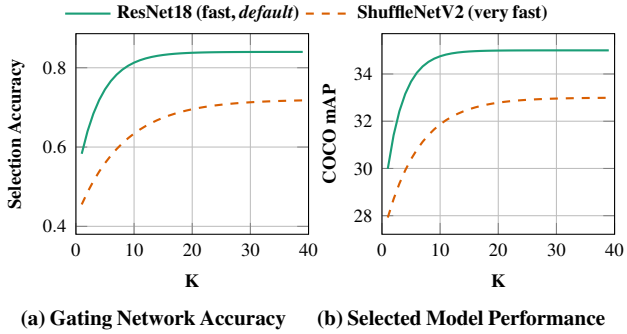


Figure 13: Impact of using top-k models suggested by the gating network for the default gating network and a faster gating network model.

mean and 90th percentile of the Time-to-RECL-Accuracy for the object detection task across a varying number of cameras sharing one GPU. We observe that Ekya takes up to 90 seconds longer than RECL, on average, to achieve the same level of accuracy. More importantly, this gap grows significantly large with increasing the number of cameras and at the tail scenarios.

Scheduler performance: We now evaluate RECL's retraining scheduler with its "in-band" profiling (§3.2) and compare its performance with Ekya's out-of-band micro-profiler and AMS's round-robin scheduling method. For a fair comparison between Ekya and RECL, we turn off RECL's model reuse and let Ekya adapt its early stop parameter, which has a similar effect to the micro-window-based scheduling in RECL. To run Ekya's profiler, we set its early stop parameter to 1, 5, and 10 epochs. Fig. 9 compares the accuracy and profiling overhead (ratio of the time spent on profiling in each window) of these schedulers vs. the number of cameras. We observe that Ekya's out-of-band profilings are either too costly to run (e.g., Ekya with an early stop of 10 epochs), or too noisy to identify a good early stop parameter, which results in low accuracy. For example, an early stop at 1 epoch has the same cost as RECL's in-band method but performs worse than round-robin when resource allocation becomes more challenging with 8 cameras.

Breakdown of compute cost: Fig. 10 shows the cost of different components in RECL over the course of 7 hours. Initially, model selector update has the dominant cost in the system. However, as the zoo grows over time, the need for updating the model zoo (and consequently the selector) reduces to the extent that after a while, the teacher labeler and training scheduler become the dominant cost of the system, but these "base cost" is the same as a typical continuous retraining system (such as Ekya and AMS). In short, the extra overhead for RECL to enable model reusing (model selector and maintaining a growing model zoo) significantly reduces over time.

5.3 Ablation Studies

Model zoo pruning: In Fig. 11, we compare RECL accuracy across various levels of pruning intensity over the course of nearly 60 hours. Naturally, reducing the value of η leads to a significant drop in model zoo size without much accuracy sacrifice. For instance, a balanced choice of the pruning threshold, $\eta = 4$ provides the same accuracy despite efficiently shrinking the size of the model zoo by a factor of 5.6×, from 830 down to about 150 experts.

Zoo admission rate impact (α-promise margin): In order to evaluate the impact of the admission rate, we turn off the zoo pruning mechanism and measure the selected model accuracy. Fig. 12 shows this accuracy for three levels of α for both the ideal oracle selector and RECL's selector. As decreasing α allows for admitting more models to the zoo, the oracle-based scheme can choose among more models. However, it gets harder for the gating network model to select from an arbitrarily large model zoo. Hence, we observe a diminishing return in increasing the admission rate beyond $\alpha = 2\%$, which seems to be a good balance between the zoo size and the model selection complexity.

It should be noticed that the exact values of these parameters (η, α) largely depend on the dynamics of video content. The

message from Fig. 12 and 11 is that there are sweet spots for them that, on a large set of videos, strike a desirable tradeoff between the cost of maintaining a reasonably sized model zoo and the quality (accuracy) of the selected models. The parameter γ controls the gating network update frequency (i.e., cost). As the Selector Update cost in Fig. 10 shows, this cost diminishes over time as the system collects a comprehensive set of experts. Therefore, RECL 's performance in steady state is not as sensitive to γ as it is to α and η.

Model selector top-k: As discussed in §3.1, we pass the top-k selected model to the safety checker (instead of 1) in order to find a better model for reuse. In Fig. 13, we show the accuracy of model selection and the performance of the selected model for our default and a faster gating network model (see Table 1 for speed comparison). Given this observation, we find $K = 10$ is a good default operating point for RECL. Notice that while a higher K increases the cost of the safety checker, as shown in Fig. 10, our safety checker still has a negligible overhead compared to the other components in the system.

6 Discussion

Safety-critical applications: Predicting the feasibility of minimum accuracy thresholds is not a trivial problem in non-convex ML training tasks. Therefore, as we cannot guarantee a minimum accuracy level using continuous adaptation for safety critical problems, the solution might come at the cost of provisioning enough resources to run the large state-of-the-art model for inference. However, if the problem is not safety-critical, one solution might be to set minimum accuracy thresholds with timeouts to achieve them, which we leave to future work.

Data residency: RECL requires sharing training samples with the adaptation server. While there are recent solutions in computation over encrypted data for secure AI [48], our current evaluation has been based on having access to the actual video frames. Depending on the data residency policies, such data sharing may constrain how far the adaptation server can be taken from the cameras.

7 Related work

Optimization of video-analytics systems: To maintain high inference accuracy with low resource usage and fast response, video-analytics systems have explored many approaches, including model distillation [16, 20, 21], model architecture pruning [49, 50], configuration adaptation [32, 51], frame selection [52, 53], and DNN feature reusing [54, 55]. The closest to RECL is model distillation—creating lightweight models (i.e., experts in RECL) that are small and fast yet accurate on a specific video scene [16, 56]. The challenge is that as the video scene evolves, the system must create new expert models on the fly to fit new video content. Existing solutions rely on either of two approaches—model retraining techniques train the lightweight models on the latest video frames [19–21] or on the most relevant images from the training set [31], and model selection techniques maintain, and then select a model from, a collection of history models [23] or a cascade of models with increasing capacities [31, 57].

In contrast, RECL uses both techniques—model retraining and model selection—as building blocks in an end-to-end framework. In particular, when an edge device queries for a model update, RECL can respond faster than Ekya [21] and AMS [20] by selecting a model from a large collection of history models used by all edge devices which might have seen a similar scene and object distribution. RECL also shares GPU cycles to enable more concurrent model retraining jobs, refreshing new models for more edge devices.

Model selection under data drifts: In the ML literature, model selection in a collection of expert models, or Mixture-of-Experts (MoE), has attracted much attention, especially after Shazeer et al. [26] demonstrated that using a sparsely gating network with an MoE of many expert models can drastically reduce the compute cost of DNNs. Recent work has obtained accuracy comparable to state-of-the-art expensive models with a fraction of compute cost [58]. One key distinction between RECL and MoE applications is that in MoE, all or a subset of the experts work together on each input. However, in RECL, there only works one expert on each input. For example, the recent MoE approach [58] operating on *tokenized* images requires access to 768 experts for inference on each input image. To implement such an approach, one must either load all experts in the accelerator's memory or quickly swap the experts on the accelerator per patch per image, introducing significant challenges for even more resourceful settings such as entirely cloud-based applications [59]. That said, many techniques in MoE also assume that the MoE consists of a static set of models. To handle MoEs that gradually incorporate new models (as in RECL), the gating network or the model selector must be retrained over time [60, 61]. To avoid retraining model selectors or saving training data, recent works leverage an autoencoder that projects input data to a latent space and map new models to a region in the latent space [23, 34].

RECL's model selection strategy (§3.1) builds on the literature on gating networks [26], but reduces the delay and compute overhead when adding new expert models. Instead of jointly training the new experts and the gating network [26], RECL freezes the new expert models already trained to fit the edge devices' recent videos and only reshapes and fine-tunes the last layer of the gating network. Compared to recent autoencoder-based model selectors [23], RECL's gating network enjoys better algorithmic intuition (see §3.1) and better empirical performance (§5.2).

Resource allocation for DNNs: Resource sharing for DNN-related jobs has been extensively studied in the systems literature, including sharing of GPU and network resources among multiple concurrent DNN training jobs (e.g., [37, 62]), inference tasks of video analytics (e.g., [51, 63]), and between inference and training jobs [21]. The common challenge facing these settings is to predict how much each job's accuracy can improve with the same amount of compute/network resources.

This is usually profiled offline [51], periodically [21], or by reusing compute data [37].

RECL is a custom design of GPU sharing for continuous learning across many edge devices. Compared to Ekya [21], the most recent related work on edge continuous learning, RECL avoids profiling the training curve of each model; instead, it tracks the actual training progress of each retraining job on the fly, similar to SLAQ's quality-driven scheduler [37] proposed for large-scale DL clusters.

8 Conclusion

Resource efficiency is one of the most important problems in modern video analytics applications, and continuous retraining and deploying expert models is a promising direction. We show that reusing historical expert models has a large potential to improve resource efficiency and response time for continuous retraining, but this approach comes with its own challenges. We present RECL, the first end-to-end system that integrates model reusing with model retraining for resource-efficient video analytics. We show that RECL achieves significantly better resource efficiency and higher accuracy simultaneously than state-of-the-art baselines with *(i)* a fast and robust model selection procedure, *(ii)* a model zoo that shares across multiple edge devices, and *(iii)* an iterative training scheduler. We hope that our findings and designs can stimulate further research in unleashing the full potential of the synergy between model reusing and model retraining.

9 Acknowledgements

We thank the NSDI reviewers and our shepherd, Dongsu Han, for their invaluable feedback. This work was supported in part by NSF grants CNS-1751009, CNS-1955370, CNS-2152313, CNS-2153449, CNS-2147909, and CNS-2140552, as well as gifts from Cisco and the sponsors of MachineLearningApplications@CSAIL program.

References

[1] Iyiola E Olatunji and Chun-Hung Cheng. Video analytics for visual surveillance and applications: An overview and survey. *Machine Learning Paradigms*, pages 475–515, 2019.

[2] MarketsAndMarkets. Video analytics market with covid-19 impact, by component, application (intrusion management, incident detection, people/crowd counting, traffic monitoring), deployment model (on-premises and cloud), type, vertical, and region - global forecast to 2026. https://www.marketsandmarkets.com/Market-Reports/intelligent-video-analytics-market-778.html, 2021.

[3] Azure outposts. https://aws.amazon.com/outposts/.

[4] Azure stack edge. https://azure.microsoft.com/en-us/services/databox/edge/.

[5] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 50(10), 2017.

[6] Si Young Jang, Yoonhyung Lee, Byoungheon Shin, and Dongman Lee. Application-aware iot camera virtualization for video analytics edge computing. In *Symposium on Edge Computing (SEC)*, 2018.

[7] European Parliament. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46 (general data protection regulation). *Official Journal of the European Union (OJ)*, 59, 2016.

[8] Behrouz Jedari, Gopika Premsankar, Gazi Karam Illahi, Mario Di Francesco, Abbas Mehrabi, and Antti Ylä-Jääski. Video caching, analytics, and delivery at the wireless edge: A survey and future directions. *IEEE Commun. Surv. Tutorials*, 23(1), 2021.

[9] Ion Stoica. The future of computing is distributed. https://www.datanami.com/2020/02/26/the-future-of-computing-is-distributed/, 2020.

[10] Shadi A. Noghabi, Landon P. Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge computing. *GetMobile Mob. Comput. Commun.*, 23(4), 2019.

[11] Andrew Howard, Ruoming Pang, Hartwig Adam, Quoc V. Le, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, and Yukun Zhu. Searching for mobilenetv3. In *International Conference on Computer Vision (ICCV)*, 2019.

[12] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[13] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. YOLOX: exceeding YOLO series in 2021. *CoRR*, abs/2107.08430, 2021.

[14] Azure linux virtual machine pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.

[15] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. MCDNN: an approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[16] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, aug 2017.

[17] Microsoft Rocket for live video analytics. https://www.microsoft.com/en-us/research/project/live-video-analytics/, 2021.

[18] Sujith Ravi. Custom on-device ML models with Learn2Compress. https://ai.googleblog.com/2018/05/custom-on-device-ml-models.html, 2018.

[19] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, and Kayvon Fatahalian. Online model distillation for efficient video inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[20] Mehrdad Khani, Pouya Hamadanian, Arash Nasr-Esfahany, and Mohammad Alizadeh. Real-time video inference on edge devices via adaptive model streaming. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[21] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[22] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2019.

[23] Abhijit Suprem, Joy Arulraj, Calton Pu, and Joao Ferreira. Odin: Automated drift detection and recovery in video analytics. *Proc. VLDB Endow.*, 13(12):2453–2465, jul 2020.

[24] Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Comput.*, 6(2), 1994.

[25] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. *CoRR*, abs/2106.05974, 2021.

[26] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[27] Paperswithcode leaderboard of object detection on PASCAL VOC 2007 dataset. https://paperswithcode.com/sota/object-detection-on-pascal-voc-2007. Accessed: September 2022.

[28] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled Ben Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Commun. Surv. Tutorials*, 19(4), 2017.

[29] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015.

[30] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[31] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 3646–3654, 2017.

[32] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[33] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *IEEE/ACM Symposium on Edge Computing (SEC)*, 2020.

[34] Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. Expert gate: Lifelong learning with a network of experts. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3366–3375, 2017.

[35] Geoffrey Hinton. Introduction to neural networks and machine learning, lecture 15.

[36] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[37] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035, 2019.

[39] gRPC. https://grpc.io/about/.

[40] NVIDIA. TensorRT. https://developer.nvidia.com/tensorrt.

[41] Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. BDD100K: A diverse driving dataset for heterogeneous multitask learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[42] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[44] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[45] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[46] NVIDIA Jetson Nano. https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[47] Coral Edge TPU. https://coral.ai/docs/edgetpu/benchmarks/.

[48] CIPHERMODE Labs. https://www.ciphermode.tech/solutions-secureai. Accessed: September 2022.

[49] Ran Xu, Rakesh Kumar, Pengcheng Wang, Peter Bai, Ganga Meghanath, Somali Chaterji, Subrata Mitra, and Saurabh Bagchi. Approxnet: Content and contention-aware video object classification system for embedded clients. *ACM Transactions on Sensor Networks (TOSN)*, 18(1):1–27, 2021.

[50] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[51] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[52] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, 2020.

[53] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[54] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[55] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[56] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proc. VLDB Endow.*, 13(4):533–546, dec 2019.

[57] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Thia: Accelerating video analytics using early inference and fine-grained query planning. *arXiv preprint arXiv:2102.08481*, 2021.

[58] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. *Advances in Neural Information Processing Systems (NeurIPS)*, 34, 2021.

[59] Tutel: An efficient mixture-of-experts implementation for large DNN model training. https://www.microsoft.com/en-us/research/blog/tutel-an-efficient-mixture-of-experts-implementation-for-large-dnn-model-training/. Accessed: September 2022.

[60] Jeremy Z Kolter and Marcus A Maloof. Using additive expert ensembles to cope with concept drift. In *Proceedings of the 22nd international conference on Machine learning (ICML)*, pages 449–456, 2005.

[61] J Zico Kolter and Marcus A Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *The Journal of Machine Learning Research*, 8:2755–2790, 2007.

[62] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[63] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

Boggart: Towards General-Purpose Acceleration of Retrospective Video Analytics

Neil Agarwal, Ravi Netravali
Princeton University

Abstract

Commercial retrospective video analytics platforms have increasingly adopted general interfaces to support the custom queries and convolutional neural networks (CNNs) that different applications require. However, existing optimizations were designed for settings where CNNs were platform- (not user-) determined, and fail to meet at least one of the following key platform goals when that condition is violated: reliable accuracy, low latency, and minimal wasted work.

We present Boggart, a system that simultaneously meets all three goals while supporting the generality that today's platforms seek. Prior to queries being issued, Boggart carefully employs traditional computer vision algorithms to generate indices that are imprecise, but are fundamentally comprehensive across different CNNs/queries. For each issued query, Boggart employs new techniques to quickly characterize the imprecision of its index, and sparingly run CNNs (and propagate results to other frames) in a way that bounds accuracy drops. Our results highlight that Boggart's improved generality comes at low cost, with speedups that match (and most often, exceed) prior, model-specific approaches.

1 Introduction

Video cameras are prevalent in our society, with massive deployments across major cities and organizations [4, 10, 11, 22, 32, 47]. These cameras continually collect video data that is queried *retrospectively* to guide traffic/city planning, business or sports analytics, healthcare, crime investigation, and many other applications [5, 14, 23, 26, 31, 33–35, 37, 61, 69, 122]. Queries typically involve running convolutional neural network (CNN) models that locate and characterize particular objects in scenes [53, 99, 104, 106, 125]. Applications tailor the architectures and weights of those CNNs to their unique requirements (e.g., accuracy, latency, and resource cost) and target tasks, e.g., via specialization to scenes or object types [8, 13, 116], proprietary training datasets [7, 27, 28].

To support these diverse applications, commercial video analytics platforms (e.g., Microsoft Rocket [41, 44, 45], Amazon Rekognition [39], Google AI [70], IBM Maximo [83]) have steadily transitioned away from exposing only predetermined video processing results, towards being platforms that allow users/applications to register custom, large-scale video analytics jobs without worrying about infrastructural details [55, 116, 118]. To register a query, users typically provide (1) a CNN model of arbitrary architecture and weights, (2) a target set of videos (e.g., feeds, time periods), and (3) an accuracy target indicating how closely the provided results must match those from running the CNN on every frame. Higher accuracy targets typically warrant more inference (and thus, slower responses and higher costs).

From a platform perspective, there exist three main goals for each registered query. First and foremost, provided results should *reliably* meet the specified accuracy target (usually above 80% [80, 92, 105, 116]). Subject to that constraint, the platform should aim to consume as few computational resources as possible (i.e., minimize unnecessary work) and deliver responses as quickly as possible. The main difficulty in achieving these goals stems from the potentially massive number of video frames to consider, and the high compute costs associated with running a CNN on each one. For example, recent object detectors would require 500 GPU-hours to process a week of 30-fps video from just one camera [77, 82].

Unfortunately, despite significant effort in optimizing retrospective video analytics [42, 48, 80, 81, 93–95], no existing solution is able to simultaneously meet the above goals for the general interfaces that commercial platforms now offer. Most notably, recent optimizations perform ahead-of-time processing of video data to build indices that can accelerate downstream queries [48, 80, 95]. However, these optimizations were designed for settings where models were known *a priori* (i.e., not provided by users), and thus deeply integrate knowledge of the specific CNN into their ahead of time processing. Porting these approaches to today's bring-your-own-model platforms fundamentally results in unacceptable accuracy violations and resource overheads. The underlying reason is that models with even minor discrepancies (in architecture or weights) can deliver wildly different results for the same tasks and frames. Consequently, using different models for ahead-of-time processing and user queries can yield accuracy drops of up to 94% (§2.3). Building an index for all potential models is unrealistic given the massive space of CNNs [102, 107, 114, 149], and the inherent risk of wasted resources since queries may never be issued [80, 137].

In this paper, we ask *"can retrospective video analytics platforms operate more like general-purpose accelerators to achieve their goals for the heterogeneous queries+models provided by users?"* We argue that they can, but doing so requires an end-to-end rethink of the way queries are executed, from the ahead-of-time processing used to develop indices, to the execution that occurs only once a user provides a model and accuracy target. We examine the challenges associated with each phase, and present **Boggart**, a complete video analytics platform that addresses those challenges.

Ahead-of-time processing (indexing). To support our goals, an index must meet the following criteria: (1) comprehensive with respect to data of interest for different models/queries – any information loss would result in unpredictable accuracy

drops, (2) links information across frames so CNN inference results – the most expensive part of query execution [80, 94] – can be propagated from one frame to another at low cost, and (3) cheap to construct since queries may never come in.

We show with Boggart that, if applied in a *conservative* manner, traditional computer vision (CV) algorithms [52, 88, 100, 127] can be repurposed to generate such an index per video. Along these lines, Boggart's ahead-of-time processing extracts a comprehensive set of *potential* objects (or blobs) in each frame as areas of motion relative to the background scene. Trajectories linking blobs across frames are then computed by tracking low-level, model-agnostic video features, e.g., SIFT keypoints [110]. Crucially, Boggart's trajectories are computed once per video (not per video/model/query tuple) using cheap CV tasks that require only CPUs, and are generated 58% faster than prior model-specific indices constructed using compressed CNNs and GPUs (§6.3).

Query execution. Once a user registers a query and CNN, the main question is how to use the comprehensive index to quickly generate results that meet the accuracy target, i.e., running inference on as few frames as possible, and aggressively propagating results along Boggart's trajectories. The challenge is that Boggart's index is extremely coarse and imprecise relative to CNN results. For instance, blob bounding boxes may be far larger than those generated by CNNs, and may include multiple objects that move in tandem. Worse, the imprecision of Boggart's index varies with respect to different models and queries; prior systems avoid this issue by using indices that directly approximate specific models.

To handle this, Boggart introduces a new execution paradigm that first selects frames for CNN inference in a manner that sufficiently bounds the potential propagation error from index imprecision and unavoidable inconsistencies in CNN results [97]. The core idea is that such errors are largely determined by model-agnostic features about the video (e.g., scene dynamics), and can be discerned via inference on only a small set of representative frames. CNN results are then propagated using a custom set of accuracy-aware techniques that are specific to each query type (e.g., detection, classification) and robustly handle (and dynamically correct) imprecisions in Boggart's trajectories.

Results. We evaluated Boggart using 96 hours of video from 8 diverse scenes, a variety of CNNs, accuracy targets, and objects of interest, and 3 widely-used query types: binary classification, counting, and detection. Across these scenarios, Boggart consistently meets accuracy targets while running CNNs on only 3-54% of frames. Perhaps more surprisingly given its focus on generality and model-agnostic indices, Boggart outperforms existing systems that (1) rely solely on optimizations at query execution time (NoScope [94]) by 19-97%, and (2) use model-specific indices (Focus [80] running with knowledge of the exact CNN) by -5-58%.

Taken together, our results affirmatively answer the question above, showing that Boggart can support the general interfaces and diverse user models that commercial platforms face, while delivering reliable accuracy and comparable (typically larger) speedups than prior, model-specific optimizations. The source code and experimental data for Boggart are available at https://github.com/neilsagarwal/boggart.

2 Background and Motivation

In this section, we first present an overview of retrospective video analytics pipelines and their use cases (§2.1). We then describe existing optimizations (§2.2), and present measurements highlighting their inability to generalize to the different models and queries that users register (§2.3). Additional related work can be found in §7.

2.1 Primer on Retrospective Video Analytics

Numerous applications leverage (and are guided by) insights gleaned from analyzing the large amount of video data previously captured in different environments. For example, sports analytics tools leverage video analytics on previous game film to detect players on a field; these detections are fed into tracking algorithms to determine the efficacy of various strategies and to evaluate player performance [16, 29]. Similarly, retail analysts use video analytics to locate customers in indoor environments with high accuracy, in order to understand customer-product interaction and, ultimately, to improve store layout designs and product placement [31, 34]. City planners and traffic engineers employ video analytics to extract trends from historical footage, e.g., identifying points of congestion or opportunities for expansion [3, 20, 33, 35].

Despite their diverse use cases, retrospective video analytics generally share two main properties that characterize their computational requirements. First, they typically process video frames using convolutional neural networks (CNNs), a class of deep neural networks that have become the norm for automated vision processing due to their success in extracting spatial dependencies within images [53, 99, 104, 106, 125]. CNNs incorporate 3 kinds of layers: convolutional (responsible for recognizing pixel-level features), pooling (responsible for making these features more abstract), and fully-connected (responsible for using acquired features for prediction). In a CNN, each successive layer learns a more complex feature representation. Earlier layers focus on simple features such as colors and edges, while later layers aim to recognize specific objects. We refer the reader to prior reports [80, 94, 103] for more details.

Second, retrospective video analytics applications typically use CNNs to perform *object-centric* queries, e.g., to locate, characterize, and label different types of objects in frames. Indeed, the output of a CNN is a set of bounding boxes that localize all identified objects in a given frame, with each box being accompanied by a probability distribution characterizing its potential labels (or types). Such object-centric queries subsume those reported by both recent academic literature [55, 64, 92, 94, 105] and industrial organizations that run video analytics platforms [36, 80, 87, 111,

116, 140]. Concretely, in this paper, we consider the following query types (and accuracy metrics):

- **binary classification**: return a binary decision as to whether a specific object or type of object appears in each frame. Accuracy is measured as the fraction of frames tagged with the correct binary value.
- **counting**: return the number of objects of a given type that appear in each frame. Per-frame accuracy is set to the percent difference between the returned and correct counts.
- **bounding box detection**: return the coordinates for the bounding boxes that encapsulate each instance of a specific object or object type. Per-frame accuracy is measured as the mAP score [67], which considers the overlap (IOU) of each returned bounding box with the correct one.

The heterogeneity in use cases also brings important differences between manifestations of retrospective video analytics applications. Most notably, applications often apply *specialized* CNNs that cater to their specific target environments, object(s) of interest, required accuracy, task complexity, and available computational resources [75, 80, 94, 113]. Recent analyses of production video analytics workloads have shown that applications carry out such specialization by (1) selecting an existing reference model architecture from a popular family (e.g., ResNet, YOLO) and (2) training that model using custom and/or proprietary datasets that yield desirable weights for the target use case [116].

2.2 Existing Acceleration Approaches

Query-time strategies. Systems such as NoScope [94] and Tahoma [42] only operate once a user issues a query. To accelerate response generation, they first train cascades of cheaper binary classification CNNs that are specialized to the user-provided CNN, object of interest, and target video. The specific cascade to use is selected with the goal of meeting the accuracy target while minimizing computation and data loading costs. If confidence is lacking with regards to meeting the accuracy target, the user's CNN is incrementally run on frames until sufficient confidence is achieved.

Ahead-of-time (preprocessing) strategies. Other systems provide speedups by performing some computation ahead of time, i.e., before a query is issued; for ease of exposition, we refer to such computations as *preprocessing* in the rest of the paper. For example, Focus [80] speeds up binary classification queries by building an approximate, high-recall index of object occurrences using a specialized and compressed CNN that roughly matches the full CNN on the target video. Objects are then clustered based on the features extracted by the compressed model such that, during query execution, the full CNN only runs on the centroid of each cluster, with labels being propagated to all other objects in the same cluster.

BlazeIt [93] and TASTI [95] accelerate aggregate versions of certain query types, e.g., total counts across all frames. Preprocessing for both systems involves generating sampled results using the full CNN. TASTI uses the sampled results to train a cheap embedding CNN that runs on all frames and clusters those that are similar from the model's perspective. During query execution, the full CNN is run only on select frames in each cluster, with the results propagated to the rest. In contrast, BlazeIt uses the sampled results to train specialized CNNs that act as control variates for the remaining frames: the specialized CNNs run on all frames, and the results are correlated with sampled results from the full CNN to provide guarantees in statistical confidence. OTIF [48] follows a similar strategy, but uses proxy models (trained using the sampled results) to extract tracks about model-specific objects that are later used to accelerate tracking queries.

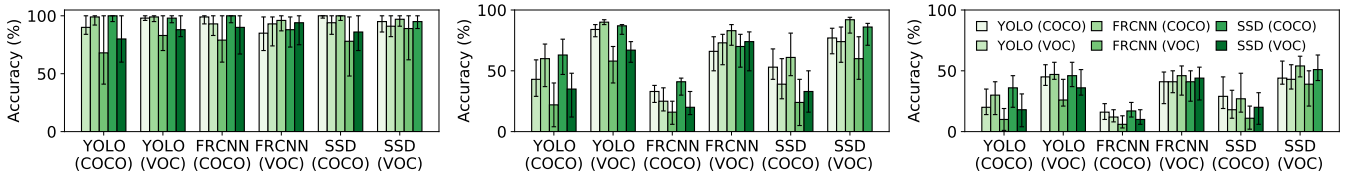
Videozilla [81] aims to extend such indexing optimizations across multiple video streams. More specifically, it identifies and exploits semantic similarities across streams that are based on the features extracted by the full CNN.

2.3 The Problem: Model-Specific Preprocessing

As confirmed by prior work [80, 93, 95] and our results in §6.3, preprocessing (intuitively) reduces the amount of computation required during query execution, and is crucial to enabling fast responses. However, *all* existing solutions suffer from the same fundamental issue: they deeply integrate a specific CNN into their preprocessing computations (e.g., to generate sampled results for training the compressed models used to build indices or group similarly-perceived frames), and assume that all future queries will use that same exact CNN. While such an approach was compatible with prior platforms that exposed only predetermined results from platform-selected CNN(s), it is no longer feasible with the bring-your-own-model interfaces that are now commonplace on commercial platforms. To make matters worse, consider that queries can be made at any point in the future and the space of potential CNNs is immense and rapidly evolving [102, 107, 114, 149], with variations in architecture (e.g., # of layers) or weights (e.g., different training datasets). In fact, building an index for even today's reference models would quickly present intractable resource challenges at the scale of retrospective video datasets: there exist tens of popular model families, each with multiple architecture options, e.g., the ResNet family alone has 8 architectures.

To quantify the issues when this assumption is violated, we ran experiments asking: how would accuracy be affected if the CNN provided by users during query execution (i.e., *query CNN*) was different than the CNN used during preprocessing (i.e., *preprocessing CNN*)? We consider the three query types above, videos and objects described in §6.1, and a wide range of CNNs: Faster RCNN, YOLOv3, and SSD, each trained on two datasets (COCO and VOC Pascal).

For each possible pair of preprocessing and query CNNs, we ran both CNNs on the video to obtain a list of object bounding boxes per frame. In line with Focus' observation that classification results from two CNNs may not identically match but should intersect for the top-k results [80], we ig-



(a) **Binary classification.** (b) **Counting.** (c) **Bounding box detection.**

Figure 1: **Query accuracies when *different* CNNs are used for preprocessing (bar types) and query execution (X axes). Bars show results for the median video, and error bars span the 25-75th percentiles. Models are listed as 'architecture (training dataset)'.**

nore the classifications from each CNN. Instead, we consider all bounding boxes from the preprocessing CNN that have an IOU of ≥ 0.5 with some box generated by the query CNN; results were largely unchanged for other IOU thresholds. This presents the *best scenario* (accuracy-wise) for existing preprocessing strategies. Finally, we compute query results separately using only the remaining preprocessing CNN's boxes or all of the query CNN's boxes, and compare them.

Figure 1 shows that discrepancies between preprocessing and query CNNs can lead to significant accuracy degradations, with the errors growing as query precision increases. For example, median degradations were 0-32% for binary classifications, but jump to 8-84% and 46-94% for counting and detections. Note that degradations for binary classification and counting are by definition due to the preprocessing CNN entirely missing objects relative to the query CNN. Parsed differently, median degradations across query types were 0-84%, 2-94%, and 1-90% when the preprocessing and query CNNs diverged in terms of only architecture, only weights, or both. Figure 2 shows that these degradations persist even for variants in the same family of CNNs.

Takeaway. Ultimately, when run on the general interfaces of today's commercial video analytics platforms where users can provide CNNs, all existing optimizations would sacrifice at least one key platform goal:

- *reliable accuracy:* running preprocessing optimizations as is (using platform-determined CNNs) would yield unpredictable and substantial (up to 94%) accuracy hits;
- *minimal wasted work:* performing preprocessing for all potential user CNNs is not only unrealistic given the sheer number of possibilities, but would also result in substantial wasted work since queries may never be issued;
- *low-latency responses:* optimizing only once a query is issued will yield higher than necessary response times.

3 Overview of Boggart

This section describes the overall workflow that Boggart uses to simultaneously meet all three platform goals for general, user-provided CNNs (Figure 3). §4 and §5 detail its preprocessing and query execution phases, and the project repository includes end-to-end visualizations of its operation [9].

Preprocessing. The main goal of Boggart's preprocessing phase is to perform cheap computations over a video dataset such that the outputs (an index) can accelerate query execution for diverse user CNNs, without sacrificing accuracy. Crucially, to avoid the pitfalls of prior work (§2.3), Boggart's preprocessing does not incorporate *any* knowledge of the specific CNN(s) that will be used during query execution. Instead, our insight is that traditional computer vision (CV) algorithms [88,100,124,127] are well-suited for such preprocessing, as they extract information purely about video data, rather than how a specific model or query would parse that data. Using generic CV algorithms enables Boggart to generate a single index per video, rather than per video/query/model tuple. Further, those CV algorithms are computationally cheaper than (even compressed) CNNs, and rely on CPUs (not GPUs), keeping monetary costs low (§6.3). Both aspects drastically reduce the potential for wasted work.

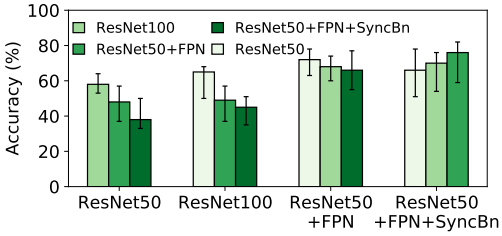


Figure 2: **Accuracies when CNNs for preprocessing (bar types) and query execution (X axis) are FasterRCNN+COCO with different ResNet backbones. Results are for counting queries; bars list medians with error bars for 25-75th percentiles.**

However, in contrast to their intended use cases, for our purposes, CV algorithms must be *conservatively* tuned to ensure that accuracy during query execution is not sacrificed. Namely, Boggart's index must *comprehensively* include all information that may influence or be incorporated in a query result (across CNNs), regardless of how coarse or imprecise that information is. Whereas coarse or imprecise results can be corrected or filtered out during query execution, missing information would result in unpredictable accuracy drops.

Accordingly, Boggart carefully uses a combination of motion extraction and low-level feature tracking techniques to identify all potential objects as areas of motion (or *blobs*) relative to a background estimate, and record their *trajectories* across frames by tracking each blob's defining pixels (or keypoints). For the former task, only high-confidence pixels are marked as being part of the background, ensuring that even minor motion is treated as a potential object; note that static objects are definitively discovered during query execution via CNN sampling on the frames across which the ob-

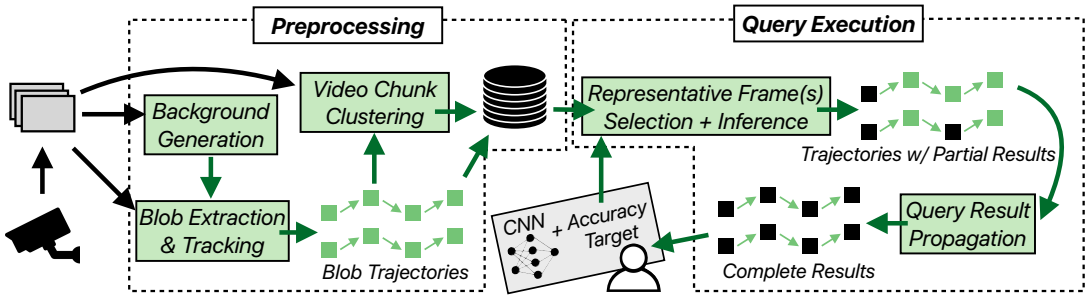


Figure 3: **Overview of Boggart.**

jects are static. For the latter task, any uncertainty in trajectory continuity (e.g., tracking ambiguities) is handled by simply starting a new trajectory; this ensures that results are not mistakenly propagated across different objects during query execution, albeit at the cost of additional inference. Overall, we did not observe *any* missed moving objects in Boggart's indices across our broad evaluation scenarios (§6.1).

Trajectories are a fundamental shift from the clustering strategies that prior systems use to group frames or objects based on how they are perceived by a specific CNN (§2.2). In contrast, trajectories are computed in a model-agnostic manner, but still provide a mechanism through which to propagate CNN results across frames during query execution – the primary source of speedups. Such generality does, however, come at a cost. Whereas prior efforts cluster on frames or object classes, Boggart's trajectories group frames on a per-object basis. This discrepancy lets Boggart defer the determination of how a user's CNN perceives each object to query execution, but it limits potential propagation, i.e., Boggart propagates the result for an object across the frames in which it appears, while prior approaches can propagate results across appearances of different objects. Note that this discrepancy does not apply to detection queries that require precise object locations (not just labels) to be propagated.

A natural question is: why not cluster objects on the features extracted by traditional CV algorithms to enable more result propagation? The issue is that, if performed without knowledge of the user-provided CNN, such clustering could lead to unsafe result propagation. More specifically, objects that are similar on some set of features but are perceived differently by the user's CNN could end up in the same cluster.

Query Execution. Once a user registers a query (providing a CNN, accuracy target, and video to consider), Boggart's goal is to generate a full set of per-frame results as quickly as possible, while reliably meeting the target accuracy. This translates to using the index from preprocessing (i.e., blobs and trajectories) to run the CNN on a small sample of frames, and efficiently propagate those results to the remaining frames.

The main challenge is that, owing to their general-purpose nature (relative to different models/queries) and closeness to noisy image signals, the CV algorithms used during preprocessing typically produce results that fail to precisely align with those from a user's CNN [55, 105]. Consequently, in being comprehensive, Boggart's index is coarse and imprecise relative to the target results from a user's CNN, e.g., with misaligned bounding boxes or extraneous objects that are not of interest to the query. Worse, the degree of imprecision is specific to the user CNN, and can lead to cascading errors (and accuracy drops) as results are propagated along Boggart's trajectories. All prior efforts avoid these issues by tuning indices to specific CNNs at the cost of generality.

To bound accuracy degradations (and reliably meet the specified target) while avoiding substantial inference, Boggart introduces a new query execution approach with two main components. First, to quickly and judiciously select the frames to run CNN inference on, our key observation is that errors from index imprecision and result propagation are largely dictated by model-agnostic features about the video, e.g., scene dynamics or trajectory lengths. Accordingly, Boggart clusters chunks of video in the dataset based on those features, and runs the user's CNN only on cluster centroids to determine the best frame selection strategy per cluster for the query at hand, i.e., the lowest frequency of CNN inference that meets the user-specified target accuracy. We note that, since clustering is based on model-agnostic features, it can be performed during preprocessing; CNN inference on centroids, however, only occurs once a user registers a query.

Second, to further limit inference overheads, Boggart introduces a new set of result propagation techniques that are specific to each query type and bolster propagation distances in spite of imprecisions in the index. For instance, for bounding box detections, Boggart leverages our empirical observation that the relative position between an object's keypoints (from preprocessing) and its bounding box edges remain stable over time. Building on this, Boggart propagates an object's CNN-produced bounding box to subsequent frames in its trajectory by efficiently searching for the coordinates that maximally preserve these spatial relationships.

Query model and assumptions. Boggart currently supports the large body of object-centric queries whose results are reported at the granularity of individual objects (e.g., labeling or locating them) and whose CNNs are run on a per-frame basis. Thus, currently handled queries include classifications, counting, and detections, as well as queries that build atop those primitives such as tracking and activity recognition. Such queries dominate the workloads reported

by commercial platforms [36, 80, 87, 116, 140], and subsume those supported by prior work (§2.2). We note that Boggart's approach is general enough to also accelerate less common, finer-grained queries, e.g., semantic segmentation [109]. For such queries, the keypoints (and their matches across frames) recorded in Boggart's index can be used to propagate groups of pixel labels; we leave implementing this to future work.

Boggart does not make any assumptions about or require any knowledge of the object type(s) that a query targets. Instead, as described above, Boggart relies on generic background estimation and motion extraction to identify potential objects. The intuition is that a moving object of any kind will involve (spatially correlated) moving pixels that can be identified purely based on the scene. Boggart leaves it to the user's CNN to determine whether those potential objects are of interest during query execution. We stress-test Boggart's robustness to different object types in §6.4.

Boggart's preprocessing operates on videos from static cameras that capture a single scene. Boggart currently does not support preprocessing for videos with changing backgrounds, e.g., CGI-generated films or videos from moving cameras. We note, however, that the CV community has actively been extending the core techniques that Boggart builds atop to deliver improved robustness in the face of moving cameras [65, 128, 135, 142]. We leave an exploration of integrating these efforts into Boggart to future work.

Reliance on Heuristics. Despite its focus on reliably meeting accuracy targets, Boggart's operation does involve multiple heuristics, i.e., tracking algorithms (§4), preset video chunk sizes (§4), thresholds for blob extraction (§4), and clustering parameters (§5.2). The upcoming sections and results in §6.4 elaborate on Boggart's sensitivity to each parameter. More generally, Boggart's approach to ensure sufficient accuracy is shared: each heuristic is conservatively configured to err on capturing too much data (resulting in unnecessary processing) rather than missing important data, i.e., prioritizing accuracy over efficiency. Examples include returning blobs for unlikely (but possible) objects, splitting trajectories upon uncertainty in object tracking, etc. §6 shows that this approach enables Boggart to consistently and efficiently deliver accurate query responses for diverse camera feeds, queries, models, objects, and accuracy targets.

4 Boggart's Preprocessing

Boggart's target output from preprocessing is a set of blobs and their trajectories. To efficiently extract this information and enable parallel processing over the dataset, Boggart operates independently on video chunks (i.e., groups of contiguous frames); the default chunk size is 1 min (profiled in §6.4), and trajectories are bound to individual chunks to eliminate any cross-chunk state sharing. The rest of this section describes the analysis that Boggart performs per chunk.

Background estimation. Extracting blobs inherently requires a point of reference against which to discern areas of motion. Thus, Boggart's first task is to generate an estimate of the background scene for the current chunk. However, existing background estimation approaches [46, 101] are ill-suited for Boggart as they are primarily concerned with generating a single, coherent background image despite scene dynamics (e.g., motion) that complicate perfect foreground-background separation. In contrast, Boggart's focus is on navigating the following tradeoff between accuracy and efficiency, not coherence. On one hand, placing truly background pixels in the foreground will lead to spurious trajectories (and query execution inefficiencies). On the other hand, incorrectly placing a temporarily static object in the background can result in accuracy degradations. Indeed, unlike entirely static objects that will surely be detected via CNN sampling and propagated to all frames in a chunk (during query execution), temporarily static objects may be missed and should only be propagated to select frames.

Boggart addresses the above tradeoff in a manner that favors accuracy. More specifically, Boggart only marks content as pertaining to the background scene when it has high confidence; all other content is conservatively marked as part of the foreground and is resolved during query execution. To realize this approach, Boggart eschews recent background estimation approaches in favor of a custom, lightweight strategy.

In its most basic form, background estimation involves recording the distribution of values assigned to each pixel (or region) across all frames in the chunk, and then marking the most frequently occurring value(s) (i.e., the peaks in the probability density function) as the background [124, 127]. This works well in scenarios where there is a clear peak in the distribution that accounts for most of the values, e.g., if objects do not pass through the pixel or do so with continuous motion, or if an object is entirely static and can thus be safely marked as the background. However, complications arise in settings with multiple peaks. For instance, consider a pixel with two peaks. Any combination of peaks could pertain to the background: a tree could sway back and forth (both), a single car could temporarily stop at a traffic light (one), or multiple cars could serially stop and go at the light (none).

To distinguish between these multi-modal cases and identify peaks that definitely pertain to the background for a chunk, Boggart extends (into the next chunk) the duration over which the distribution of pixel values is computed. The idea is that motion amongst background components should persist with more video, while cases with temporarily static objects should steadily transform into uni-modal distributions favoring either the background scene or the object (if it remains static). To distinguish between the object and background in the latter case, Boggart further extends the distribution of pixel values to incorporate video from the previous chunk. If the same peak continues to rise, it must pertain to the background since we know that the object was not static throughout the entire chunk. Otherwise, Boggart conservatively assigns an empty background for that pixel.

Figure 4: **Example screenshots from the Auburn video (Table 1). CNN (YOLOv3+COCO) detections are shown in white, while each of Boggart's trajectories (and their constituent blobs) is shown in a different color.**

Blob Extraction. Using the background estimate, Boggart takes a second pass through the chunk in order to extract areas of motion (blobs) on each frame. More specifically, Boggart segments each frame into a binary image whereby each pixel is annotated with a marker specifying whether it is in the foreground or background. Our implementation deems a pixel whose value falls within 5% of its counterpart(s) in the background estimate as a background pixel, but we find our results to be largely insensitive to this parameter. Given the noise in low-level pixel values [105], Boggart further refines the binary image using a series of morphological operations [115], e.g., to convert outliers in regions that are predominantly either background or foreground. Lastly, Boggart derives blobs by identifying components of connected foreground pixels [71], and assigning a bounding box using the top left and bottom right coordinates of each component.

Computing Trajectories. Boggart's final preprocessing task is to convert the set of per-frame blobs into trajectories that track each blob across the video chunk. At first glance, it may appear that sophisticated multi-object trackers (e.g., Kalman Filters) [50, 91, 134, 138] could directly perform this task. However, most existing trackers rely on pristine object detections as input. Blobs do not meet this criteria, and instead are far coarser and imprecise (Figure 4). At any time, a single blob may contain multiple objects, e.g., two people walking together. Blobs may split or merge as their constituent objects move and intersect. Lastly, the dimensions of a given object's blob bounding boxes can drastically fluctuate across frames based on interactions with the estimated background.

To handle these issues, we turn to tracking algorithms that incorporate low-level feature keypoints (SIFT [110] in particular) [88, 89], or pixels of potential interest in an image, e.g., the corners that *may* pertain to a car windshield. Associated with each keypoint is a descriptor that incorporates information about its surrounding region, and thus enables the keypoint (and its associated content) to be matched across images. Boggart conservatively applies this functionality to generate correspondences between blobs across frames.

For each pair of consecutive frames, Boggart pairs the constituent keypoints of each blob. This may yield any form of an N $\rightarrow$ N correspondence depending on the underlying tracking event, e.g., blobs entering/leaving a scene, fusion or splitting of blobs. For instance, if the keypoints in a blob on frame f_i match with keypoints in N different blobs on frame f_{i+1}, there is a 1 $\rightarrow$ N correspondence. To generate trajectories, Boggart makes a series of forwards and backwards scans through the chunk. For each correspondence that is not 1 $\rightarrow$ 1, Boggart propagates that information backwards to account for the observed merging or splitting. For example, for a 1 $\rightarrow$ N correspondence between frames f_i and f_{i+1}, Boggart would split f_i's blob into N components using the relative positions of the matched keypoints on f_{i+1} as a guide.

Index Storage. Preprocessing outputs are stored in MongoDB [1]; overheads are profiled in §6.4. Matched keypoints are stored with the corresponding frame IDs: row = [<((x,y)-coordinates, frame #)>]. Blob coordinates (and their trajectory IDs) are stored per frame to facilitate the matching of CNN results and blobs on sampled frames during query execution (§5.1): row = [<((x,y)-coordinates of top left corner, (x,y)-coordinates of bottom right corner, trajectory ID)>].

5 Fast, Accurate Query Execution

During query execution, Boggart's sole goal is to judiciously use the user-provided CNN and the index from preprocessing to quickly generate a complete set of results that meet the specified accuracy target. Doing so involves answering two questions: (1) what sampled (or *representative*) frames should the CNN be run on such that we can sufficiently adapt to the registered query (i.e., CNN, query type, and accuracy target) and bound errors from index imprecisions?, and (2) how can we use preprocessing outputs to accurately propagate sampled CNN results across frames for different query types? For ease of exposition, we describe (2) first, assuming CNN results on representative frames are already collected.

5.1 Propagating CNN Results

Regardless of the query type, Boggart's first task is to pair the CNN's bounding box detections on representative frames with the blobs on those same frames; this, in turn, associates detections with trajectories, and enables cross-frame result propagation. To do this, we pair each detection bounding box with the blob that exhibits the maximum, non-zero intersection. Trajectories that are not assigned to any detection are deemed spurious and are discarded. Further, detections with no matching blobs are marked as 'entirely static objects' and are handled after all other result propagation (described below). Note that, with this approach and in spite of the trajectory corrections from §4, multiple detections could be associated to a single blob, i.e., when objects move together and

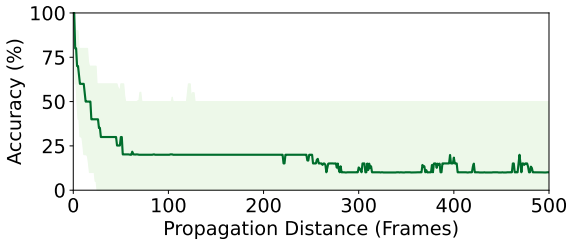


Figure 5: **Accuracy (mAP) degradations when CNN bounding boxes are propagated by computing the blob-to-detection coordinate transformation on a representative frame, and applying it to all other blobs in the trajectory. Line represents median detections, with ribbons spanning 25-75th percentiles.**

never separate. Using these associations, Boggart propagates CNN results via techniques specific to the target query type.

Binary classification and counting. To support both query types, each trajectory is labeled with an object count according to the number of detections associated with it on representative frames. If a trajectory passes through multiple representative frames, Boggart partitions the trajectory into segments, and assigns each segment a count based on the associations from the closest representative frame. Lastly, Boggart sums the counts across the trajectories that pass through each frame, and returns either the raw count (for counting), or a boolean indicating if count > 0 (for binary classification).

Bounding box detections. Whereas binary classification and count queries simply require propagating coarse information about object presence, bounding box queries require precise positional information to be shared across frames. However, as noted in §4, blobs and trajectories are inherently imprecise and fail to perfectly align with detections. A natural approach to addressing such discrepancies is to compute coordinate transformations between paired detections and blobs on representative frames, and apply those transformations to the remainder of each blob's trajectory; equivalently, one could compute transformations for a blob across its own trajectory, and apply them to add detections to non-representative frames. Unfortunately, Figure 5 shows that detection accuracy rapidly degrades with this approach, e.g., median degradations are 30% when propagating a box over 30 frames. The reason is that blobs and their paired detections move/resize differently across frames, resulting in median errors of 84% between the Euclidean distances of blob-blob and detection-detection coordinate transformations.

To fill the void of stable propagation mechanisms, Boggart leverages our finding that the relative positions between an object's constituent keypoints (i.e., those extracted and tracked during preprocessing) and its detection bounding box edges remain largely unchanged over short durations; we refer to these relative positions as *anchor ratios* since they 'anchor' an object's content to a relative position within the bounding box. This stability is illustrated in Figure 6, and is intuitive: objects tend to remain rigid over short time scales, implying that the points they are composed of move in much

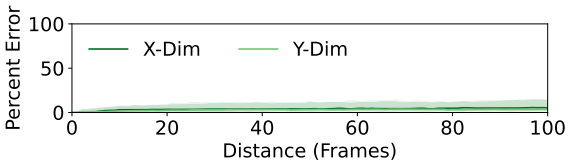


Figure 6: **Percent difference in anchor ratios for each object's keypoints across its trajectory. Lines show medians, with ribbons spanning 25-75th percentiles.**

the same way as the entire object does (including as the object scales in size). Building on this, Boggart propagates detections by using matching keypoints along the trajectories to which they have been associated, and efficiently solving an optimization problem in search of bounding box coordinates that maximally preserve the anchor ratios for each keypoint.

More formally, for each detection on each representative frame, Boggart considers the set of keypoints K that fall in the intersection with the associated blob. Each keypoint k in K has coordinates (x_k, y_k). Further, let the coordinates of the detection bounding box be (x_1, y_1, x_2, y_2), where (x_1, y_1) and (x_2, y_2) refer to the top left and bottom right corners. The anchor ratios (ax_k, ay_k) for keypoint k are computed as:

$$(ax_k, ay_k) = \left(\frac{x_2 - x_k}{x_2 - x_1}, \frac{y_2 - y_k}{y_2 - y_1}\right) \quad (1)$$

For each subsequent non-representative frame (until the next representative frame) that includes the same trajectory, Boggart finds the set of keypoints that match with those in K; denote the set of matching keypoints as K', where each k' in K' matches with keypoint k in K. Finally, to place the bounding box on the subsequent frame, Boggart solves for the corresponding coordinates (x_1, y_1, x_2, y_2) by minimizing the following function to maximally preserve anchor ratios:

$$\sum_{k'}^{K'} \left[\left(\frac{x_2 - x_{k'}}{x_2 - x_1} - ax_k \right)^2 + \left(\frac{y_2 - y_{k'}}{y_2 - y_1} - ay_k \right)^2 \right] \quad (2)$$

Note that this optimization (which takes 1 ms for the median detection) can be performed in parallel across frames and across detections on the same frame. Further, Boggart initializes each search with the coordinates of the corresponding detection box on the representative frame, thereby reducing the number of steps to reach a minima.

Propagating entirely static objects. Thus far, we have only discussed how to propagate detection bounding boxes that map to a blob/trajectory, i.e., moving objects. However, recall from §4 that certain objects which are entirely static will be folded into the background. These objects are discovered by the CNN on representative frames, but they will not be paired with any blob. Instead, Boggart broadcasts these objects to nearby frames (until the next representative frame) in a query-specific manner: such objects add to the per-frame counts used for classification and count queries, and their boxes are statically added into frames for detection queries.

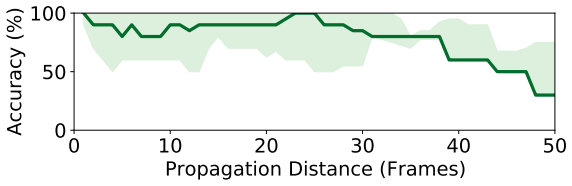


Figure 7: **Accuracy (mAP) degradations grow as Boggart propagates detection bounding boxes over longer durations. Results consider all object trajectories in the median video. Line represents medians, with ribbons spanning 25-75th percentiles.**

5.2 Selecting Representative Frames

To use the result propagation techniques from §5.1, we must determine the set of sampled, representative frames to collect CNN results on. Because CNN execution is the largest contributor to query execution delays (§6.4), we aim to select the smallest set of representative frames such that Boggart can sufficiently discern the relationship between its index and the CNN results, and generate a complete set of accurate results.

A natural strategy for selecting representative frames is to pick the smallest set of frames such that every trajectory appears at least once. In theory, executing the CNN on this set of frames should be sufficient to generate a result (e.g., object label, bounding box) for each trajectory, and propagate that result to all of the trajectory's frames. However, this straightforward approach falls short for two reasons:

1. CNNs can be inconsistent and occasionally produce different results for the same object across frames, e.g., a car in frame i may be ignored in frame $i+1$ [97, 98]. In line with prior analyses, we mostly observe this behavior for small or distant objects, e.g., YOLOv3 mAP scores are 18% and 42% for the small and large objects in the COCO dataset [120]. The consequence is that, if such an inconsistent result appears on a representative frame, Boggart would propagate it to all other frames in the trajectory, thereby spreading the error.
2. Even for consistent CNN results, propagation errors inherently grow with longer trajectories (i.e., as a given result is propagated to more frames). For instance, median accuracies are 90% and 30% when Boggart propagates bounding boxes over 10 and 50 frames (Figure 7).

These issues are more pronounced in busy/dynamic scenes with significant object occlusions/overlap [79, 132]. Moreover, the implication of both is that solely ensuring that the set of representative frames covers each trajectory is insufficient and can result in unacceptable accuracy degradations. To address this, Boggart introduces an additional constraint to the selection of representative frames: any blob in a trajectory must be within *max_distance* frames of a representative frame that contains the same trajectory. This, in turn, bounds both the duration over which inconsistent CNN results can be propagated, as well as the magnitude of propagation errors.

Tying back to our original goal, we seek the largest *max_distance* (and thus, fewest representative frames) that allows Boggart to meet the accuracy target. However, the appropriate *max_distance* depends on how the above issues manifest with the current query, CNN, and video. Digging deeper, we require an understanding of how Boggart's propagation techniques (for the query type at hand) and the user's CNN interact with each frame and trajectory, i.e., how accurate are Boggart's propagated results compared to the CNN's results. Though important for ensuring sufficient accuracy, collecting this data (particularly CNN results) for each frame during query execution would forego Boggart's speedups.

To achieve both accuracy and efficiency, Boggart clusters video chunks based on properties of the video and its index that characterize the aforementioned issues. The idea is that the chunks in each resulting cluster should exhibit similar interactions with the CNN and Boggart's result propagation, and thus should require similar *max_distance* values. Accordingly, Boggart could determine the appropriate *max_distance* for all chunks in a cluster by running the CNN and result propagation only on the cluster's centroid chunk.

To realize this approach, for each chunk, Boggart extracts distributions of the following features: object sizes (i.e., pixel area per blob), trajectory lengths (i.e., number of frames), and busyness (i.e., number of blobs per frame and trajectory intersections). These match our observations above: CNN inconsistencies are most abundant in frames with small objects, the potential for propagation errors is largest with long trajectories, and both issues are exacerbated in busy scenes.

With these features, Boggart clusters chunks using the K-means algorithm. We find that setting the number of target clusters to ensure that the centroids cover 2% of video strikes the best balance between CNN overheads and robustness to diverse and rapidly-changing video chunks; we profile this parameter in §6.4. Note that since clustering is based on model-agnostic features (from the extracted trajectories), it can be performed during preprocessing. Then, during query-execution, for each resulting cluster, Boggart runs the CNN on all frames in the centroid chunk. Using the collected results, Boggart runs its result propagation for a range of possible *max_distance* values, and computes an achieved accuracy for each one relative to the ground truth CNN results. More precisely, for each *max_distance*, Boggart selects the set of representative frames by greedily adding frames until our criteria is met, i.e., all blobs are within *max_distance* of the closest representative frame containing the same trajectory. From there, Boggart selects the largest *max_distance* that meets the specified accuracy goal, and applies it to pick representative frames for all other chunks in the same cluster.

Figure 8 highlights the effectiveness of Boggart's clustering strategy in terms of (quickly) adapting to different query types, accuracy targets, objects of interest, and CNNs. As shown in Figure 8(top), the median discrepancy between each chunk's ideal *max_distance* value and that of the corresponding cluster centroid is only 0-8 frames; this jumps to 45-898 frames when comparing chunks with the centroid

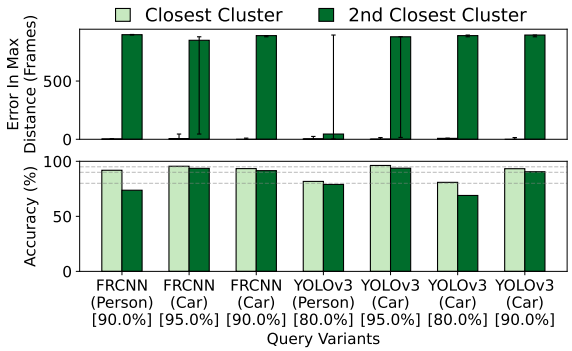


Figure 8: **Effectiveness of Boggart's clustering with different CNNs, (object types), and [accuracy targets]. Results are for the median video, and compare the ideal *max_distance* value for each chunk with those of the centroids in its cluster and the nearest neighboring cluster. The top graph measures the discrepancies in per-chunk *max_distance* (bars list medians, with error bars for 25-75th percentile); the bottom graph evaluates the corresponding hits on average accuracy (for detections).**

of the closest neighboring cluster. Figure 8(bottom) illustrates the importance of shrinking these discrepancies. More specifically, applying each centroid's ideal *max_distance* to all chunks in the corresponding cluster (i.e., Boggart's approach) yields average accuracies that are consistently above the targets. The same is not true when using the ideal *max_distance* values from the nearest neighboring cluster.

In summary, Boggart meets accuracy targets through co-analysis of the video content and specified query, i.e., object of interest, model, and query type. Boggart performs query/model-specific profiling of representative video chunks (where representative is defined by video content/-dynamics) to identify the frame inference strategy that most efficiently meets the accuracy target, and then executes this strategy for the remaining video chunks within each cluster.

6 Evaluation

We evaluated Boggart on a wide range of queries, CNNs, accuracy targets, and videos. Our key findings are:

- Boggart consistently meets accuracy targets while running the CNN on only 3-54% of frames, highlighting its comprehensive (model-agnostic) index and effective adaptation during query execution.
- Despite its goal of generality, Boggart's response times are 19-97% lower than NoScope's. Compared to Focus (which requires a priori knowledge of the CNN that will be used during query execution), Boggart's response times are 33% and 52% lower on counting and detection queries, and only 5% higher on classifications.
- Boggart's preprocessing (and index construction) runs 58% faster than Focus', while also generalizing to different CNNs/queries and requiring only CPUs (not GPUs).
- Boggart's preprocessing and query execution tasks speed up nearly linearly with increasing compute resources.

Camera location	Resolution
Auburn, AL (University crosswalk + intersection) [12]	1920×1080
Atlantic City, NJ (Boardwalk) [24]	1920×1080
Jackson Hole, WY (Crosswalk + intersection) [17]	1920×1080
Lausanne, CH (Street + sidewalk) [18]	1280×720
Calgary, CA (Street + sidewalk) [2]	1280×720
South Hampton, NY (Shopping village) [15]	1920×1080
Oxford, UK (Street + sidewalk) [21]	1920×1080
South Hampton, NY (Traffic intersection) [25]	1920×1080

Table 1: **Summary of our main video dataset.**

6.1 Methodology

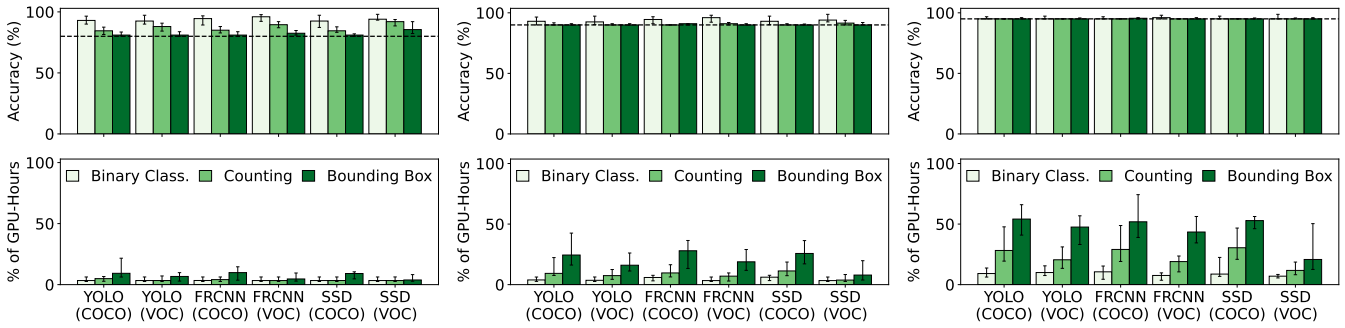
Videos. Table 1 summarizes the primary video sources used to evaluate Boggart. Video content across the cameras exhibits diversity in setting, resolution, and camera orientation (relative to the scene). From each camera, we scraped 12 hours of continuous video (at 30 fps) in order to capture varying levels of lighting and object densities (i.e., busyness). We consider additional videos and scene types in §6.4.

Queries. We consider the three query types (and their corresponding accuracy definitions) described in §2, i.e., binary classification, counting, and bounding box detection. For each type, we ran the query across our entire video dataset, and considered two objects of interest, people and cars, that cover drastically different size, motion, and rigidity properties; §6.4 presents results for additional object types. We evaluated Boggart with three accuracy targets – 80%, 90%, and 95% – and report accuracies as averages for each video. Accuracies are computed relative to running the model directly on all frames; as in prior systems and commercial platforms [41, 64, 80, 87, 105], Boggart does not aim to improve the accuracy of the provided model, and instead targets the same per-frame results at lower resource costs and delays.

CNN models. We consider three popular architectures: (1) SSD with a ResNet-50 backbone, (2) Faster RCNN with a ResNet-50 backbone, and (3) YOLOv3 with a Darknet53 backbone. For each, we used one version trained on the COCO dataset, and another trained on VOC Pascal. Trends for any results shown on a subset of CNNs (due to space constraints) hold for all considered models.

Hardware. Experiments used a server with an NVIDIA GTX 1080 GPU (8 GB RAM) and 18-core Intel Xeon 5220 CPU (2.20 GHz; 125 GB RAM), running Ubuntu 18.04.3.

Metrics. In addition to accuracy, we evaluate query execution performance of all considered systems (Boggart, Focus [80], and NoScope [94]) in terms of the number of GPU-hours required to generate results. We report GPU-hours for two reasons: (1) CNN execution (on GPUs) accounts for almost all response generation delays with all three systems, and (2) it is directly applicable to all of the systems, e.g., it incorporates NoScope's specialized CNNs. For preprocessing, we report both GPU- and CPU-hours since Boggart only requires the latter. As in prior work [80, 94], we exclude the video decoding costs shared by all considered systems.



(a) **80% Accuracy Target** (b) **90% Accuracy Target** (c) **95% Accuracy Target**

Figure 9: **Boggart's query execution performance across CNNs, query types, and accuracy targets; results are aggregated across object types. Bars summarize the distributions of per-video average result accuracy (top) and percentage of GPU-hours required to generate results relative to running the CNN on all frames (bottom). Bars list medians with error bars spanning 25-75th percentiles.**

Object Type →	People		Cars	
Query Type ↓	Acc.	% GPU-hrs	Acc.	% GPU-hrs
Binary Classif.	92%	6%	98%	3%
Counting	90%	11%	90%	7%
Bounding Box	91%	27%	90%	16%

Table 2: **Average accuracy and percentage of GPU-hours (relative to the naive baseline) for different query types and objects of interest. Results list median per-video values across all CNNs.**

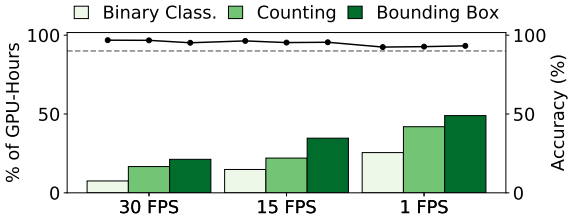


Figure 10: **Average accuracy (line) and percentage of GPU hours (relative to the naive baseline) for different video sampling rates. Results are listed for the median video, and consider YOLOv3+COCO and a 90% accuracy target.**

6.2 Query Execution Speedups

Figure 9 evaluates Boggart's query response times relative to a naive baseline that runs the CNN on all frames. Boggart always used the same, model-agnostic index per video.

There are three points to take away from Figure 9. First, across all of the conditions, Boggart *consistently* meets the specified accuracy targets. Second, the percentage of GPU-hours required to meet each accuracy target with Boggart grows as we move from coarse classification and counting queries to finer-grained bounding box detections. For example, with a target accuracy of 90%, the median percentage of GPU-hours across all models was 3-6%, 4-11%, and 8-28% for the three query types, respectively. Third, the percentage of GPU-hours also grows as the target accuracy increases for each query type. For instance, for counting queries, the percentage (across all CNNs) was 3-5% when the target accuracy was 80%; this jumps to 12-30% when the target accuracy grows to 95%. The reason is intuitive: higher accuracy targets imply that Boggart must more tightly bound the duration over which results are propagated (to limit propagation errors) by running the CNN on more frames.

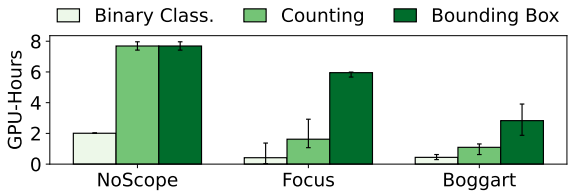
Different object types. Table 2 reports the results from Figure 9 separately per object type. As shown, the high-level trends from above persist for each. However, for a given query type, the percentage of required GPU-hours is consistently lower when considering cars versus people. The reason is twofold. First, inconsistencies in CNN results are more prevalent for people since they appear as smaller objects in our scenes (§5.2). Second, cars are inherently more rigid than people, and thus deliver more stability in the anchor ratios that Boggart relies on for bounding box propagation (§5.1); consequently, propagation errors for bounding box queries grow more quickly with people. Boggart handles both issues by running the CNN on more representative frames.

Downsampled video. Users may issue queries on sampled versions of each video [80]. We evaluated Boggart with three different sample rates: {30, 15, 1} fps. Although the number of considered frames drops, Figure 10 shows that Boggart's query execution speedups persist when operating over downsampled videos. For instance, with 1-fps video, Boggart requires only 25-49% of the GPU-hours that the naive baseline would need across all query types. Figure 10 also shows that Boggart's ability to consistently meet accuracy targets holds across sampling rates. We find that Boggart can hit accuracy targets without resorting to running the CNN on all frames because object keypoints – the primitive that Boggart tracks across frames during both trajectory construction (preprocessing) and detection propagation (query execution) – typically persist across frames even at these sample rates. For instance, Boggart matches 85% of the median object's keypoints across the 29-frame gap induced by the 1-fps rate.

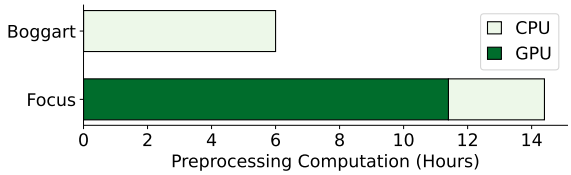
6.3 Comparison to State-of-the-Art

We compared Boggart with two recent retrospective video analytics systems: (1) NoScope [94], which only employs optimizations during query execution, and (2) Focus [80], which performs model-specific preprocessing by assuming a priori knowledge of user CNNs. §2.2 details each system.

For these experiments, we set the user-provided CNN to be YOLOv3+COCO, and the accuracy target to be 90%. Our



(a) **Query execution efficiency. Bars list values for the median video, with error bars spanning 25-75th percentiles.**



(b) **Preprocessing efficiency. Bars list GPU/CPU-hours for the median video. Note that NoScope does not perform preprocessing.**

Figure 11: **Comparing Boggart, Focus [80], and NoScope [94]. Results use YOLOv3+COCO and a target accuracy of 90%.**

Focus implementation used Tiny YOLO [120] as the specialized/compressed model (i.e., we ran Focus as if it knew the user CNN a priori), while NoScope used all of its open-source models. Following the training methodology used in both papers, we train the specialized/compressed models on 1-fps versions of the first half (i.e., 6 hours) of each video in our dataset, and run queries on the second half of each video.

Query Execution. Figure 11a compares the query response times of all three systems. As shown, Focus requires 5% fewer median GPU-hours than Boggart for binary classification queries. The main reason is that Focus' model-specific preprocessing (i.e., clustering of objects) enables more result propagation than Boggart's general, model-agnostic trajectories, i.e., Focus can propagate labels across objects, whereas Boggart can propagate labels only along a given object's trajectory (§3). Median propagation distances for results from the full CNN are 58 and 44 frames with Focus and Boggart.

Summing Focus' classifications to generate per-frame counts was insufficient for our 90% target. Thus, for counting queries, we performed favorable sampling until Focus hit 90% in each video: we greedily select a set of contiguous frames with constant count errors, run the CNN on a single frame, and correct errors on the remaining ones in the set. Even with such favorable sampling, Boggart required 33% fewer GPU-hours than Focus for counting queries.

Bounding box detections paint a starker contrast, with Boggart needing 52% fewer GPU-hours than Focus. Unlike with classification labels, Focus cannot propagate bounding boxes across frames. Instead, to accelerate these queries, Focus relies on binary classification, and runs the full CNN on all frames with an object of interest (to obtain their bounding boxes); for our videos, this translates to running the full CNN on 63-100% of frames. In contrast, Boggart propagates bounding boxes along each trajectory (median propagation distance of 23 frames) and reduces CNN tasks accordingly.

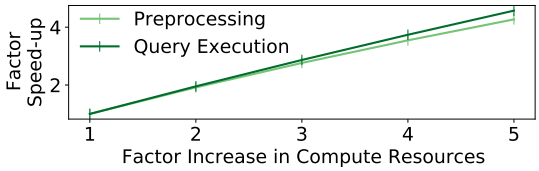


Figure 12: **Boggart's performance with increasing compute resources. Resource factors are multiples of the 18-core CPU and single GPU listed in §6.1. Results consider YOLOv3+COCO, a 90% accuracy target, and the median video.**

Compared to NoScope, Boggart's query execution tasks consume 19-97% fewer GPU-hours across query types. Boggart's speedups are largely due to three limitations with NoScope. First, NoScope does not perform preprocessing, and instead must train and run inference with its specialized and compressed CNNs during query execution. Second, results are not propagated across frames. Third, bounding box detections are sped up only via binary classification; note that NoScope performs binary classification on each frame (not object, like Focus), so we cannot simply sum the classification results to answer a counting query, and instead must execute counting queries as bounding box queries.

Preprocessing. Figure 11b shows that Boggart's preprocessing tasks take 58% fewer computation hours than Focus'. The discrepancy is from the training and inference costs that Focus incurs by using a specialized/compressed model. Note that *all* of Boggart's preprocessing is CPU-based, while Focus' costs are dominated (79%) by GPU operations. Further, Boggart's preprocessing runs once per video to support all future CNNs. To avoid accuracy drops (§2.3), Focus would have to run preprocessing for each CNN it wishes to support, leading to higher costs and potential for wasted work.

6.4 Profiling Boggart

Dissecting Boggart's performance. Boggart's preprocessing delays are dominated (83% on the median video) by the extraction of SIFT keypoints across frames; background estimation, trajectory construction, and clustering together account for only 17% of the time. Query execution profiles are similar, with CNN inference on centroid chunks and representative frames contributing 7% and 91% of runtime; result propagation (mostly for detections) takes the remaining 2%.

Resource scaling. Figure 12 shows that Boggart's preprocessing and query execution performance scale nearly linearly with increasing CPU and GPU resources, respectively. The reason is that feature extraction and CNN inference, the tasks that dominate delays in the two phases, inherently operate on a per-frame basis and can thus naturally be parallelized across frames. Note that these results only consider parallel processing within each chunk; Boggart can also parallelize across chunks since trajectories are bound to single chunks, i.e., there is no cross-chunk state sharing (§5).

Storage costs. Boggart's preprocessing generates, on average, 306 MB of data per 1 hour of video. For context, (1) the

average video in our dataset consumes 1 GB when encoded with H.264, and (2) Focus' preprocessing generates 70 MB of data for the same video. Recall that NoScope does not involve video preprocessing, and thus does not incur storage costs for indices. Note that 98% of Boggart's storage overheads are for keypoints used to propagate bounding boxes; blobs and trajectories consume only 2%.

Sensitivity to parameters. Boggart includes parameters for video chunk size (default: 1 min) and target number of clusters (default: centroids cover 2% of video). On average, we find that Boggart's performance is largely insensitive to both: varying chunk sizes from 0.2-10 min and the videos covered by centroids from 0.5-5% altered Boggart's performance by less than 5% (note that accuracy never dropped below the targets). However, the effects of each parameter are more pronounced on short amounts of video and are dependent on the content being considered. More specifically, smaller chunk sizes reduce the potential result propagation, but also shrink cluster centroids and increase the potential for parallel processing. Similarly, more clusters implies fewer suboptimalities in the selection of representative frames, but also additional centroids on which to run the CNN.

Generalizability. To further evaluate Boggart's ability to generalize, we ran experiments with three additional videos (3 hours each) and new object types specific to those scenes: birds in nature [19], boats in a canal [30], and people, cups, chairs, and tables in a restaurant [6]. For these experiments, we ran Boggart in the same way as above, i.e., it is not tuned in any way to the video or objects of interest. We also ran experiments considering different object types (trucks and bicycles) in the traffic videos from Table 1; these experiments used the same indices as in our main evaluation. All results exhibit similar trends as above, with Boggart always meeting accuracy targets (80%, 90%, 95%) and running CNNs on only 11.7-34.2%, 11.7-53.4%, and 12.6-56.7% of frames for binary classification, counting, and detection.

7 Additional Related Work

Live video analytics. Multiple systems accelerate queries on live video, with optimizations along the following axes: (1) profiling pipeline knobs to identify cheaper (but accurate) configurations [87, 140], (2) integrating on-camera or edge server resources for partial inference, frame filtering, or reusing results from prior frames [43,54,58,63,66,73,74, 105,129,136,141,148], (3) content/model-aware encoding to reduce data transfers [64, 133], and (4) spatiotemporal coordination for efficient multi-camera queries [86, 111]. These systems target an entirely different computational model (stream processing vs. "after-the-fact" querying) and thus face a different set of goals, optimization knobs, and constraints, e.g., by not having the entire dataset up front, live analytics can only propagate results to later frames.

Accelerating GPU tasks. One line of work optimizes DNNs for accelerated inference via distillation [78], quantization [60, 84, 144], or pruning [51, 108]. Another direction targets faster inference for a model, either through better scheduling of GPU resources across inference tasks [85,123, 126], or hardware acceleration [38,68,90,117]. These works are complementary to Boggart, which focuses on reducing the number of frames on which inference must be performed.

Video Object Detection. In addition to those in §4, Boggart builds on a line of work in the CV community that leverages the spatiotemporal aspect of video to accelerate detection and classification tasks. These techniques swap inference on sampled frames with optical flow networks that extend results from earlier frames [49, 56, 57, 59, 62, 72, 76, 98, 112, 130,131,145–147], and are thus similar in spirit to Boggart's result propagation strategy. However, unlike Boggart, these approaches are model-specific, in that the networks used for propagation must be trained according to the specific CNN (e.g., its feature extractor) used in the target query.

Video storage and indexing. Many systems balance video storage and lookup costs for specific query types [121, 139, 143] or CNNs [40, 96, 119, 137]. Boggart is complementary to these works in that its focus is on performing generalizable preprocessing and accelerating response generation after video frames are loaded into memory.

8 Conclusion

This paper described Boggart, a system for retrospective video analytics that supports the general "bring your own model" interfaces that are now commonplace in commercial platforms. To meet the core accuracy, speed, and efficiency goals of those platforms, Boggart holistically rethinks the query execution process, introducing cheap techniques to generate comprehensive (but imprecise) indices during preprocessing, and later use those indices to limit costly inference while bounding accuracy drops from imprecisions. Our results show that such generality can come at low cost, as Boggart outperforms prior, model-specific approaches.

Ethics. The focus of this work is on making the ethical processing of videos (public or private, according to the law) more efficient. We do not advocate for the processing of video for illicit purposes, unlawful tracking, etc. Moreover, Boggart is developed to improve the resource efficiency of existing retrospective video analytics platforms in a manner that does not change the interfaces they expose, i.e., the videos, models/queries, and customers they handle remain unchanged. In sum, Boggart does not alter the set of information exposed to applications – the videos that an application can query and the queries that the application can run on those videos are unchanged, and Boggart's internal state (e.g., preprocessing results) is not exposed.

Acknowledgements. We thank Ganesh Ananthanarayanan, Amit Levy, Jennifer Rexford, the NSDI reviewers, and our shepherd, Siddhartha Sen, for their valuable feedback and constructive comments. This work was supported by a Sloan Research Fellowship, a Cisco grant, and NSF CNS grants 2152313, 2153449, 2147909, and 2140552.

REFERENCES

[1] https://www.mongodb.com/.

[2] 11 Street SW Calgary. https://www.youtube.com/watch?v=iGxFLjqhkSA.

[3] 3 Reasons Why City Planners Need Video Analytics. https://www.briefcam.com/resources/blog/3-reasons-why-city-planners-need-video-analytics/.

[4] Absolutely everywhere in beijing is now covered by police video surveillance. https://qz.com/518874/.

[5] Are we ready for ai-powered security cameras? https://thenewstack.io/are-we-ready-for-ai-powered-security-cameras/.

[6] Beach Bar St. John Webcam. https://www.youtube.com/watch?v=2wqpy036z24.

[7] Betterview Combines Computer Vision and Post-Event Imagery to Map Tornado Damage. https://blog.betterview.com/betterview-combines-computer-vision-and-post-event-imagery-to-quickly-map-tornado-damage.

[8] Bird Cams Lab. https://www.zooniverse.org/organizations/cornellbirdcams/bird-cams-lab.

[9] Boggart Repository. https://github.com/neilsagarwal/boggart.

[10] British transport police: Cctv. http://www.btp.police.uk/advice_and_information/safety_on_and_near_the_railway/cctv.aspx.

[11] Can 30,000 cameras help solve chicago's crime problem? https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html.

[12] City of Auburn Toomer's Corner Webcam 1. https://www.youtube.com/watch?v=wVDtzDwo-1Q.

[13] Computer Vision AI. https://techsee.me/computer-vision/.

[14] Global Sports Analytics Market Size Report, 2021-2028. https://www.grandviewresearch.com/industry-analysis/sports-analytics-market.

[15] Hamptons.com Southampton Village Cam, Hildreth's Home Goods LIVE. https://www.youtube.com/watch?v=9IbruokZzx0.

[16] How to Be Ahead of Your Competition with Data. https://www.hudl.com/blog/how-to-be-ahead-of-your-competition-with-data.

[17] Jackson Hole Wyoming USA Town Square Live Cam. https://www.youtube.com/watch?v=1EiC9bvVGnk.

[18] Lausanne, pont Bessières. https://www.youtube.com/watch?v=TyElel0QjCI.

[19] Live BACKYARD Animal Cam in Ohio! . https://www.youtube.com/watch?v=OIqUka8BOS8.

[20] One traffic framework. Any video source. All traffic tasks. https://datafromsky.com/.

[21] Oxford Martin School Webcam - Broad Street, Oxford. https://www.youtube.com/watch?v=St7aTfoIdYQ.

[22] Paris hospitals to get 1,500 cctv cameras to combat violence against staff. https://bit.ly/2OYiBz2.

[23] Powering the edge with ai in an iot world. https://www.forbes.com/sites/forbestechcouncil/2020/04/06/powering-the-edge-with-ai-in-an-iot-world/.

[24] Resorts Casino Hotel Beach Camera. https://www.youtube.com/watch?v=vVyBOU9Huvo.

[25] SouthHampton Traffic Cam. https://www.youtube.com/watch?v=Z9P_2pCgfBA.

[26] The Hudl Algorithm: Turning Video into Player Tracking Data. https://www.maryecollins.com/hudl-tracking.

[27] Toyota Research Institute accelerates safe automated driving with deep learning. https://www.wired.com/brandlab/2018/08/tri-accelerates-safe-automated-driving-deep-learning-2/.

[28] Unique web-based facial recognition tool enhances security and fights crime. https://www.securityinfowatch.com/access-identity/biometrics/facial-recognition-solutions/article/21261325/unique-webbased-facial-recognition-tool-enhances-security-and-fights-crime.

[29] Using Deep Learning to Find Basketball Highlights. https://www.hudl.com/bits/using-deep-learning-to-find-basketball-highlights.

[30] Venice Italy Live Camera - Grand Canal. https://www.youtube.com/watch?v=P393gTj527k.

[31] Video analytics applications in retail - beyond security. https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html/.

[32] Video Analytics Market - Growth, Trends, COVID-19 Impact, and Forecasts (2022 - 2027). https://www.mordorintelligence.com/industry-reports/video-analytics-market.

[33] The vision zero initiative. http://www.visionzeroinitiative.com/.

[34] How retail stores can streamline operations with video content analytics. https://www.briefcam.com/resources/blog/how-retail-stores-can-streamline-operations-with-video-content-analytics/, 2020.

[35] Video analytics traffic study creates baseline for change. https://www.govtech.com/analytics/Video-Analytics-Traffic-Study-Creates-Baseline-for-Change.html, 2020.

[36] Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, Apr. 2022. USENIX Association.

[37] Video analytics market. https://www.fortunebusinessinsights.com/industry-reports/video-analytics-market-101114, 2022.

[38] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos. Bit-

pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 382–394. Association for Computing Machinery, 2017.

[39] Amazon. Rekognition. https://aws.amazon.com/rekognition/.

[40] Amazon. AWS DeepLens. https://aws.amazon.com/deeplens/, 2019.

[41] G. Ananthanarayanan, Y. Shu, M. Kasap, A. Kewalramani, M. Gada, and V. Bahl. Live video analytics with microsoft rocket for reducing edge compute costs, July 2020.

[42] M. R. Anderson, M. J. Cafarella, G. Ros, and T. F. Wenisch. Physical representation-based predicate optimization for a visual analytics database. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 1466–1477, 2019.

[43] K. Apicharttrisorn, X. Ran, J. Chen, S. V. Krishnamurthy, and A. K. Roy-Chowdhury. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 96–109, New York, NY, USA, 2019. Association for Computing Machinery.

[44] M. Azure. Computer vision api. https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/, 2021.

[45] M. Azure. Face api. https://azure.microsoft.com/en-us/services/cognitive-services/face/, 2021.

[46] O. Barnich and M. Van Droogenbroeck. Vibe: A universal background subtraction algorithm for video sequences. *IEEE Transactions on Image processing*, 20(6):1709–1724, 2010.

[47] D. Barrett. One surveillance camera for every 11 people in Britain, says CCTV survey. https://www.telegraph.co.uk/technology/10172298/\One-surveillance-camera-for-every-11-people-in-Britain\-says-CCTV-survey.html, 2013.

[48] F. Bastani and S. Madden. Otif: Efficient tracker preprocessing over large video datasets. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, 2022.

[49] G. Bertasius, L. Torresani, and J. Shi. Object detection in video with spatiotemporal sampling networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 331–346, 2018.

[50] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft. Simple online and realtime tracking. In *2016 IEEE international conference on image processing (ICIP)*, pages 3464–3468. IEEE, 2016.

[51] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

[52] S. Brutzer, B. Hoferlin, and G. Heidemann. Evaluation of background subtraction techniques for video surveillance. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pages 1937–1944, Washington, DC, USA, 2011. IEEE Computer Society.

[53] Z. Cai, M. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 3361–3369, Washington, DC, USA, 2015. IEEE Computer Society.

[54] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor. Scaling video analytics on constrained edge nodes. In *2nd SysML Conference*, 2019.

[55] F. Cangialosi, N. Agarwal, V. Arun, J. Jiang, S. Narayana, A. Sarwate, and R. Netravali. Privid: Practical, privacy-preserving video analytics queries. In *Proceedings of the 19th USENIX Conference on Networked Systems Design and Implementation*, NSDI'22, Berkeley, CA, USA, 2022. USENIX Association.

[56] Y. Chai. Patchwork: A patch-wise attention network for efficient object detection and segmentation in video streams. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3415–3424, 2019.

[57] K. Chen, J. Wang, S. Yang, X. Zhang, Y. Xiong, C. C. Loy, and D. Lin. Optimizing video object detection via a scale-time lattice. In *CVPR*, 2018.

[58] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[59] Y. Chen, Y. Cao, H. Hu, and L. Wang. Memory enhanced global-local aggregation for video object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10337–10346, 2020.

[60] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[61] S. R. E. Datondji, Y. Dupuis, P. Subirats, and P. Vasseur. A survey of vision-based traffic monitoring of road intersections. *Trans. Intell. Transport. Sys.*, 17(10):2681–2698, Oct. 2016.

[62] J. Deng, Y. Pan, T. Yao, W. Zhou, H. Li, and T. Mei. Relation distillation networks for video object detection. In *Proceedings of the IEEE/CVF International*

Conference on Computer Vision, pages 7023–7032, 2019.

[63] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan. Cachier: Edge-caching for recognition applications. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 276–286, 2017.

[64] K. Du, A. Pervaiz, X. Yuan, A. Chowdhery, Q. Zhang, H. Hoffmann, and J. Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 557–570, New York, NY, USA, 2020. Association for Computing Machinery.

[65] A. Elqursh and A. Elgammal. Online moving camera background subtraction. In A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, editors, *Computer Vision – ECCV 2012*, pages 228–241, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[66] J. Emmons, S. Fouladi, G. Ananthanarayanan, S. Venkataraman, S. Savarese, and K. Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, HotEdgeVideo'19, pages 27–32, New York, NY, USA, 2019. Association for Computing Machinery.

[67] M. Everingham, L. Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010.

[68] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 1–14. IEEE Press, 2018.

[69] I. Ghodgaonkar, S. Chakraborty, V. Banna, S. Allcroft, M. Metwaly, F. Bordwell, K. Kimura, X. Zhao, A. Goel, C. Tung, et al. Analyzing worldwide social distancing through large-scale computer vision. *arXiv preprint arXiv:2008.12363*, 2020.

[70] Google. Cloud vision api. https://cloud.google.com/vision, 2021.

[71] C. Grana, D. Borghesani, and R. Cucchiara. Optimized block-based connected components labeling with decision trees. *IEEE Transactions on Image Processing*, 19(6):1596–1609, 2010.

[72] C. Guo, B. Fan, J. Gu, Q. Zhang, S. Xiang, V. Prinet, and C. Pan. Progressive sparse local attention for video object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3909–3918, 2019.

[73] P. Guo, B. Hu, R. Li, and W. Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 19–34, New York, NY, USA, 2018. Association for Computing Machinery.

[74] P. Guo and W. Hu. *Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications*, page 271–284. Association for Computing Machinery, New York, NY, USA, 2018.

[75] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.

[76] F. He, N. Gao, Q. Li, S. Du, X. Zhao, and K. Huang. Temporal context enhanced feature aggregation for video object detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 10941–10948, 2020.

[77] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[78] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[79] D. Hoiem, Y. Chodpathumwan, and Q. Dai. Diagnosing error in object detectors. In *European conference on computer vision*, pages 340–353. Springer, 2012.

[80] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, 2018. USENIX Association.

[81] B. Hu, P. Guo, and W. Hu. Video-zilla: An indexing layer for scalable live video analytics. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, 2022.

[82] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *CoRR*, abs/1611.10012, 2016.

[83] IBM. Maximo remote monitoring. https://www.ibm.com/products/maximo/remote-monitoring, 2021.

[84] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang,

A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[85] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041*, 2018.

[86] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, V. Bahl, and J. Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing (SEC 2020)*, November 2020.

[87] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 253–266, New York, NY, USA, 2018. ACM.

[88] J. Jodoin, G. Bilodeau, and N. Saunier. Urban tracker: Multiple object tracking in urban mixed traffic. In *IEEE Winter Conference on Applications of Computer Vision*, pages 885–892, 2014.

[89] J. Jodoin, G. Bilodeau, and N. Saunier. Tracking all road users at multimodal urban traffic intersections. *IEEE Transactions on Intelligent Transportation Systems*, 17(11):3241–3251, 2016.

[90] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[91] C. Ju, Z. Wang, C. Long, X. Zhang, G. Cong, and D. E. Chang. Interaction-aware kalman neural networks for trajectory prediction. *CoRR*, abs/1902.10928, 2019.

[92] D. Kang, P. Bailis, and M. Zaharia. Blazeit: Fast exploratory video queries using neural networks. *CoRR*, abs/1805.01046, 2018.

[93] D. Kang, P. Bailis, and M. Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proc. VLDB Endow.*, 13(4):533–546, Dec. 2019.

[94] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, Aug. 2017.

[95] D. Kang, J. Guibas, P. Bailis, T. Hashimoto, and M. Zaharia. Task-agnostic indexes for deep learning-based queries over unstructured data, 2020.

[96] D. Kang, A. Mathur, T. Veeramacheneni, P. Bailis, and M. Zaharia. Jointly optimizing preprocessing and inference for dnn-based visual analytics. *Proc. VLDB Endow.*, 14(2):87–100, Oct. 2020.

[97] K. Kang, H. Li, J. Yan, X. Zeng, B. Yang, T. Xiao, C. Zhang, Z. Wang, R. Wang, X. Wang, and W. Ouyang. T-CNN: Tubelets With Convolutional Neural Networks for Object Detection From Videos. *IEEE Trans. Cir. and Sys. for Video Technol.*, 28(10):2896–2907, Oct. 2018.

[98] K. Kang, W. Ouyang, H. Li, and X. Wang. Object detection from video tubelets with convolutional neural networks. In *CVPR*, 2016.

[99] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[100] B. Kueng, E. Mueggler, G. Gallego, and D. Scaramuzza. Low-latency visual odometry using event-based feature tracks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 16–23, Oct 2016.

[101] B. Laugraud, S. Piérard, and M. Van Droogenbroeck. Labgen: A method based on motion detection for generating the background of a scene. *Pattern Recognition Letters*, 96:12–21, 2017.

[102] J. Le. Part 1: An overview of dataops for computer vision. https://www.superb-ai.com/blog/part-1-an-overview-of-dataops-for-computer-vision, 2021.

[103] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[104] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, June 2015.

[105] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. SIGCOMM '20, page 359–376, New York, NY, USA, 2020. Association for Computing Machinery.

[106] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, July 2017.

[107] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen. Deep learning for generic object detection: A survey, 2019.

[108] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[109] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, Los Alamitos, CA, USA, jun 2015. IEEE Computer Society.

[110] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004.

[111] Y. Lu, A. Chowdhery, and S. Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 57–70, New York, NY, USA, 2016. ACM.

[112] H. Mao, T. Kong, and W. J. Dally. Catdet: Cascaded tracked detector for efficient object detection from video. *arXiv preprint arXiv:1810.00434*, 2018.

[113] A. Mhalla, T. Chateau, H. Maamatou, S. Gazzah, and N. E. B. Amara. Smc faster r-cnn: Toward a scene-specialized multi-object detector. *Computer Vision and Image Understanding*, 164:3–15, 2017.

[114] A. Moschitti. Updating neural networks to recognize new categories, with minimal retraining. https://www.amazon.science/blog/updating-neural-networks-to-recognize-new-categories-with-minimal-retraining, 2019.

[115] OpenCV. Morphological Transformations. https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html, 2020.

[116] A. Padmanabhan, N. Agarwal, A. Iyer, G. Ananthanarayanan, Y. Shu, N. Karianakis, G. H. Xu, and R. Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge, 2022.

[117] S. Park, J. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H. Yoo. An energy-efficient and scalable deep learning/inference processor with tetra-parallel mimd architecture for big data applications. *IEEE Transactions on Biomedical Circuits and Systems*, 9(6):838–848, 2015.

[118] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-preserving video analytics as a cloud service. In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1039–1056. USENIX Association, 2020.

[119] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian. Scanner: Efficient video analysis at scale. *ACM Trans. Graph.*, 37(4), July 2018.

[120] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[121] W. Ren, S. Singh, M. Singh, and Y. S. Zhu. State-of-the-art on spatio-temporal information-based video retrieval. *Pattern Recogn.*, 42(2):267–282, Feb. 2009.

[122] A. Rizzoli. 7 Game-Changing AI Applications in the Sports Industry. https://www.v7labs.com/blog/ai-in-sports, 2022.

[123] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[124] C. Stauffer and W. E. L. Grimson. Adaptive background mixture models for real-time tracking. In *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, volume 2, pages 246–252 Vol. 2, 1999.

[125] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, pages 3476–3483, Washington, DC, USA, 2013. IEEE Computer Society.

[126] Y. Ukidave, X. Li, and D. Kaeli. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 353–362. IEEE, 2016.

[127] P. D. Z. Varcheie, M. Sills-Lavoie, and G.-A. Bilodeau. A multiscale region-based motion detection and background subtraction algorithm. *Sensors*, 10(2):1041–1061, 2010.

[128] A. Viswanath, R. K. Behera, V. Senthamilarasu, and K. Kutty. Background modelling from a moving camera. volume 58, pages 289–296, 2015. Second International Symposium on Computer Vision and the Internet (VisionNet'15).

[129] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. pages 159–173, 10 2018.

[130] S. Wang, H. Lu, and Z. Deng. Fast object detection in compressed video. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages

7104–7113, 2019.

[131] S. Wang, Y. Zhou, J. Yan, and Z. Deng. Fully motion-aware network for video object detection. In *Proceedings of the European conference on computer vision (ECCV)*, pages 542–557, 2018.

[132] X. Wang, T. Xiao, Y. Jiang, S. Shao, J. Sun, and C. Shen. Repulsion loss: Detecting pedestrians in a crowd. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7774–7783, 2018.

[133] Y. Wang, W. Wang, J. Zhang, J. Jiang, and K. Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[134] N. Wojke, A. Bewley, and D. Paulus. Simple online and realtime tracking with a deep association metric, 2017.

[135] S. Xie, W. Zhang, W. Ying, and K. Zakim. Fast detecting moving objects in moving background using orb feature matching. In *2013 Fourth International Conference on Intelligent Control and Information Processing (ICICIP)*, pages 304–309, 2013.

[136] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 129–144, New York, NY, USA, 2018. Association for Computing Machinery.

[137] T. Xu, L. M. Botelho, and F. X. Lin. Vstore: A data store for analytics on large videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 16:1–16:17, New York, NY, USA, 2019. ACM.

[138] Q. Xue, X. Li, J. Zhao, and W. Zhang. Deep kalman filter: A refinement module for the rollout trajectory prediction methods. *CoRR*, abs/2102.10859, 2021.

[139] J. Yuan, H. Wang, L. Xiao, W. Zheng, J. Li, F. Lin, and B. Zhang. A formal study of shot boundary detection. *IEEE Trans. Cir. and Sys. for Video Technol.*, 17(2):168–186, Feb. 2007.

[140] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 377–392, Berkeley, CA, USA, 2017. USENIX Association.

[141] T. Zhang, A. Chowdhery, P. Bahl, K. Jamieson, and S. Banerjee. The design and implementation of a wireless video surveillance system. pages 426–438, 09 2015.

[142] D. Zhou, L. Wang, X. Cai, and Y. Liu. Detection of moving targets with a moving camera. In *2009 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 677–681, 2009.

[143] X. Zhou, X. Zhou, L. Chen, and A. Bouguettaya. Efficient subsequence matching over large video databases. *The VLDB Journal*, 21(4):489–508, Aug. 2012.

[144] C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[145] X. Zhu, J. Dai, L. Yuan, and Y. Wei. Towards high performance video object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7210–7218, 2018.

[146] X. Zhu, Y. Wang, J. Dai, L. Yuan, and Y. Wei. Flow-guided feature aggregation for video object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 408–417, 2017.

[147] X. Zhu, Y. Xiong, J. Dai, L. Yuan, and Y. Wei. Deep feature flow for video recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2349–2358, 2017.

[148] Y. Zhu, A. Samajdar, M. Mattina, and P. Whatmough. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 547–560. IEEE Press, 2018.

[149] Z. Zou, Z. Shi, Y. Guo, and J. Ye. Object detection in 20 years: A survey, 2019.

Tambur: Efficient loss recovery for videoconferencing via streaming codes

Michael Rudow[†], Francis Y. Yan[¶], Abhishek Kumar[†], Ganesh Ananthanarayanan[§], Martin Ellis[§], K.V. Rashmi[†]
[†]Carnegie Mellon University, [¶]Microsoft Research, [§]Microsoft

Abstract

Packet loss degrades the quality of experience (QoE) of videoconferencing. The standard approach to recovering lost packets for long-distance communication where retransmission takes too long is forward error correction (FEC). Conventional approaches for FEC for real-time applications are inefficient at protecting against bursts of losses. Yet such bursts frequently arise in practice and can be better tamed with a new class of theoretical FEC schemes, called "streaming codes," that require significantly less redundancy to recover bursts. However, existing streaming codes do not address the needs of videoconferencing, and their potential to improve the QoE for videoconferencing is largely untested. *Tambur* is a new streaming-codes-based approach to videoconferencing that overcomes the aforementioned limitations. We first evaluate Tambur in simulation over a large corpus of traces from Microsoft Teams. Tambur reduces the frequency of decoding failures for video frames by 26% and the bandwidth used for redundancy by 35% compared to the baseline. We implement Tambur in C++, integrate it with a videoconferencing application, and evaluate end-to-end QoE metrics over an emulated network showcasing substantial benefits for several key metrics. For example, Tambur reduces the frequency and cumulative duration of freezes by 26% and 29%, respectively.

1 Introduction

The quality of videoconferencing calls dictates the effectiveness of remote meetings [17] which are now ubiquitous. Videoconferencing calls can be one-on-one [27] or multi-party [39]. Our work focuses on one-on-one calls. Video quality depends on several key performance indicators, such as freeze, bandwidth, packet loss, and latency [14, 28, 41].

Recovering lost packets is crucial for providing high-quality videoconferencing [33, 48]. Losing even a single packet may prevent rendering a video frame. It may also prohibit rendering multiple future frames (i.e., causing the video to freeze) due to inter-frame dependencies of compressed video. Due to this, it is common for videoconferencing applications to handle packet losses at the application level. The two broad viable solutions are retransmissions and forward error correction (FEC). Both approaches transmit redundant data. Consequently, there is a trade-off between bandwidth allocated for redundancy and transmitting original data. Furthermore, videoconferencing applications must recover lost packets within a strict latency—preferably less than 150 ms [33]—to meet the real-time playback requirement.

Retransmission involves minimal redundant data since it resends only the lost packets. Hence, it is preferred whenever possible [64]. However, retransmission is suitable only for scenarios with short round trip times due to the strict real-time latency requirement of videoconferencing applications. For all other cases, videoconferencing applications rely on FEC to recover lost packets within an acceptable latency.

Block codes are the most common form of FEC employed in production systems today. Under a block code, k "data packets" are used to create r redundant packets—called "parity packets." When some of these $(k+r)$ packets are lost, the k data packets can still be recovered. There are r extra parity packets, so the bandwidth overhead is $(r/k) \times 100\%$. One main objective in designing FEC schemes is to minimize the bandwidth overhead. Common examples of block codes include Reed-Solomon (RS) block codes [55] and fountain (i.e., rateless) codes [40]. Many of the codes, e.g., RS codes, are optimal for *random losses* in which packets are lost independently. For instance, in the above example, if RS codes are used, any k packets suffice for recovery. Hence, block codes are popular for production videoconferencing applications. For example, Microsoft Teams uses RS codes.

Videoconferencing applications send data from compressed video frames over multiple packets. We refer to losing several packets over one or more consecutive frames as a "burst" loss. Burst losses can occur for various reasons, including persistent Wi-Fi interference and network congestion (when applications overflow router buffers and cause correlated losses [29]). Our analysis of packet traces from thousands of video calls from Teams (§3.3) shows that real-world losses faced by videoconferencing applications are indeed bursty.

Block codes are highly inefficient in their bandwidth consumption when recovering from burst losses under real-time latency requirements. In contrast, a relatively new theoretical FEC framework, known as "streaming codes" [5, 42, 43], handle burst losses along with strict latency constraints efficiently. At a high level, streaming codes recover packets lost in a burst sequentially by their respective playback deadlines, whereas block codes recover all the lost packets simultaneously by the earliest playback deadline. Using block codes for loss recovery wastes later parity packets sent before the deadline of the final lost packet. Most prior work on streaming codes is theoretical [5, 20, 22, 26, 35, 36, 42, 43, 59–61], studying bounds and code constructions. A few existing works [6, 25] explore the practical applicability of streaming codes but only

for VoIP (i.e., audio, but not video).

Given the dual importance of bandwidth and loss recovery, streaming codes are appealing to videoconferencing applications. However, there are two main challenges. First, there are gaps between existing streaming codes and videoconferencing applications. Most practical variants of streaming codes [6, 25] are limited to settings in which the sizes of the input data are a fixed constant over time. In contrast, in videoconferencing, the sizes of compressed video frames are variable. The only streaming codes that accommodate such variability [59–61] pessimistically assume that a frame is entirely lost or received because the framework involves sending each frame in a single packet. This is seldom true in videoconferencing applications. Existing streaming codes also require that every burst is followed by a *guard space* where all packets are received. But this assumption often does not hold in practice (as we show in §3.2). Also, existing streaming codes set the amount of redundancy with a parameter of a theoretical channel model that is unknown in practice. Second, streaming codes' effectiveness for improving the QoE for real-world videoconferencing applications is untested on real-world data.

This work addresses the aforementioned challenges. We present Tambur, a new communication scheme for bandwidth-efficient loss recovery for videoconferencing.[1] Tambur comprises two components. First, a new streaming code that builds upon a prior theoretical framework [61] while overcoming its limitations with respect to real-world videoconferencing applications. Specifically, Tambur allows for specifying a bandwidth overhead for each frame. Furthermore, for any given bandwidth overhead, Tambur creates data packets and parity packets in a manner that (a) is not overly pessimistic by facilitating recovery from bursts where only some packets are lost per frame and (b) is robust to losses in the guard space. Second, the streaming code is integrated with a machine learning (ML) model to take a predictive decision on the bandwidth allocated to streaming codes. Specifically, a lightweight approach is employed, which uses only a simple model and a single bit of additional feedback.

We analyze packet traces collected from thousands of video calls from Teams and present three key observations in §3: (a) Bursts of packet losses frequently arise. (b) Losses are often, but not always, followed by a guard space of several frames with no losses. (c) Codes employed in production (RS codes) use a significant bandwidth overhead to recover lost packets in real time, depleting the bandwidth for the original data.

We first evaluate Tambur in simulation over a large corpus of traces from Teams (§5.2). We compare Tambur with Teams's FEC ("Block-Within," a block code within a frame; §5.1) and show that Tambur recovers 26.5% more frames with 35.1% less bandwidth overhead.

We also implement and integrate Tambur, several baselines (Block-Within and "Block-Multi," a block code across multiple frames) and several variants of Tambur ("Tambur-full-BW," which matches the bandwidth overhead of Block-Within and "Tambur-0.9," which reduces the bandwidth overhead more at the cost of recovering fewer frames) with a videoconferencing benchmark platform that we developed. We then evaluate the schemes over an emulated network to assess the impact on the QoE (§5.4). Tambur, Tambur-full-BW, and Tambur-0.9 reduce the average frequency of video freezes by 26%, 29%, and 17%, respectively, compared with the better of Block-Within and Block-Multi. Fig. 1 shows that these benefits hold across many percentiles. These benefits highlight that Tambur improves the QoE, as it has been shown [44, 53, 66] that video freezes have a detrimental effect on user engagement.

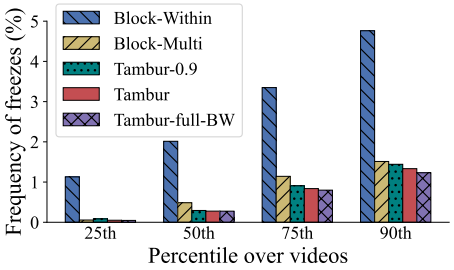
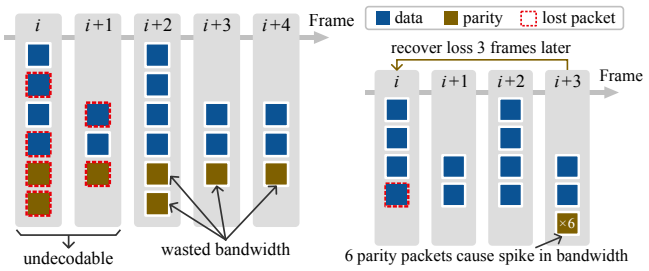


Figure 1: Tambur reduces the ratio of frozen frames to total frames per-video by 78% and 26% compared to Block-Within and Block-Multi, respectively, at a lower bandwidth overhead.

In summary, our main contributions are to:

- Analyze thousands of packet loss logs for video calls taken from a large commercial videoconferencing application, and characterize their suitability for using streaming codes. To the best of our knowledge, this is the first work to evaluate the potential of streaming codes using large-scale, real-world traces.
- Present *Tambur*, which bridges the gap between the theory behind streaming codes and videoconferencing applications by (a) designing a new streaming code that is well-suited to videoconferencing and (b) integrating it with a lightweight ML model to take a predictive decision on the bandwidth allocated to streaming codes.
- Implement a new benchmark platform to enable research on videoconferencing with an easy-to-use interface to integrate and assess new FEC schemes. In addition, implement Tambur, Block-Within, and Block-Multi in C++ and incorporate them into the benchmark platform using the interface.
- Evaluate Tambur over a large corpus of production traces through simulation, and show that it simultaneously reduces the frequency of non-recoverable frames and bandwidth overhead by 26.5% and 35.1%, respectively.
- Evaluate Tambur over emulated networks and show significant improvements over key metrics pertaining to end-to-end QoE (e.g., reducing the frequency of freezes

[1] Named to convey *Tam*ing *bur*st losses.



(a) Within-frame (b) Multi-frame

Figure 2: Two approaches for employing block codes: (a) within each frame and (b) across multiple frames.

by 26% and the cumulative duration of freezes by 29%).

Overall, to the best of our knowledge, this work is the first to establish that streaming codes can improve key metrics relating to the QoE for videoconferencing. This work also showcases the potential of a new form of FEC, streaming codes along with learning-based bandwidth allocation, for bandwidth-efficient loss recovery in videoconferencing. This work poses no ethical issues.

2 Background and motivation

2.1 Conventional FEC and its challenges in videoconferencing

Block codes. One of the most commonly used FECs is the so-called "block codes." The idea of block codes is to encode k data packets, $\langle D[1],\ldots,D[k]\rangle$ to r parity packets into $\langle D[1],\ldots,D[k],P[1],\ldots,P[r]\rangle$, so that the k data packets can be recovered using a subset of the $(k+r)$ packets. When any k of the $(k+r)$ packets suffice for recovery, the block code is termed "maximally distance separable (MDS)." One of the best known examples of MDS codes is the Reed-Solomon (RS) block codes [55]. Other examples of block codes include fountain (i.e., rateless) codes [40], or two-dimensional block codes [67].

Traditionally, FEC applies to packets, but videoconferencing involves transmitting multiple packets for each video frame. One natural solution is to apply a block code to the data packets *within each frame* (Fig. 2a). The parity packets are sent immediately after the final data packet of a frame. A second approach is to apply a block code across the data packets of *multiple frames* (Fig. 2b) by sending all parity packets after the final data packet of the last frame in the block. Our analysis of the production packet loss traces (§3) from Teams shows that the packet losses are bursty. Both approaches have significant limitations for burst losses.

Limitations of block codes for videoconferencing. When packet losses occur as bursts, the within-frame approach wastes the redundancy sent in frames immediately following a burst because it is useless for recovering the lost frames. Although the multi-frame approach overcomes this problem, it has two main drawbacks. First, the latency of recovering losses is high due to waiting for the parity packets, which are sent after the final frame in the block, to recover any packets. The length of the block code must be short lest the latency exceeds the real time deadline to play a frame, leading to an increased bandwidth overhead and reduced robustness to burst losses. Second, packets sent in rapid succession may be lost if a router buffer is full. When a full router buffer coincides with the final frame of a block, no lost packets are recovered.

The bandwidth consumed by parity packets of FEC can be substantially higher than retransmission, even for modest packet loss rates. Unlike retransmission, which only resends lost packets, even an "optimal" FEC scheme does not know which packets will be lost. Hence, it must send far more parity packets than lost packets. For example, to prevent a video freeze, at least one parity packet must be sent every $\approx$ 150ms to cover the scenario of losing a data packet. However, this parity packet is not used if there are no losses.

2.2 Streaming codes

A class of codes, known as "streaming codes" [5, 42, 43, 59–61], specifically addresses burst losses and sequential communication between a sender and a receiver. At a high level, streaming codes avoid the limitations of within-frame and multi-frame by (a) sending parity packets with each frame and (b) using all parity packets received by the playback deadline of the *final* frame of a burst for recovering losses. We describe the theoretical framework of streaming codes in detail, provide an illustrative example, and then discuss how it is a promising option for videoconferencing applications impeded by a large gap between the framework and practice.

The streaming codes framework consists of the following. (1) A sender that generates data packets sequentially at regular intervals and transmits packets sequentially to a receiver. (2) An adversarial packet loss channel that introduces burst losses of length b followed by guard spaces of packet receptions. (3) A requirement that the receiver recovers lost data packets within a specified time. The data packet that arrives at the time index i must be recovered by time index $(i+\tau)$. We call the parameter τ the "latency deadline." After a burst, the guard space must be at least τ packets (but longer guard spaces are not needed since bursts are to be recovered within τ packets).

Sequential encoding. The sequential nature of encoding in streaming codes is well suited for videoconferencing, wherein a sequence of compressed frames are to be transmitted periodically (e.g., one every 33.3 ms for a video showing 30 frames per second). We will denote the symbols sent for the ith video frame as $D[i]$, where each symbol can be thought of as a vector of bits.[2] These symbols are distributed over

[2]More formally, a symbol is an element of a mathematical entity called a finite field, and all operations are performed over finite fields using modular arithmetic. For simplicity, readers can just assume the usual arithmetic.

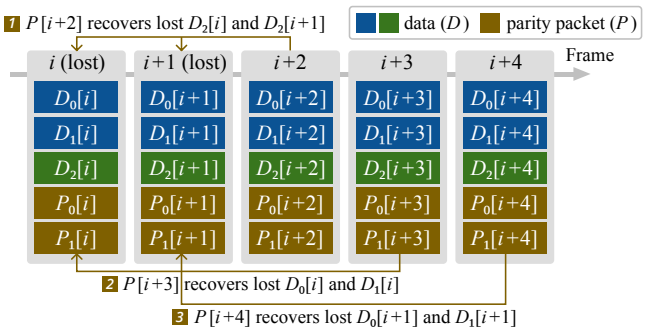


Figure 3: Recovering $b = 2$ lost frames starting in frame i within a latency deadline of $\tau = 3$ using a streaming code. For each frame in the burst, all parity symbols sent τ packets later recover its lost symbols.

one or more packets to be sent to the receiver. In addition, some parity symbols, denoted as $P[i]$, are transmitted in one or more packets. These parity symbols are a function (linear combinations) of the data symbols of the past few frames.

Sequential recovery. Under the streaming codes model, the latency deadline parameter τ determines the delay in recovering a lost packet: if $D[j]$ is lost, it must be recovered using symbols from parity packets until $P[j+\tau]$. Each video frame must be recovered within a strict latency to be rendered in real time. The latency deadline parameter τ is set according to the frame rate and one-way delay to induce a suitable maximum latency to recover lost frames. For example, if the maximum tolerable latency is 150 ms (a standard value for real-time video communication [33]), the one-way propagation delay is 50 ms, and a frame is encoded every 33.3 ms (i.e., at 30 fps), τ could be set as 3 $(= (150-50)/33.3)$.

An example of sequential loss recovery of a burst of length 2 starting in frame i within a latency deadline of $\tau = 3$ using existing streaming codes (e.g., [5, 42, 43]) is shown in Fig. 3. Each frame comprises the same amount of data. First, the parity symbols of the packet sent immediately after the burst recovers one-third of the missing data symbols of each lost frame (i.e., $D_2[i]$ and $D_2[i+1]$).Second, the remaining lost data symbols of frames i and $(i+1)$ (i.e., $(D_0[i], D_1[i])$ and $(D_0[i+1], D_1[i+1])$, respectively) are recovered with the parity symbols sent in frames $(i+3)$ and $(i+4)$, respectively.

Streaming codes recover a burst loss by *sequentially recovering* each frame in the burst within its deadline. For a burst loss that encompasses b consecutive frames $\{i, \ldots, i+b-1\}$, a data packet $D[j]$ in the burst is recovered using the parity symbols of $P[i+b], \ldots, P[j+\tau]$. This sequential nature of the recovery of streaming codes allows them to use *all* parity symbols that are received within the deadline. For example, $P[j+\tau]$ is used to recover $D[j]$ *after* the latency deadline of $D[i]$ for $i < j$. In contrast, block codes recover all lost packets together. Hence, the recovery occurs by the first lost frame's deadline (i.e., by the time the symbols of $P[i+\tau]$ are received), wasting the parity symbols sent subsequently. This key difference enables streaming codes to attain significantly lower bandwidth overhead; the longer the burst, the greater the benefits of streaming codes. However, it also requires a guard space of at least τ frames after the burst lest some frame not be recovered with the latency deadline.

2.3 Challenges of using streaming codes for videoconferencing

There are two main challenges in using streaming codes for videoconferencing. First, significant gaps between the theoretical models and practical systems render existing streaming codes incompatible with videoconferencing applications. Second, streaming codes's effectiveness for videoconferencing is untested on large-scale real-world traces. Hence, the potential of streaming codes improving QoE of videoconferencing applications is yet unknown. These challenges are discussed in more detail below.

Gaps between the existing model and videoconferencing applications. The existing practical work on streaming codes [6, 25], like the theoretical work they build upon [5, 42], is limited to settings where the amount of data to be transmitted at each time instant is a fixed constant. However, videoconferencing involves sending compressed video frames whose sizes vary. Only a few streaming code constructions [59–61] can handle this variability. However, as discussed in §2.2, existing streaming codes, including those in [59–61], consider an *adversarial* loss model that imposes bursts of length b. When applied for videoconferencing, the parameter b translates into the number of consecutive frames for which *all* packets are lost. However, videoconferencing applications frequently send multiple packets per frame, and often only some of these packets are lost, as we show in greater detail in §3 for packet loss traces from production. Existing streaming codes are overly pessimistic because they can recover from losing all packets for multiple consecutive frames. This requirement imposes a significant bandwidth penalty, negating the potential bandwidth savings of streaming codes. Streaming codes are also vulnerable to recovery failures if there are *any* losses in the guard space after a burst. But, in practice, many bursts are not followed by such guard spaces (see §3.2).

Applicability of streaming codes in the wild. The benefits of streaming codes for VoIP applications have been studied using simulated losses under theoretical loss models, such as the Gilbert-Elliott channel [23] and over traces [6,25], wherein each frame is sent in one packet and all frames/packets are of a fixed constant size. However, these results do not apply to videoconferencing applications, which send (a) multiple packets for each frame and (b) varying amounts of data per frame. Streaming codes perform best when each burst occurs across multiple frames and is followed by a guard space of several frames without losses. A natural question is whether

such losses arise in videoconferencing and if they can be exploited via streaming codes. To the best of our knowledge, no study of large-scale real-world packet losses establishes the applicability of streaming codes in the wild. Furthermore, establishing that streaming codes are viable to improve the QoE hinges on improving several metrics relating to the QoE. Yet an analysis of streaming codes' impact on such metrics is similarly lacking in the existing literature. Finally, the effect of inter-frame dependencies on the benefits of streaming codes has yet to be assessed, even though inter-frame dependencies are prevalent in videoconferencing.

3 Packet loss in the wild

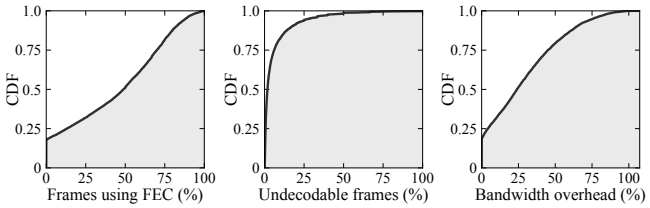
Logs (specifically, packet loss traces) from Microsoft Teams were collected from a random sample of calls over two weeks. One week's traces were held out as a test set for the evaluation.

Teams uses FEC only after a packet loss occurs, which is fairly standard in the industry [64] to avoid wasting bandwidth for the many calls that do not experience any loss. We limit our study to traces with at least two instances of loss since our focus is on improving scenarios *after* FEC is activated (i.e., FEC is turned on after the first loss and then used to recover the second). Our analysis involves approximately 9700 traces, which constitute 16% of all the traces. Studying these traces sheds light on the tail performance, which is crucial for real-world commercial applications. Each trace corresponds to one call and contains the size, sequence number, and send/receive timestamps for each received packet, as well as whether it is a parity packet or data packet; lost packets are identified via missing sequence numbers. Due to the application's data collection method, the traces are limited to the final one minute of the call. Although the logs are for packets, we approximate frame-level information by combining the logs with Teams's packetization logic and have corroborated with the Teams engineers that this approximation is good.

3.1 FEC metrics

Teams employs an RS block code within each frame and varies the bandwidth overhead based on infrequent feedback from the receiver on packet losses. We will denote the FEC scheme used by the application simply as "Block-Within."

We evaluate three metrics over the traces. First, the percent of video frames using FEC for each videoconferencing call (Fig. 4a). The 25th, 50th, and 75th percentile for the percent of video frames over each trace using FEC are 13%, 48.8%, and 70% of calls respectively, indicating that FEC is applied to a significant portion of the frames. Second, the percent of decoding failures for video frames over all frames for each videoconferencing call (Fig. 4b). The 25th, 50th, and 75th percentile for the percent of decoding failures of frames are 0.6%, 1.8%, and 6.1% of calls. Note that the decoding failures should be kept below around 1% to provide high QoE [33].



(a) FEC usage (b) Decoding failures (c) Redundancy

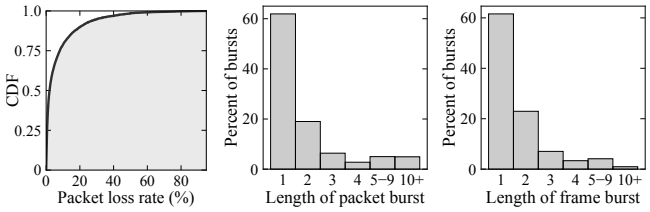
Figure 4: CDFs over the traces from Teams of (a) how often FEC is used to encode frames to protect against packet loss, (b) how often the lost packets are not decoded, and (c) the bandwidth overhead of parity packets.

As such, decoding failures are prevalent enough to tangibly negatively impact the QoE, prompting the need for a more effective FEC mechanism. Third, the bandwidth overhead for each call (Fig. 4c). The 25th, 50th, and 75th percentile for the bandwidth overhead are 4.2%, 24%, and 45% of calls. Thus, reducing the bandwidth overhead will free a significant portion of the bandwidth for these calls.

3.2 Network quality

We analyze the packet losses to assess streaming codes' suitability for real-world videoconferencing applications. To the best of our knowledge, this is the first work to analyze large-scale real-world packet loss traces from this perspective. We analyze three key metrics of losses in Fig. 5. (1) The packet loss rate for each trace (Fig. 5a). (2) The distribution of lengths of bursts of packets measured over all of the calls (Fig. 5b). (3) The distribution of the lengths of bursts of frames (i.e., the number of consecutive frames with at least one packet lost) measured over all of the calls (Fig. 5c). This metric indicates streaming codes' suitability, as they are most effective when bursts of lost packets encompass multiple frames (see § 2.2).

The mean percent of packets lost over the traces is 7%. It is higher than the packet loss in the FCC report [15] since we focus on the traces where FEC is employed. If the other traces from Teams are also considered, the mean packet loss over all traces is 1.7%, which is comparable to the FCC measurement. In earlier studies of end-to-end Internet packet loss, loss rates tended to vary over time and between ISPs and access network technology [8, 18, 24, 54], with ISP queue management policies impacting the loss patterns seen by applications. As discussed in [24], in home broadband networks, loss rates are often less than 1% for long periods, with infrequent periods of very bursty packet loss. Similar patterns are seen in mobile networks, where loss rates tend to increase during handovers [8], and much longer packet loss bursts are seen. Our traces from Teams, described in this section, show similar behavior, with a large number showing very low loss rates, with a long-tail of traces showing extremely bursty packet loss. Specifically, 38.1% of the instances of packet loss in-



(a) Packet loss (b) Packet bursts (c) Frame bursts

Figure 5: Packet loss is prominent (e.g., Fig 5a shows $1-10\%$ packet loss for most traces) and often occurs as bursts across consecutive packets (Fig 5b) or frames (Fig 5c).

volve at least two consecutive packets being lost (Fig. 5b), and 38.4% of instances of packet loss encompass more than one video frame. Such loss patterns can be efficiently recovered by streaming codes (§3.3).

There is also a trade-off between the bandwidth overhead (i.e., the bandwidth used for parity packets) and the probability of decoding failure. The bandwidth overhead cannot be prohibitively high lest there be insufficient bandwidth for the original data. Consequently, the frequency of decoding failures for frames is non-negligible despite using FEC.

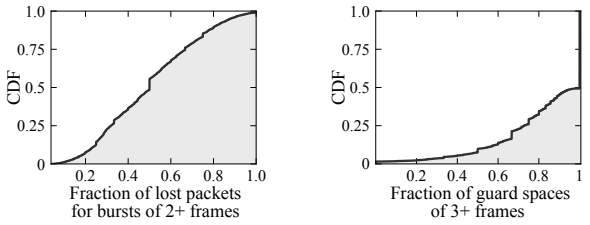
3.3 Potential of streaming codes

Recall from §2.3 that streaming codes are most effective when (a) packet loss occurs as a burst across multiple consecutive frames and (b) the burst loss is followed by a guard space of multiple consecutive frames with no losses. We formalize two metrics to capture these conditions. We then show that the packet losses in the traces exhibit these features.

Measuring bursts. The bandwidth overhead needed to decode a burst depends on the fraction of the packets being lost when losses occur across multiple frames. We introduce a new metric to formalize this notion.

Definition 1 (Multi-frame burstiness) *Suppose a burst occurs across two or more frames, i through j, over which s packets are sent. If l of the s packets are lost, the* ***multi-frame burstiness*** *is defined as l/s.*

For example, suppose Tambur sends packets $(D_0[i], D_1[i], D_2[i])$ and $(D_0[i+1], D_1[i+1])$ over frames i and $(i+1)$, respectively. Suppose $D_1[i], D_2[i]$, and $D_0[i+1]$ are lost. Then the multi-frame burstiness is $3/5$. The multi-frame burstiness is always positive since at least one packet is lost for each frame in the burst. The maximum value of 1 occurs when all packets are lost for all frames in the burst. High values correspond to situations of a high percentage of the packets being lost for multiple consecutive frames. The value of the multi-frame burstiness directly relates to the minimum bandwidth overhead needed for any code to decode lossy frames in real time.



(a) Multi-frame burstiness (b) Guard space sufficiency

Figure 6: The CDFs over the traces of the (a) the multi-frame burstiness (for traces with at least one burst over 2+ frames), and (b) the guard space sufficiency.

Measuring guard spaces. Streaming codes can reduce bandwidth overhead for scenarios where a burst of packet losses is followed by a guard space of at least τ frames that experience reliable transmission, where τ is the latency deadline parameter (§2). We now introduce a new metric to measure the extent to which the guard spaces exhibit this property.

Definition 2 (Guard space sufficiency) *The* ***τ-guard space sufficiency*** *is the fraction of instances in which one or more frames with packet loss are followed by at least τ consecutive frames which experience lossless transmission.*

The value of the guard space sufficiency varies from 0 to 1. It is *negatively* related to the bandwidth overhead needed when using streaming codes. High values for the guard space sufficiency indicate that the bandwidth overhead can be reduced.

Suitability of streaming codes. The multi-frame burstiness and 3-guard space sufficiency is evaluated over the traces in Fig. 6.[3] In Fig. 6a, the value of the multi-frame burstiness is shown to vary over the range 0 to 1, with values at the 25th, 50th, and 75th percentiles of 0.32, 0.5, and 0.67 respectively. This indicates that the bandwidth overhead needed when using streaming codes varies over the traces, as expected. For higher values, more bandwidth must be allocated to redundancy to decode the losses. For lower values, it is possible to make do with less bandwidth used for redundancy. The guard space sufficiency is evaluated over the traces in Fig. 6b, and its values at the 25th, 50th, and 75th percentiles are 0.73, 1.0, and 1.0 respectively. These values imply that *streaming codes are often suitable*. For example, for the traces with a value of 1.0, every single time a burst occurs across multiple frames, streaming codes could have been used to decode the losses with the optimal amount of bandwidth overhead. Yet, the low values indicate insufficient guard spaces for using existing streaming codes to reduce the bandwidth overhead, as they are vulnerable to losses in the guard space.

[3] Recall from § 2.2 that $\tau = 3$ applies for a realistic choice of parameters, in which case a guard space of length 3 is beneficial for loss recovery with streaming codes.

3.4 Key findings

Bursts of packet losses followed by guard spaces arise frequently and are conducive to streaming codes. However, this is not always the case. Bursts are sometimes followed by short guard spaces or involve significant packet loss, in which case the bandwidth overhead cannot be reduced via streaming codes. Hence, integrating streaming codes into real-world applications *requires (a) predicting whether the bandwidth overhead can be reduced without incurring decoding failures, (b) leveraging partial losses in a frame (i.e., losses of only some packets per frame rather than all packets) and (c) adding robustness to losses in the guard space.*

4 Tambur

We present Tambur, which exploits the potential discussed in §3.3 and addresses the challenges discussed in §2.3 by (1) using an ML model to take predictive decisions on the bandwidth overhead, and (2) designing a new streaming code suitable for videoconferencing given any setting for the bandwidth overhead.[4] First, an ML model makes a predictive decision on the number of parity symbols to allocate for each frame. This helps to set the bandwidth overhead to match the network conditions. Second, the parity symbols are defined to provide (a) sequential recovery of bursts over multiple frames while exploiting partial losses, (b) recovery of occasional losses within a single frame immediately, and (c) robustness to a small amount of loss in the guard space after a burst. Third, a new methodology is employed to distribute each frame's data and parity symbols over multiple packets. The design of the parity symbols and their distribution across packets constitute Tambur's streaming code. During loss recovery, Tambur uses the received packets from the frames involved in a burst (i.e., partial losses), which allows for a lower bandwidth overhead than is possible for existing streaming codes that ignore such packets.

Fig. 7 shows how Tambur fits into the stack of a videoconferencing application. The *streaming encoder* encodes data from compressed frames into data packets and parity packets. A *Bandwidth Overhead Predictor* periodically selects bandwidth overhead for each frame using a predictive (ML) model based on the losses observed at the decoder and sends the value to the encoder. The *streaming decoder* uses parity packets to recover lost data packets. We will now describe these components in detail.

4.1 Tambur's streaming code

We present the code in two parts: encoding and decoding.

Encoding. We illustrate how Tambur encodes the ith frame. Fig. 8 shows an example of encoding for $\tau = 3$. The data

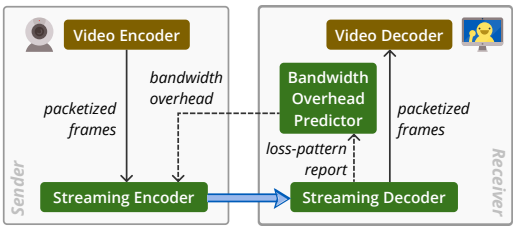


Figure 7: Overview of Tambur. The components in green are specific to Tambur.

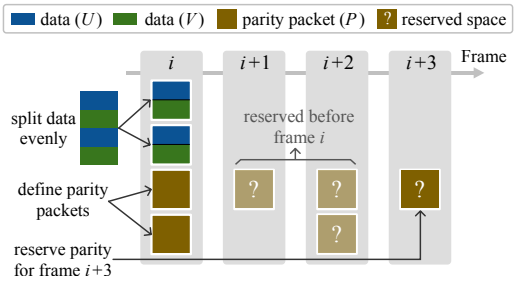


Figure 8: Encoding for $\tau = 3$. Tambur splits frame i evenly into $(V[i], U[i])$ and sends them over data packets. Also, Tambur sends parity packets for recovering $V[i-3], \ldots, V[i], U[i]$ and $U[i-3]$ and reserves space for parity symbols of frame $(i+3)$.

symbols of this frame, $D[i]$, are sent in data packets, and the parity symbols, $P[i]$, are sent in parity packets. The sizes of the packets are maximized subject to (a) not exceeding an MTU (for example, 1500 bytes in our experiments) to be equal. The previous value of the Bandwidth Overhead Predictor determines how many parity symbols are allocated for frame i. These parity symbols will be sent τ frames later (see "reserved space" in Fig. 8). The number of parity symbols sent for frame i was determined by the size of frame $(i-\tau)$.

Next, we describe how parity symbols are formed. The symbols of $P[i]$ are linear combinations of the symbols of the $(\tau+1)$ frames, $\{D[i], \ldots, D[i-\tau]\}$. To define the parity symbols, it helps to view the data symbols of the associated $(\tau+1)$ frames as being divided evenly into two parts as $D[j] = (V[j], U[j])$, for $j \in \{i, \ldots, i-\tau\}$. Fig. 8 shows these components in blue and green, respectively.

The symbols of $P[i]$ are carefully designed linear combinations of the symbols of $V[i], \ldots, V[i-\tau], U[i]$, and $U[i-\tau]$. Specifically, $P[i]$ is sum of three quantities: $P[i] := P_1[i] + P_2[i] + P_3[i]$. The symbols of $P_1[i]$ are linear combinations of the symbols of $V[i-\tau], \ldots, V[i-1]$. The symbols of $P_2[i]$ are linear combinations of the symbols of $U[i-\tau]$. The symbols of $P_3[i]$ are linear combinations of the symbols of $U[i]$ and $V[i]$. All linear combinations are carefully chosen to be *linearly independent* linear equations.[5]

Decoding. We describe the decoding process in two parts:

[4] The new streaming code builds upon recently developed theoretical streaming codes [60, 61] while overcoming the limitations discussed in §2.3.

[5] It suffices to take linear equations from three different Cauchy matrices.

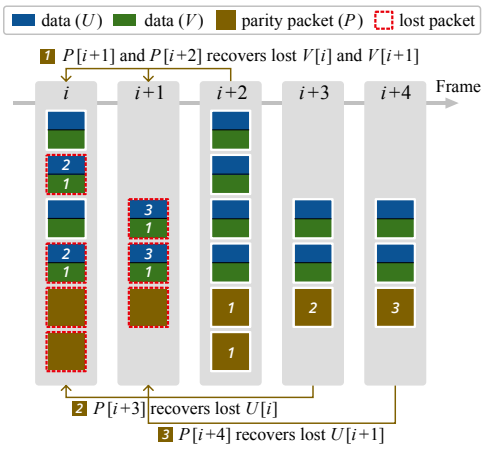


Figure 9: Decoding a burst across 2 frames within $\tau = 3$ frames delay using Tambur's streaming code. Data symbols labeled 1, 2, and 3 are decoded using the parity packets with the same label.

(1) occasional packet losses and (2) burst of packet losses. Let all frames before the loss be decoded. First, suppose that packet loss is rare, and the size of $P[i]$ exceeds the number of symbols lost for frame i. Then $P[i]$ suffices to decode the ith frame (specifically, by solving a system of linear equations).

Second, consider a burst of packet losses across two consecutive frames for $\tau = 3$, as is shown in Fig. 9. Packet losses (red-dashed border) span frames i and $(i+1)$. For each frame i, the blue, green, and brown parts represent $U[i]$, $V[i]$, and $P[i]$, respectively. First, $V[i]$ and $V[i+1]$ are both decoded using $P[i+2]$, which consists of independent linear combinations of (a) the symbols of $V[i]$ and $V[i+1]$, and (b) the (received) symbols of $V[i-1]$, $U[i-1]$, $U[i+2]$, and $V[i+2]$. Second, for $j \in \{i, i+1\}$, $U[j]$ is decoded using $P[j+3]$, which consists of independent linear combinations of (a) the symbols of $U[j]$, and (b) the (available) symbols of $V[j-3], \ldots, V[j]$, and $U[j]$. The key to this methodology is that $U[i+1]$ is *not* recovered by the latency deadline of $D[i]$ (i.e., $(i+3)$). This enables using extra parity symbols (i.e., $P[i+4]$) to recover $U[i+1]$ while still decoding each data packet within $\tau = 3$ frames. Appendix A presents the general case. If decoding fails, the receiver queries the sender to generate a new keyframe (i.e., a self-sufficient frame) to handle inter-frame dependencies.

There are three key differences from existing streaming codes for videoconferencing: (1) The data symbols and parity symbols of a frame are sent over multiple packets instead of a single packet. (2) Each frame's parity packets are designed such that they are useful in recovering its lost data packets (in addition to being useful in recovering previously sent frames). (3) The code is flexible enough to allow per-frame bandwidth overhead to be set using the Bandwidth Overhead Predictor.

4.2 Bandwidth overhead predictor

At a high level, Tambur makes use of a predictive model to determine the bandwidth overhead employed by its streaming code (i.e., the amount of "reserved space" in Fig. 8). This predictive model takes as input a feature vector computed by the receiver periodically (e.g., every two seconds), dubbed a *loss-pattern report*. The predictive model's output is then sent to the sender to set the bandwidth overhead for each frame for Tambur's streaming code until the next loss-pattern report is received. For example, a bandwidth overhead of 50% means that if frame i comprises 1000 bytes, 500 bytes of parity symbols are allocated.

Loss-pattern report. Let P be the bitmap of packet losses since the last loss-pattern report, where 1 denotes a loss and 0 is a reception. Let F be a bitmap over all frames since the last loss-pattern report of whether at least one of the frame's packets was lost. The loss-pattern report consists of the following 13 quantities, all of which can be computed in linear time with a single sequential pass over F and P.

- Multi-frame burstiness and guard space sufficiency (§3).
- Fraction of losses for P and F.
- Mean number of consecutive losses for P and F.
- Mean length of guard spaces for P and F.
- Burst density [12] and gap density [12] for P and F[6].
- A score employed by Teams to choose its bandwidth overhead, which is based on the observed fraction of packet losses and lengths of bursts.

Bandwidth overhead prediction via weighted classification. Tambur uses an ML model to determine the bandwidth overhead allocated per frame based on the recent loss conditions. As discussed above in § 4.1, Tambur's streaming code enables such an approach by allowing fine-grained tuning of the bandwidth overhead. To keep the model simple, we select two options for the bandwidth overhead. This approach easily generalizes to more than two values for bandwidth overhead by using a multiclass classifier to enable tuning the bandwidth overhead used by Tambur. In our implementation, we use a small neural network (discussed further in §4.3), although any methodology could be substituted.[7]

The ML model is trained with different *weights* for the two classes based on prioritization of bandwidth savings versus minimizing decoding failures. Essentially, the higher the weight for the class corresponding to the higher bandwidth overhead, the greater the frequency of decoding frames, but the lower the reduction in bandwidth overhead. Videoconferencing service operators can use these weights as a knob to prioritize reducing decoding failures or bandwidth overhead.

Neural network details. Binary classification is conducted using a small fully connected neural network with one hidden layer. The input is the values of the 13 metrics for the

[6] The parameter gMin [12] is set to be 1 and τ for P and F respectively.
[7] We found ML models to outperform heuristics empirically.

previous 3 loss-pattern reports. The cross-entropy loss is applied, and by default the weights for mistakenly reducing the bandwidth overhead (i.e., causing a decoding failure) and not reducing the bandwidth overhead by half (i.e., failing to save bandwidth) are 0.999 and 0.001, respectively. We tested various number of hidden neurons (e.g., 100, 1000, and 10000) and selected 1000 as the smallest option to reach the point of diminishing returns. The model is implemented and trained in PyTorch offline using the traces based on the optimal decision for reducing the bandwidth overhead without causing decoding failures. During inference, it is instantiated in C++.

4.3 Implementation

We implemented Tambur in C++ as part of a new independent library called Tambur that any videoconferencing application can use.[8] At the sender, Tambur takes successive compressed frames as input and outputs data packets and parity packets. At the receiver, Tambur decodes lost packets by solving a system of linear equations using the symbols of the received packets. When packets are lost, we combine properties of streaming codes with an open-source min-cut/max-flow algorithm [11] to determine which data symbols can be decoded using which parity symbols in negligible time (see Appendix B). Data is then decoded by solving the smallest full-rank systems of linear equations.

We use a small header to provide frame-level information needed for decoding. This includes sequence numbers for packets and frames and relative positions of a packet within a frame and amongst parity packets. The streaming decoder also needs the size of the lost frame in order to decode it (even when all packets corresponding to the frame are lost); hence, we encode the sequence of frame sizes using a streaming code and send one parity symbol of this code in each packet.

The library provides an interface for rapidly prototyping new FEC schemes. We used this interface to implement the baselines from §1 (i.e., Block-Within and Block-Multi).

The core arithmetic of linear encoding and solving a system of linear equations for decoding is done using Jerasure 2.0 [52], an open-source library in C/C++ with modules for key operations of erasure coding. Jerasure 2.0 is built on top of the GF-Complete library [51], which uses Intel SIMD instructions to perform Galois Field arithmetic quickly. Tambur involves encoding data into "coding blocks" of 256 bytes, each of which uses the same linear equations. Extending Tambur to use hardware offload to encode and decode frames is a potential avenue of future work.

Integration with videoconferencing. To validate Tambur's effectiveness in the real world, we integrate it with *Ringmaster*[9], a newvideoconferencing platform that emulates one-on-one video calls for benchmarking FEC schemes. Ringmaster is implemented in ∼4000 lines of C++. Its video sender reads raw frames from an input Y4M video file on disk at a precise frame rate (e.g., 30 fps) and compresses them with the VP9 encoder in the `libvpx` [1] library using similar codec configuration as in WebRTC [2]. A user-provided FEC scheme provides parity data for the encoded frames, which is sent over UDP after packetization to the video receiver. Upon receiving the frames, the video receiver applies the FEC decoder and VP9 decoder sequentially to decode and render the original video frames. In addition, Ringmaster allows for requesting new keyframes, e.g., when the receiver fails to recover a video frame due to excessive loss of packets and thus requests the sender to encode a new keyframe so as to resume the video. At the end of the automated call, QoE metrics are computed by aggregating logs from both endpoints, which record the timestamps when each frame is encoded or decoded, along with its frame ID, size, FEC bandwidth overhead, etc.

Ringmaster provides clean and modular interfaces that we use to integrate it into Tambur. Combining Ringmaster with Tambur enables benchmarks of FEC schemes'performance featuring QoE metrics, e.g., video freezes, per-frame delay, rendered frame rate, for FEC schemes implemented via Tambur's interface. Furthermore, Ringmaster also allows researchers to isolate the impact of FEC and disable modules that interfere with FEC, such as bandwidth estimation [13] and packet retransmission.

5 Evaluation

To assess whether Tambur can improve the QoE, we ask:

- Can Tambur provide significant benefits for metrics relating to FEC on real-world losses?
- Do the benefits of Tambur lead to a higher QoE?

5.1 Experimental methodology and highlights

Videoconferencing application parameters. In our experiments, we aim for a maximum tolerable latency of 150 ms to meet industry recommendations [33], which is a fairly standard value for interactive video. The frame rate is taken to be 30 fps, which is a typical value in videoconferencing. The inter-frame arrival time for 30 fps is 33.3ms. Allowing for a one-way frame delay of 50 ms leaves room for a decoding delay of around 100ms. Thus, the parameter τ can be at most 3 (frames) for the end-to-end latency (i.e., $33.3\tau+50$) to be at most $\approx$ 150 ms. The two options for the bandwidth overhead of Tambur are to match or use half of the bandwidth overhead of the baseline coding scheme, Block-Within, which is introduced next.

Coding schemes. We evaluate six coding schemes. (1) **Block-Within** (Fig. 2a), which applies RS codes within a frame. This scheme is employed in production by Teams. (2) **Block-Multi** (Fig. 2b) which applies RS codes across $(\tau+1)=4$

[8] https://github.com/Thesys-lab/tambur
[9] https://github.com/microsoft/ringmaster

frames. RS codes are *optimal* block codes, and hence the above two baselines outperform other block codes such as fountain or rateless codes in recovering losses and bandwidth overhead. (3) **Tambur-full-BW**, which is a variant of Tambur that matches Block-Within's bandwidth overhead. (4) **Tambur-0.9**, which is Tambur with the neural network trained to prioritize bandwidth savings more by decreasing the weight of misclassification from 0.999 to 0.9 in the loss function. Thus, Tambur-0.9 prioritizes reducing the bandwidth overhead more than Tambur. (5) **Tambur-low-BW**, which is a variant of Tambur that uses 50% of the bandwidth overhead of Block-Within. (6) **Oracle**, which optimally selects between Tambur-full-BW, Tambur-low-BW, or Block-Within. Each time the sender obtains feedback from the receiver, the Oracle selects the scheme with the smallest bandwidth overhead among the scheme(s) that recover the most frames. This choice never causes a non-recoverable loss. Consequently, the Oracle always recovers at least as many frames as Block-Within, Tambur, and Tambur-full-BW. The bandwidth overhead of Block-Within and Block-Multi is never reduced to ensure a fair comparison of Tambur's loss recovery capabilities and because both baselines already perform worse than Tambur despite using the full bandwidth overhead. Like Tambur, Block-Within and Block-Multi send feedback to the sender once FEC decoding has failed to trigger a new keyframe as a fallback mechanism to handle inter-frame dependencies.

Metrics. We evaluate the following metrics: (1) Percent of non-recoverable frames, which is the percentage of compressed frames that are not recovered. (2) Bandwidth overhead for FEC. (3) Percent of non-rendered frames, which is the percent of frames that are not played by the receiver—this includes non-recovered frames and recovered frames that depend on non-recovered frames. (4) Latency, which is the duration between a frame being created and rendered. (5) Frequency of freezes, which is the number of times the receiver's video is frozen. (6) Duration of freezes, which is the cumulative length of time where the receiver's video is frozen.[10] We calculate these metrics only for the frames where FEC is applied (i.e., where FEC affects the quality). We compute one value per call (e.g., median duration of freezes, bandwidth overhead, etc.) and then consider the percentiles over these values. For latency, we consider all frames over all calls.

QoE is difficult to measure precisely with so-called "QoE models" [68] because it depends on video-specific properties (e.g., in sports, video quality during gameplay matters more than during timeouts). But several works [7, 21, 37] have shown that key metrics for QoE (e.g., frequency of freezes, duration of freezes, bandwidth, etc.) impact the mean opinion score—the gold standard measure of QoE. These metrics also affect user interactions (e.g., users watch more video when there are fewer freezes). In fact, [19] showed that cumulative freeze duration is crucial for QoE, as well as the importance of bitrate and frequency of video freezes for live video.

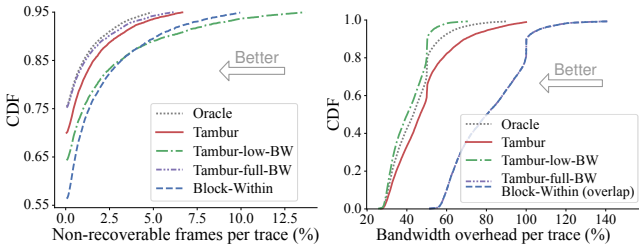
Offline evaluation. We evaluate the performance of Block-Within, Tambur, Tambur-full-BW, Tambur-0.9, Tambur-low-BW, and Oracle over the test set of traces from Teams described in §3, which was held out from the previous analyses. The packet logs provide the performance of Block-Within. We make two safe assumptions to evaluate the remaining schemes over the traces: (a) modifying the payload of a packet, but not its size, would not change whether it is lost or received; (b) reducing the size of a packet's payloads would not incur any new packet losses. Each data packet is sent identically as in the trace, the payloads for the parity packets are changed, the sizes of the parity packets are sometimes reduced, and the bitmap of packet losses from the trace is used. To satisfy the assumptions, we must force Tambur to send the number of parity symbols allocated for each frame within the frame (rather than delayed by τ frames), which we expect to *degrade Tambur's performance*. This enforcement alters the number of parity packets sent under Tambur but not how their symbols are defined. Block-Multi is excluded because it sends all parity packets after the final data packet of the final frame of the block, so its performance cannot be fairly simulated using the production traces.

Online evaluation. We evaluate prototype implementations of Block-Within, Block-Multi, Tambur, Tambur-full-BW, and Tambur-0.9 integrated with Ringmaster (the videoconferencing benchmark platform described in §4.3) via network emulation using Mahimahi [46] while simulating a Gilbert-Elliott (GE) [23] loss model over a dataset of 80 videos. Specifically, we evaluate 20 video calls from [16,47] at four constant bitrates each (namely, 500, 1000, 1500, and 2000 kbps) to isolate the effect of FEC. The bandwidth overhead is set to 50% for Block-Within (likewise, for Block-Multi and Tambur-full-BW).[11] The GE loss model is a standard loss model which is a Markov model with two states: "good" and "bad," each with an associated probability of packet loss. For a fair and realistic comparison, different coding schemes must experience the same distribution of burst losses at the frame level even though they send differing numbers of packets per frame. Therefore, we consider transitions between the states occurring once at the start of every frame (i.e., once every 33.3 ms) rather than a transition between states every packet, which is commonly used in the literature when only one packet is sent per frame. Packets within each frame are lost independently with the same probability. The modified GE channel can be viewed as a buffer overflowing for a short period, as can arise from on/off characteristics of traffic [49]. Appendix C details how we set the parameters of the GE model based on the losses from the traces.

Result highlights.

[10] We use the definition of freezes and duration of freezes from the most recent (unofficial) draft of identifiers for WebRTC's statistics [10].

[11] The bandwidth overhead is sometimes slightly higher for *all* schemes due to rounding and ensuring at least one parity packet is sent per frame.



(a) Non-recoverable frames (b) Bandwidth overhead

Figure 10: CDFs for the percent of non-recoverable frames for the 55th through 95th percentiles and the bandwidth overhead for the offline evaluation.

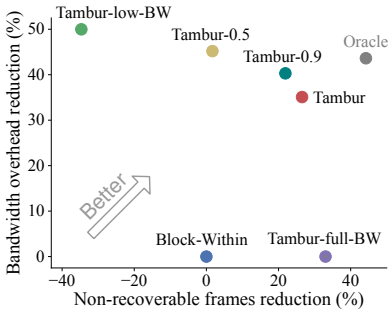


Figure 11: Sensitivity analysis of the weights for the classes used in the predictive model for the frequency of non-recoverable frames and bandwidth overhead over all of the frames where FEC is used in the traces.

- In offline evaluation, Tambur reduces the frequency of non-recoverable frames by 26.5% while using 35.1% less bandwidth overhead.
- In online evaluation, Tambur reduces frequency of non-rendered frames, frequency of freezes, and duration of freezes by 28%, 26%, and 29%, respectively compared to Block-Multi, and by 73%, 78%, and 77% compared to those of Block-Within. Block-Multi has a significantly higher latency than Block-Within (see Fig. 13b).
- Modest memory overhead and median encoding and decoding times of 575 KB, 1.7ms, and 3.4ms, respectively.

5.2 Offline evaluation

We assess only the frequency of non-recoverable frames and the bandwidth overhead for offline traces because the remaining metrics are unavailable. Fig. 10a shows the CDF of the percent of non-recoverable frames from 55th to 95th percentiles over the traces. These percentiles correspond roughly to the 92nd to 99th percentile over all traces. The Oracle reduces the total number of non-recoverable frames by 44.2% compared to Block-Within and reflects an upper bound on performance. Tambur-full-BW reduces the frequency of non-recoverable frames by 33% compared to Block-Within, indicating the potential improvements of using streaming codes. In contrast, Tambur-low-BW *increases* the frequency of non-recoverable frames by 34.7% compared to Block-Within, indicating the need for more sophisticated methods to reduce the bandwidth overhead without incurring a significant penalty in non-recoverable frames. By using a predictive model to determine the bandwidth overhead, Tambur reduces the bandwidth overhead by 35% while simultaneously reducing the number of non-recoverable frames by 26.5% compared to Block-Within (Fig. 10b). §5.3 summarizes the spectrum of bandwidth savings versus recovering frames for Tambur based on tuning the associated weight parameter.

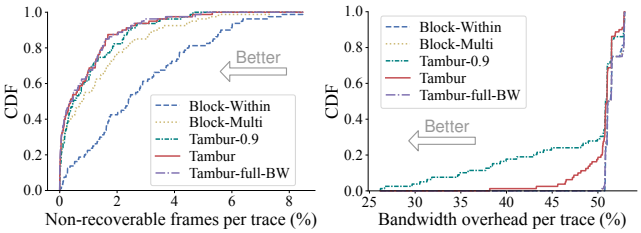
5.3 Sensitivity analysis

There is an inherent trade-off in performance between the non-recoverable frames and bandwidth overhead metrics. The ML model for Tambur is trained using a loss function with a weight of 0.999 on avoiding recovery failures and the remaining weight (i.e., 0.001) on saving bandwidth overhead (§4.3). Fig. 11 shows the impact of this parameter on the frame recovery performance of Tambur with the weight set to 0.9 (i.e., Tambur-0.9) and to 0.5. The improvement in non-recoverable frames for the two schemes are respectively 21.9% and 1.7%. The reduction in the bandwidth overhead is respectively 40.3% and 45.2%. By contrast, recall that Tambur leads to a 26.5% improvement in non-recoverable frames and reduces the bandwidth overhead by 35.1%. Reducing the value of the parameter reduces the frequency of recovering frames and increases the reduction in the bandwidth overhead. Videoconferencing service operators can use these weights as a knob to prioritize one metric over another.

5.4 Online evaluation

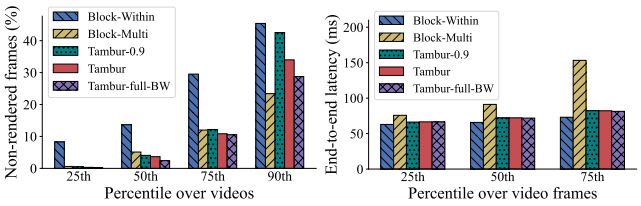
Next, we establish Tambur's potential to improve the QoE. To facilitate an easy comparison with the offline evaluation, we show the frequency of non-recoverable losses and the bandwidth overhead (as in §5.2) in Fig. 12. On average, Tambur reduces the number of non-recoverable frames by 69% compared to Block-Within and 34% compared to Block-Multi. Tambur-0.9 reduces the number of non-recoverable frames by 65% compared to Block-Within and 26% compared to Block-Multi despite Block-Multi's much higher latency (Fig. 13b). The results differ slightly from the offline evaluation at the lower percentiles because of setting the parameters of the channel based on average loss statistics over all the traces. This significantly reduced the frequency of calls with low loss rates where any coding scheme suffices to recover nearly all frames (i.e., sophisticated FEC schemes are unnecessary).

Tambur—which is conservative in risking recovery failures to save bandwidth—reduces the bandwidth overhead by 3%



(a) Non-recoverable frames (b) Bandwidth overhead

Figure 12: CDFs for the percent of non-recoverable frames and the bandwidth overhead for the online evaluation.



(a) Frequency of non-rendered frames (b) Latency of rendered frames

Figure 13: Tambur renders significantly more frames than Block-Multi and with lower latency. Tambur's modestly higher latency than Block-Within is more than offset by the improvement in rendering frames.[12]

on average of the calls. In contrast, Tambur-0.9 reduces the bandwidth overhead by an average of over 8%. These results reflect both schemes reducing the bandwidth overhead significantly on some calls but only negligibly on many others, which is expected given the loss rates of most calls. Tambur-0.9's bandwidth savings are especially pronounced at the lower percentiles (e.g., 31% at the 10th percentile and 15% at the 20th percentile). Tambur-0.9 provides a win-win by both recovering more frames and saving bandwidth despite the online evaluation reflecting out-of-sample performance for its neural network, which was trained offline over the production traces. The results further validate the trade-off between the bandwidth overhead and recovering frames discussed in §5.3.

Next, we examine the percent of non-rendered frames in Fig. 13a; recall that fewer frames are rendered than recovered due to inter-frame dependencies. Tambur reduces the frequency of failing to render frames compared to Block-Multi and Block-Within by an average of 28% and 73%, respectively. Tambur does worse than Block-Multi at the tail, but this only occurs after all schemes have a failure rate above 23%. Thus, all schemes should employ more redundancy. Tambur-0.9 decreases the frequency of failing to render frames by an average of 70% and 20% compared to Block-Within and Block-Multi, respectively. Tambur-0.9 modestly increases the frequency by 1% at the 75th percentile compared to Block-Multi. Overall, the rate of rendering frames can be improved while simultaneously reducing the bandwidth overhead for most calls. The results are the first to establish the benefits of streaming codes when there are inter-frame dependencies.

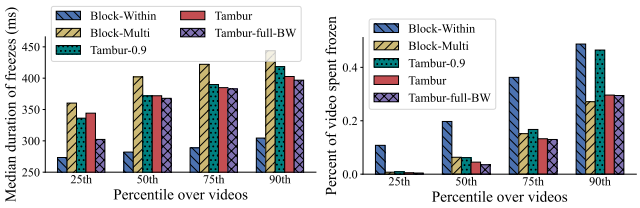
Fig. 13b shows that the end-to-end latency is within the upper limit of approximately 150ms for all schemes. Block-Within's latency is slightly lower due to a shorter encode/decode time and always recovering rendered frames using the parity of the same frame (see Fig. 15 and Fig. 16 in Appendix D); Tambur decodes 87% of frames without extra frames versus 88% for Block-Within, so the extra latency from the waiting for extra frames should really be compared to Block-Within failing to decode at all. We argue that Tambur's small cost (e.g., an extra 1.7ms to encode and 3.4ms to decode at the median) is worthwhile due to substantial improvements across the remaining QoE metrics. We also note that our implementation of Tambur's streaming code is not yet optimized for fast encoding/decoding; hence, we believe it can be significantly faster. Our goal is to establish that Tambur's streaming code is practical enough for videoconferencing applications.

Recall from Fig. 1 that Tambur reduces the frequency of freezes by 78% and 26% compared to Block-Within and Block-Multi, respectively, and Tambur-0.9 reduces the frequency of freezes by 75% and 17% compared to the two respective baselines. Fig. 14a shows that Tambur and Tambur-0.9 each reduce the median duration of freezes compared to Block-Multi by 30ms on average. Tambur and Tambur-0.9 each have a 90ms longer median duration of freezes than theBlock-Within because Block-Within has over 300% more freezes than Tambur does. Many of the extra freezes are short, reducing Block-Within's median value to below Tambur's.

Tambur-0.9 reduces the cumulative duration of freezes by an average of 69% compared to Block-Within. The cumulative duration of freezes is 17% lower for Block-Multi than for Tambur-0.9 despite Tambur-0.9 having on average 11% shorter median durations of freezes and 17% fewer freezes. While the combined effect of the frequency and duration of freezes on the QoE for Block-Multi and Tambur-0.9 are similar, recall that Tambur-0.9 also improves the bandwidth overhead and renders more frames for most traces. As such, we expect Tambur-0.9 to provide an overall higher QoE. Tambur has an average of 77% and 28% shorter cumulative durations of freezes than Block-Within and Block-Multi, respectively, which is a clear win. Tambur, Tambur-0.9, and Tambur-full-BW exhibit higher cumulative durations of freezes at the tail than Block-Multi. We argue that this matters less because the tail QoE is already bad, indicating that all schemes needed more bandwidth overhead. Appendix E explains how this phenomenon is an artifact of the implementation and includes our proposed a solution.

The benefits across QoE metrics of Tambur, Tambur-full-BW, and Tambur-0.9 suggest a markedly improved QoE compared to Block-Within and Block-Multi. Without using ML to reduce the bandwidth overhead, Tambur-full-BW offers a substantial improvement over the two baselines. Tambur and

[12] We omit the 90th percentile since over 10% of frames are not rendered.



(a) Median duration of freezes (b) Percent of video spent frozen

Figure 14: Tambur has a higher median duration of freezes than Block-Within but a significantly smaller cumulative duration of freezes because Tambur has 78% fewer freezes than Block-Within (Fig. 1). Tambur has a lower cumulative and median duration of freezes than Block-Multi.

Tambur-0.9 progressively trade off improvements in loss recovery with bandwidth overhead. Overall, the results illustrate a Pareto frontier of the benefits of streaming codes across the QoE metrics that could be studied further in future work.

6 Related work

FEC for videoconferencing. From the early days of VoIP and Internet-based audio and videoconferencing, FEC has played a key role in recovering lost packets (e.g., [50]). As standards for real-time media and conferencing developed, RTP payload formats for various FEC schemes were defined (e.g., [62]). Later, the FECFRAME working group of the IETF [58] documented traditional FEC schemes such as parity codes [9] and RS codes [57], as well as LDPC [56] and Raptor codes [65]. As the WebRTC project developed based on these standards, it also incorporated the use of FEC to protect media streams [32, 64]. All these codes are block codes. As such, RS codes (i.e., the main baselines against which Tambur was evaluated) have the best loss recovery capabilities of any of them, including fountain [40] and raptor codes [63]. FEC has also been used for rate adaptation. For example, a proposed rate adaptation algorithm for WebRTC, known as FBRA [45], uses extra FEC packets to probe for additional bandwidth, with the benefit that some of the packet losses due to self-induced congestion can be recovered by the FEC.

Streaming codes. In addition to the prior work discussed in §2 and §4, streaming codes have also been studied under various other theoretical models, such as multiplexing with two different decoding delays [4] and multiple burst losses [38]. However, these settings are not directly relevant to our focus on videoconferencing applications.

Alternatives to FEC. Prior work has explored avoiding lossy paths using overlay networks (e.g., Via [34] and J-QoS [31]). While these can be effective in some circumstances, there are two drawbacks to relying only on such approaches. Firstly, these assume that a suitable alternative path *exists* (i.e., that the lossy portion of the path is on a transit network that can be avoided, rather than on the user's home network or last-mile to the ISP, and that there is available interconnectivity with the provider's overlay network); in the current era of hybrid work, enterprises cannot completely eliminate loss through traditional QoS approaches. Secondly, when overlay networks are a feasible solution, there needs to be a careful analysis of when to apply this approach since there is a high financial cost to relaying traffic and fixed capacity on provider networks. Another alternative to FEC is using retransmission to recover losses (e.g., [30]), provided the end-to-end latency is tolerable. However, when there is both high latency and loss (e.g., in cases of acute congestion), retransmission is not always feasible [3]. Tambur provides a flexible end-to-end solution within the application that adapts to any path and is orthogonal to these other approaches.

7 Conclusion

This work introduces Tambur—a new communication scheme for bandwidth-efficient loss recovery for videoconferencing comprising two components. First, a new streaming code that bridges the gap between theoretical streaming codes and videoconferencing applications, which takes as input any given bandwidth overhead. Second, a learning-based predictive model to set the bandwidth overhead. Tambur simultaneously reduces the frequency of non-recoverable frames and the bandwidth overhead by 26.5% and 35.1%, respectively, in our evaluation over large-scale real-world traces from a commercial videoconferencing application. We also design the first videoconferencing framework for implementing and evaluating FEC schemes. The framework enables easy evaluation of the QoE benefits of new communication schemes by providing a simple interface to incorporate (a) new FEC schemes and (b) new learning-based predictive models. Using the framework, we evaluate Tambur and show improvements in QoE metrics, including 26% fewer freezes and 28% fewer non-rendered frames. The benefits establish streaming codes as a viable solution to recovering lost packets for videoconferencing applications. The results thus also show the promise of streaming codes for other live-streaming applications such as cloud gaming.

Acknowledgments

This work was funded in part by an NSF grant (CCF-1910813). We thank our anonymous reviewers for their comments and Nandita Dukkipati for shepherding this work. We also thank Sasikanth Bendapudi, Sivakumar Ananthakrishnan, Nelson Pinto, Jiannan Zheng, and Ross Cutler from Microsoft for their helpful discussions and supporting the project.

References

[1] libvpx. https://chromium.googlesource.com/webm/libvpx/.

[2] WebRTC. https://webrtc.org/.

[3] A. Badr, A. Khisti, W. Tan, and J. Apostolopoulos. Perfecting protection for interactive multimedia: A survey of forward error correction for low-delay interactive applications. *IEEE Signal Processing Magazine*, 34(2):95–113, March 2017.

[4] A. Badr, D. Lui, A. Khisti, W. Tan, X. Zhu, and J. Apostolopoulos. Multiplexed coding for multiple streams with different decoding delays. *IEEE Transactions on Information Theory*, 64(6):4365–4378, June 2018.

[5] A. Badr, P. Patil, A. Khisti, W. Tan, and J. Apostolopoulos. Layered constructions for low-delay streaming codes. *IEEE Transactions on Information Theory*, 63(1):111–141, Jan 2017.

[6] Ahmed Badr, Ashish Khisti, Wai-tian Tan, Xiaoqing Zhu, and John Apostolopoulos. FEC for VoIP using dual-delay streaming codes. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[7] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 339–350, New York, NY, USA, 2013. Association for Computing Machinery.

[8] Dziugas Baltrunas, Ahmed Elmokashfi, Amund Kvalbein, and Özgü Alay. Investigating packet loss in mobile broadband networks under mobility. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 225–233. IEEE, 2016.

[9] Ali C. Begen. RTP Payload Format for 1-D Interleaved Parity Forward Error Correction (FEC). RFC 6015, October 2010.

[10] Henrik Boström, Harald Alvestrand, and Varun Singh. Provisional identifiers for WebRTC's statistics API unofficial draft. Draft of a potential specification, W3C, July 2022. https://w3c.github.io/webrtc-provisional-stats/#RTCVideoReceiverStats-dict.

[11] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[12] Ramon Caceres, Alan Clark, and Timur Friedman. RTP Control Protocol Extended Reports (RTCP XR). RFC 3611, November 2003.

[13] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the Google congestion control for web real-time communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[14] Hyunseok Chang, Matteo Varvello, Fang Hao, and Sarit Mukherjee. Can you see me now? A measurement study of Zoom, Webex, and Meet. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 216–228, 2021.

[15] Federal Communications Commission. Measuring Broadband America, 2021. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-eleventh-report (Last accessed: 2022-02-02).

[16] Mauro Conti, Simone Milani, Ehsan Nowroozi, and Gabriele Orazi. Do not deceive your employer with a virtual background: A video conferencing manipulation-detection system. *CoRR*, abs/2106.15130, 2021.

[17] Ross Cutler, Yasaman Hosseinkashi, Jamie Pool, Senja Filipi, Robert Aichner, Yuan Tu, and Johannes Gehrke. Meeting effectiveness and inclusiveness in remote collaboration. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW1), apr 2021.

[18] Marcel Dischinger, Andreas Haeberlen, Krishna P. Gummadi, and Stefan Saroiu. Characterizing residential broadband networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 43–56, 2007.

[19] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 362–373, New York, NY, USA, 2011. Association for Computing Machinery.

[20] E. Domanovitz, S. L. Fong, and A. Khisti. An explicit rate-optimal streaming code for channels with burst and arbitrary erasures. In *2019 IEEE Information Theory Workshop (ITW)*, pages 1–5, 2019.

[21] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2017.

[22] D. Dudzicz, S. L. Fong, and A. Khisti. An explicit construction of optimal streaming codes for channels with burst and arbitrary erasures. *IEEE Transactions on Communications*, 68(1):12–25, 2020.

[23] E. O. Elliott. Estimates of Error Rates for Codes on Burst-Noise Channels. *Bell System Technical Journal*, 42(5):1977–1997, September 1963.

[24] Martin Ellis. *Understanding the performance of Internet video over residential networks*. PhD thesis, University of Glasgow, 2012.

[25] Salma Shukry Emara, Silas Fong, Baochun Li, Ashish Khisti, Wai-Tian Tan, Xiaoqing Zhu, and John Apostolopoulos. Low-latency network-adaptive error control for interactive streaming. *IEEE Transactions on Multimedia*, pages 1–1, 2021.

[26] S. L. Fong, A. Khisti, B. Li, W. Tan, X. Zhu, and J. Apostolopoulos. Optimal streaming codes for channels with burst and arbitrary erasures. *IEEE Transactions on Information Theory*, 65(7):4274–4292, July 2019.

[27] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-Latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, Renton, WA, April 2018. USENIX Association.

[28] Boni García, Micael Gallego, Francisco Gortázar, and Antonia Bertolino. Understanding and estimating quality of experience in WebRTC applications. *Computing*, 101(11):1585–1607, 2019.

[29] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40–54, 2011.

[30] Dongsu Han, Ashok Anand, Aditya Akella, and Srinivasan Seshan. RPT: Re-architecting loss protection for content-aware networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pages 71–84, 2012.

[31] Osama Haq, Cody Doucette, John W Byers, and Fahad R. Dogar. Judicious QoS using cloud overlays. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 371–385, 2020.

[32] S. Holmer, M. Shemer, and M. Paniconi. Handling packet loss in WebRTC. In *2013 IEEE International Conference on Image Processing*, pages 1860–1864, 2013.

[33] International Telecommunication Union. ITU-T G. 1010: End-User Multimedia Qos Categories. *G SERIES: Transmission Systems and Media, Digital System and Networks-Multimedia Quality of Service and Performance Generic and User-Related Aspects*, 2001.

[34] Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A. Chou, Venkata Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, et al. VIA: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 286–299, 2016.

[35] M. N. Krishnan and P. V. Kumar. Rate-optimal streaming codes for channels with burst and isolated erasures. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pages 1809–1813, June 2018.

[36] M. N. Krishnan, D. Shukla, and P. V. Kumar. A quadratic field-size rate-optimal streaming code for channels with burst and random erasures. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 852–856, 2019.

[37] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. In *Proceedings of the 2012 Internet Measurement Conference*, IMC '12, page 211–224, New York, NY, USA, 2012. Association for Computing Machinery.

[38] Z. Li, A. Khisti, and B. Girod. Correcting erasure bursts with minimum decoding delay. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 33–39, Nov 2011.

[39] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. GSO-Simulcast: Global stream orchestration in simulcast video conferencing systems. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 826–839, New York, NY, USA, 2022. Association for Computing Machinery.

[40] David J. C. MacKay. Fountain codes. *IEE Proceedings-Communications*, 152(6):1062–1068, 2005.

[41] Kyle MacMillan, Tarun Mangla, James Saxon, and Nick Feamster. Measuring the performance and network utilization of popular video conferencing applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 229–244, New York, NY, USA, 2021. Association for Computing Machinery.

[42] E. Martinian and C.-E. W. Sundberg. Burst erasure correction codes with low decoding delay. *IEEE Transactions on Information Theory*, 50(10):2494–2502, Oct 2004.

[43] E. Martinian and M. Trott. Delay-optimal burst erasure code construction. In *2007 IEEE International Symposium on Information Theory*, pages 1006–1010, June 2007.

[44] Anush Krishna Moorthy, Lark Kwon Choi, Alan Conrad Bovik, and Gustavo de Veciana. Video quality assessment on mobile devices: Subjective, behavioral and objective studies. *IEEE Journal of Selected Topics in Signal Processing*, 6(6):652–671, 2012.

[45] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion Control using FEC for Conversational Multimedia Communication. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 191–202, 2014.

[46] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, Santa Clara, CA, July 2015. USENIX Association.

[47] Ehsan Nowroozi, Ali Dehghantanha, Reza M Parizi, and Kim-Kwang Raymond Choo. A survey of machine learning techniques in adversarial image forensics. *Computers & Security*, page 102092, 2020.

[48] P. Orosz, T. Skopkó, Z. Nagy, P. Varga, and L. Gyimóthi. A case study on correlating video QoS and QoE. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–5, 2014.

[49] Kohong Park and Walter Willinger. *Sele-Similar network traffic and performance evaluation*. Wiley & Son, 2000.

[50] C. Perkins, O. Hodson, and V. Hardman. A survey of packet loss recovery techniques for streaming audio. *IEEE Network*, 12(5):40–48, 1998.

[51] James S. Plank, Ethan L. Miller, Kevin M. Greenan, Benjamin A. Arnold, John A. Burnum, Adam W. Disney, and Allen C. McBride. GF-Complete: A comprehensive open source library for galois field arithmetic version 1.02, 2014.

[52] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[53] Yining Qi and Mingyuan Dai. The effect of frame freezing and frame skipping on video quality. In *2006 International Conference on Intelligent Information Hiding and Multimedia*, pages 423–426, 2006.

[54] Ramya Raghavendra and Elizabeth M. Belding. Characterizing high-bandwidth real-time video traffic in residential broadband networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 597–602. IEEE, 2010.

[55] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[56] Vincent Roca, Mathieu Cunche, and Jerome Lacan. Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME. RFC 6816, December 2012.

[57] Vincent Roca, Mathieu Cunche, Jerome Lacan, Amine Bouabdallah, and Kazuhisa Matsuzono. Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME. RFC 6865, February 2013.

[58] Vincent Roca, Mark Watson, and Ali C. Begen. Forward Error Correction (FEC) Framework. RFC 6363, October 2011.

[59] Michael Rudow and K. V. Rashmi. Learning-augmented streaming codes are approximately optimal for variable-size messages. In *2022 IEEE International Symposium on Information Theory (ISIT)*, pages 474–479, 2022.

[60] Michael Rudow and K. V. Rashmi. Streaming codes for variable-size messages. *IEEE Transactions on Information Theory*, 68(9):5823–5849, 2022.

[61] Michael Rudow and K.V. Rashmi. Online versus offline rate in streaming codes for variable-size messages. *IEEE Transactions on Information Theory*, pages 1–1, 2023.

[62] Henning Schulzrinne and Jonathan Rosenberg. An RTP Payload Format for Generic Forward Error Correction. RFC 2733, December 1999.

[63] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, 2006.

[64] Justin Uberti. WebRTC Forward Error Correction Requirements. RFC 8854, January 2021.

[65] Mark Watson, Thomas Stockhammer, and Mike Luby. Raptor Forward Error Correction (FEC) Schemes for FECFRAME. RFC 6681, August 2012.

[66] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and

Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[67] Mo Zanaty, Varun Singh, Ali C. Begen, and Giridhar Mandyam. RTP Payload Format for Flexible Forward Error Correction (FEC). RFC 8627, July 2019.

[68] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. SENSEI: Aligning video streaming quality with dynamic user sensitivity. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 303–320. USENIX Association, April 2021.

Appendices

A Recovering a burst with Tambur's streaming code

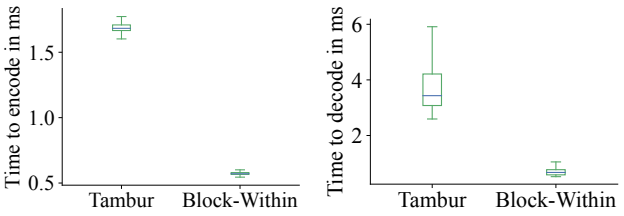
Consider a burst of length b starting in frames i and delay constraint τ. Suppose all frames before the burst have been decoded. First, the received symbols of $P[i],\ldots,P[i+b-1]$ as well as $P[i+b],\ldots,P[i+\tau]$ are used to decode the lost symbols of $V[i],\ldots,V[i+b-1]$. Second, each $U[j]$ for $j \in \{i,\ldots,i+b\}$ is decoded using $P[j+\tau]$. In both steps, decoding follows from solving a system of linear equations.

B Tambur's streaming code's flow network

The graph of the flow network at a high level represents each $P[i]$ that may be used in decoding with a node with an edge into nodes corresponding to each of $U[i]$, $U[i-\tau],V[i],\ldots,V[i-\tau]$, where one unit of flow represents decoding one symbol. The flow network is small (i.e., at most $(5\tau+3)$ vertices and $(2\tau^2+11\tau+5)$ edges for $\tau=3$). Therefore, the time to solve it is negligible compared to solving the system of linear equations.

C Parameters of the GE channel

To set the parameters of the GE channel for the offline evaluation, we first identify settings that match several aggregate statistics of the production traces as follows. The probability of transitioning from the bad state to the good state (respectively, vice versa) is the mean over traces of one divided by the mean length of bursts (respectively, guard spaces) in frames. The probability of loss in the bad state equals the mean over traces of the multi-frame burstiness. The probability of loss in the good state is then set so that the expected loss rate matches the mean loss rate over traces given the other three parameters. To ensure our results hold for varying network conditions, we then draw the values for each of the four parameters uniformly at random from intervals around these values (rounded to increments of 0.05) as follows. The probability of transitioning from the good state to the bad state and vice versa are distributed as Uniform(0, 0.05) and Uniform(.75, .9), respectively. The probability of loss in the good and bad states are distributed as Uniform(0, 0.05) and Uniform(0.05, 1), respectively.[13]



(a) Encoding (b) Decoding

Figure 15: The encoding and decoding times are modest.

D Encoding and decoding overheads

We compare the encoding and decoding time for Tambur with that of Block-Within, which is the fastest of all the baselines (Fig. 15). As seen in Fig. 15, the time to encode and decode is comparable to Block-Within and is only a small fraction of the end-to-end latency budget of 150 ms. The median times for encoding are 1.7ms and .6ms for Tambur and Block-Within, respectively, whereas decoding takes 3.4ms and .7ms for Tambur and Block-Within, respectively. Because Tambur operates over multiple frames of varying sizes, encoding and decoding times are slightly longer and more variable. Our implementation of Tambur requires a fixed amount of memory of approximately 575 KB during encoding and decoding.

But times for encoding and decoding are just a small component of the end-to-end latency. The 50ms one-way delay and the number of extra frames used in decoding (see Fig. 16) have more pronounced effects. Recall that each additional frame used in adds approximately 33 ms to the end-to-end latency, so using fewer extra frames is faster. Tambur does not decode within the same frame only 1% more frequently than Block-Within, which cannot use extra frames in decoding. Tambur uses extra frames to decode only 8% of the time. Block-Multi decodes 24%, 23%, 23%, and 23% of frames with $0,1,2$, and 3 extra frames, respectively. Each extra frame adds ≈ 33 ms to the end-to-end latency.

[13] The results were similar when we varied the ranges.

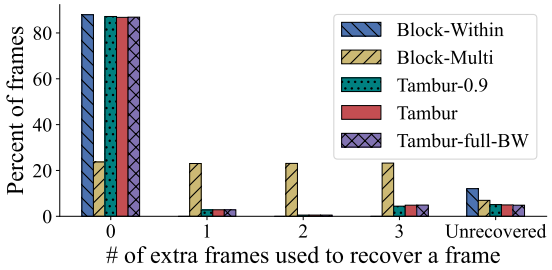


Figure 16: Tambur recovers nearly as many frames as Block-Within using no extra frames and also recovers more overall. Block-Multi recovers an approximately equal number of frames using 0, 1, 2, and 3 extra frames.

E Tail duration of freezes

Recall from Fig. 14b that Tambur, Tambur-0.9, and Tambur-full-BW have higher tail durations of freezes than Block-Multi. The reason for the poor performance is threefold. First, Tambur, Tambur-0.9, and Tambur-full-BW fail to render more frames at the tail, as was discussed in §5.2. Second, the sender generates a keyframe (often ending a freeze) once it learns of recovery failures. Because Block-Within can only recover a frame using the parity packets within the same frame, a keyframe is requested 3 frames sooner (i.e., ≈ 100 ms faster) than when Tambur (or Tambur-0.9) is used. Many of the 78% of freezes under the Block-Within where Tambur does not freeze are therefore short and shift the entire distribution of cumulative duration of freezes for Block-Within, including the tail; if we added 0ms freezes for Tambur (or Tambur-0.9) for these instances, their distributions would likewise shift. Third, encoding across multiple frames can make it harder to recover a keyframe triggered by a freeze of several lost frames. This phenomenon does not impact Block-Within and affects Block-Multi less than any of Tambur, Tambur-0.9, and Tambur-full-BW (e.g., does not affect on Block-Multi whenever the keyframe is in the first position within the block of $(\tau+1)=4$ frames). The phenomenon also contributes to a difference in the frequency of recovered frames (Fig. 12a) and rendered frames (Fig. 13a). There is a natural solution that is outside of the scope of this work. When the sender triggers a new keyframe due to a loss, it should stop taking linear combinations of frames from before the new keyframe. Doing so will strictly (a) increase the frequency of displaying frames and (b) decrease the mean and median duration of freezes. It will benefit Tambur, Tambur-0.9, and Tambur-full-BW the most, but it will also improve Block-Multi to a lesser extent.

F Analysis of recovering bursts

Next, we evaluate Tambur's capabilities for recovering bursts of packets across multiple frames; to do so fairly, we must fix the bandwidth overhead, so "Tambur" refers to Tambur-full-BW for the remainder of §F. Fig. 17 shows the distribution of the number of packets recovered for each burst length (in frames) for the offline evaluation. In Fig. 17, the distribution of the number of packets recovered for each burst length (in frames) is shown. Bursts encompassing 2, 3, and 4 frames constitute 23%, 7%, and 3.3% of all lossy events, respectively. For these losses, Tambur recovers all lossy frames 66.8%, 103%, and 97.3% more frequently than Block-Within. For the longer (less frequent) bursts of lengths 3 and 4, when the bandwidth overhead is insufficient, Tambur fails to recover any frames 65.9% and 87% more frequently than the Block-Within. This follows from the Block-Within being more likely to recover some (but not all) of the frames when there is insufficient bandwidth overhead to recover all losses. In contrast, when the bandwidth overhead is insufficient to recover a burst in its entirety, streaming codes are likely to fail to recover all of the frames. However, note that the overall performance of Tambur is still better than the Block-Within: Tambur recovers 21.8%, 12.4%, and 2.3% more frames than the Block-Within for bursts of 2,3, and 4 frames, respectively. Tambur also outperforms Block-Within in recovering losses limited to a single frame, as parity packets sent with later frames can be used in recovery. In short, Tambur performs significantly better for bursts across up to 3 frames than Block-Within and offers

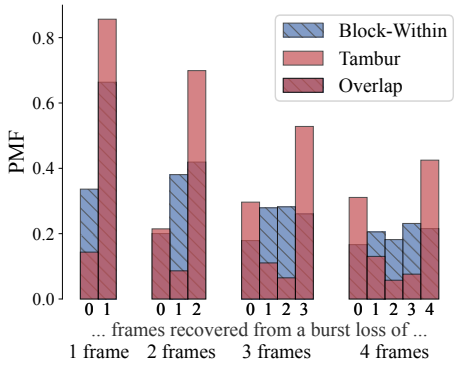


Figure 17: Given the same bandwidth budget as Block-Within, Tambur is more likely to recover all or zero frames from a burst loss over production traces.

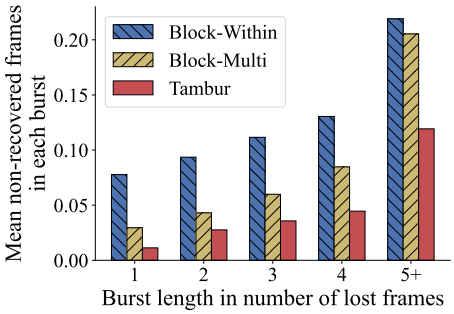


Figure 18: Given the same bandwidth budget as Block-Within/Block-Multi, Tambur provides greater improvement for longer bursts over an emulated network.

more modest gains for bursts across 4 frames.

We also evaluate Tambur's effectiveness at recovering bursts in the online evaluation. Because the loss of a single packet of a frame means that the frame is "lost" under our definition of a burst, longer bursts usually only involve being in the bad state for one, two, or sometimes three frames. We consider the mean number of frames recovered among a burst encompassing 1, 2, 3, 4, or greater than 4 frames in Fig. 18. Tambur reduces the frequency of non-recoverable frames by 70.5%, 68.0%, and 65.8% compared to Block-Within over bursts of 2, 3, and 4 frames respectively. Tambur reduces the frequency of non-recoverable frames by 35.8%, 40.3%, and 47.4% compared to Block-Multi over bursts of 2, 3, and 4, respectively.

Gemel: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge

Arthi Padmanabhan[⋆§] Neil Agarwal[⋆¶] Anand Iyer[†] Ganesh Ananthanarayanan[†]
Yuanchao Shu[‡] Nikolaos Karianakis[†] Guoqing Harry Xu[§] Ravi Netravali[¶]

[§]UCLA [†]Microsoft Research [‡]Zhejiang University [¶]Princeton University

Abstract

Video analytics pipelines have steadily shifted to edge deployments to reduce bandwidth overheads and privacy violations, but in doing so, face an ever-growing resource tension. Most notably, edge-box GPUs lack the memory needed to concurrently house the growing number of (increasingly complex) models for real-time inference. Unfortunately, existing solutions that rely on time/space sharing of GPU resources are insufficient as the required swapping delays result in unacceptable frame drops and accuracy loss. We present *model merging*, a new memory management technique that exploits architectural similarities between edge vision models by judiciously sharing their layers (including weights) to reduce workload memory costs and swapping delays. Our system, Gemel, efficiently integrates merging into existing pipelines by (1) leveraging several guiding observations about per-model memory usage and inter-layer dependencies to quickly identify fruitful and accuracy-preserving merging configurations, and (2) altering edge inference schedules to maximize merging benefits. Experiments across diverse workloads reveal that Gemel reduces memory usage by up to 60.7%, and improves overall accuracy by 8-39% relative to time or space sharing alone.

1 Introduction

Fueled by the proliferation of camera deployments and significant advances in deep neural networks (DNNs) for vision processing (e.g., classification, detection) [19, 28, 46, 69, 74], live video analytics have rapidly grown in popularity [25, 35, 60, 71, 113]. Major cities and organizations around the world now employ thousands of cameras to monitor intersections, homes, retail spaces, factories, and more [1, 5, 6, 10]. The generated video feeds are continuously and automatically queried using DNNs to power long-running applications for autonomous driving, footfall tracking, traffic coordination, business analytics, and surveillance [2, 11–13, 34].

In order to deliver highly-accurate query responses in real time, video analytics deployments have steadily migrated to the edge [25, 78, 107]. More specifically, pipelines routinely incorporate *on-premise* edge servers (e.g., Microsoft Azure Stack Edge [4], Amazon Outposts [3]) that run in hyper-proximity to cameras (in contrast to traditional edge servers [32, 37, 79, 104]), and possess on-board GPUs to aid video processing. These *edge boxes* are used to complement (or even replace [21, 29]) distant cloud servers by locally performing as many inference tasks on live video streams as possible [29, 53, 71, 117]. Generating responses directly on edge boxes reduces transfer delays for shipping data-dense video over wireless links [44, 73, 117] while also bringing resilience to outbound edge-network link failures [7, 80] and compliance with regional data privacy restrictions [77, 85].

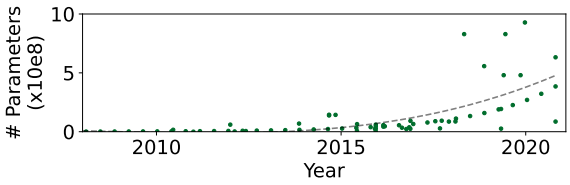


Figure 1: Parameter counts in popular vision DNNs over time. Data drawn from [92].

To reap the above benefits, video analytics deployments must operate under the limited computation resources offered by edge boxes. On the one hand, due to cost, power, and space constraints, edge boxes typically possess weaker GPUs than their cloud counterparts [4, 21, 95]. On the other hand, analytics deployments face rapidly increasing workloads due to the following trends: (1) more camera feeds to analyze [21, 53, 55], (2) more models to run due to increased popularity and shifts to bring-your-own-model platforms [16, 24, 38, 54], and (3) increased model complexity, primarily through growing numbers of layers and parameters (Figure 1) [15, 56, 57, 108]. Taken together, the result is an ever-worsening resource picture for edge video analytics.

Problems. Although GPU computation resources are holistically constrained on edge boxes, this paper focuses on *GPU memory restrictions*, which have become a primary bottleneck in edge video analytics for three main reasons. First, GPU memory is costly due to its high-bandwidth nature [83, 86, 93], and is thus unlikely to keep pace with the ever-growing memory needs of DNNs (Figure 1). Second, we empirically find that existing memory management techniques that time/space-share GPU resources [26, 39, 50, 56, 94, 110] are insufficient for edge video analytics, resulting in skipped processing on 19-84% of frames, and corresponding accuracy drops up to 43% (§3). The underlying reason is that the costs of loading vision DNNs into GPU memory (i.e., swapping) are prohibitive and often exceed the corresponding inference times, leading to sub-frame-rate (<30 fps) processing and dropped frames due to SLA violations [94, 114]. Such accuracy drops are unacceptable for important vision tasks, especially given that each generation of vision DNNs brings only 2-10% of accuracy boosts – that after painstaking tuning [22, 52, 64, 98]. Third, compared to computation bottlenecks [29, 39, 40, 60, 71], GPU memory restrictions during inference have been far less explored in video analytics.

⋆ These authors contributed equally to this work.

Contributions. We tackle this memory challenge by making two main contributions described below. The design and evaluation of our solution are based on a wide range of popular vision DNNs, tasks, videos, and resource settings that reflect workloads observed in both our own multi-city pilot video analytics deployment and in prior studies (§2).

Our first contribution is *model merging*, a fundamentally new approach to tackling GPU memory bottlenecks in edge video analytics that is complementary to time/space-sharing strategies (§4). With merging, we aim to share *architecturally identical* layers across the models in a workload such that only one copy of each shared layer (i.e., one set of weights) must be loaded into GPU memory for all models that include it. In doing so, merging reduces both the number of swaps required to run a workload (by reducing the overall memory footprint) and the cost of each swap (by lowering the amount of new data to load into GPU memory).

Our merging approach is motivated by our (surprising) finding that vision DNNs share substantial numbers of layers that are architecturally (i.e., excluding weights) identical (§4.1). Such commonalities arise not only between identical models (100% sharing), but also across model variants in the same (up to 84.6%) and in different (up to 96.3%) families. The reason is that, despite their (potentially) different goals, vision DNNs ultimately employ traditional computer vision (CV) operations (e.g., convolutions) [22, 64], operate on unified input formats (e.g., raw frames), and perform object-centric tasks (e.g., detection, classification) that rely on common features such as edges, corners, and motion [27, 31, 65, 66, 88, 106, 118, 119].

Our analysis reveals that exploiting these architectural commonalities via merging has the potential to substantially lower memory usage (17.9-86.4%) and boost accuracy (by up to 50%) in practice. However, achieving those benefits is complicated by the fact that edge vision models typically use different weights for common layers due to training non-linearities [62, 63] and variance in target tasks, objects, and videos; and yet, merging requires using unified weights for each shared layer. Digging deeper, we observe that there exists an *inverse relationship* between the number of shared layers and achieved accuracy during retraining. Intuitively, this is because for shared layers to use unified weights, other layers must adjust their weights accordingly during retraining; the more layers shared, the harder it is for (the fewer) other layers to find weights to accommodate such constraints and successfully learn the target functions [23, 70]. Worse, determining the right layers to merge is further complicated by the fact that (1) it is difficult to predict precisely how many layers will be shareable before accuracy violations occur, and (2) each instance of retraining is costly.

Our second contribution is Gemel, an end-to-end system that practically incorporates model merging into edge video analytics by automatically finding and exploiting merging opportunities across user-registered vision DNNs (§5). Gemel tackles the above challenges by leveraging two key observations: (1) vision DNNs routinely exhibit power-law distributions whereby a small percentage of layers, often towards the end of a model, account for most of the model's memory usage, and (2) merging decisions are agnostic to inter-layer dependencies, and accordingly, a layer's mergeability does not improve if other layers are also shared.

Building on these observations, Gemel follows an *incremental* merging process whereby it attempts to share one additional layer during each iteration, and selects new layers in a memory-forward manner, i.e., prioritizing the (few) memory-heavy layers. In essence, this approach aims to reap most of the potential memory savings as quickly, and with as few shared layers, as possible. Gemel further accelerates the merging process by taking an adaptive approach to retraining that detects and leverages signs of early successes and failures. At the end of each successful iteration, Gemel ships the resulting merged models to the appropriate edge servers, and carefully alters the time/space-sharing scheduler – a merging-aware variant of Nexus [94] in our implementation – to maximize merging benefits, i.e., by organizing merged models to reduce the number of swaps, and the delay for each one. Importantly, Gemel verifies that merging configurations meet accuracy targets *prior* to deployment at the edge, and also periodically tracks data drift.

Results. We evaluated Gemel on a wide range of workloads and edge settings (§2, §6.3). Overall, Gemel reduces memory requirements by up to 60.7%, which is 5.9-52.3% more than stem-sharing approaches that are fundamentally restricted to sharing contiguous layers from the start of models (Mainstream [59]), and within 9.3-29.0% of the theoretical maximum savings (that disregard layer weights). These memory savings lead to 13-44% fewer skipped frames and overall accuracy improvements of 8-39% compared to space/time-sharing GPU schedulers alone (Nexus [94]). Source code and datasets for Gemel are available at https://github.com/artpad6/gemel_nsdi23.

2 Methodology & Pilot Study

We begin by describing the workloads used in this paper. They were largely derived from our experience in deploying a pilot video analytics system in collaboration with two major US cities (one per coast), for road traffic monitoring.

Models and tasks. In line with other video analytics frameworks [16, 24, 38, 54], users in our deployment provided pretrained models when registering queries to run on different video feeds. Due to the complexity of model development, we observe that users opt to leverage existing (popular) architectures geared for their target task (e.g., YOLOv3 for object detection), and train those models for specific object(s) of interest and datasets (e.g., detecting vehicles at Main St.) to generate a unique set of weights. Despite being allowed, custom architectures were never provided in our deployment.

Accordingly, we selected the 7 most popular families of models across our pilot deployment and recent literature [21,26,49,50,53,59–61,71,109]: YOLO, Faster RCNN, ResNet, VGG, SSD, Inception, and Mobilenet. From each family, we selected up to 4 model variants (if available) that exhibit different degrees of complexity and compression. For instance, from YOLO, we consider {YOLOv3, Tiny YOLOv3}; similarly, we consider ResNet{18, 50, 101, 152}. The selected models focus on two tasks – object classification and detection – and for each, we train different versions for all combinations of the following objects: people and vehicles (e.g., cars, trucks, motorbikes). Classification and detection accuracy are measured using F1 and mAP [36].

Videos. Our dataset consists of video streams from 12 cameras in our pilot deployment that span two metropolitan areas. From each region, we consider cameras at adjacent intersections, and those spaced farther apart within the same metropolitan area; this enables us to consider different edge box placements, i.e., at a traffic intersection vs. further upstream to service a slightly larger geographic location. From each stream, we scraped 120 minutes of video that cover 24-hour periods from four times of the year.

Edge boxes. Our review of on-premise edge boxes focused on 5 commercial offerings: Microsoft Azure Stack Edge [4], Amazon Outposts [3], Sony REA [97], NVIDIA Jetson [8], and Hailo Edge-AI-box [43]. These servers each possess onboard GPUs and offer 2-16 GB of total GPU memory. Since edge inferences do not typically span multiple GPUs, we focus on model merging and inference scheduling *per GPU*. This does not restrict Gemel to single-GPU settings; rather, it means that our merging and scheduling techniques are applied separately to the DNNs in each GPU, with the assumption that each merged model runs on only one GPU.

Workload construction. Recent works highlight that 10s of videos are usually routed to each edge box [13, 53], which runs upwards of 10 queries (or DNNs) on each feed [16, 21]. Our experience was similar: it was typical to direct the max possible number of feeds to an edge box, with the goal of *minimizing the number of edge boxes required to process the workload*. To cover this space, and since we focus on per-GPU inference optimization, we generated an exhaustive list of all possible workloads sized between 2-50 DNNs using the models above. We then sorted the list in terms of the potential (percentage) memory savings (using the methodology from §4), and selected 15 workloads: 3 random workloads from the lower quartile (i.e., *Low Potential (LP1-3)*), 6 from the middle 50% (i.e., *Medium Potential (MP1-6)*), and 6 from the upper quartile (i.e., *High Potential (HP1-6)*). We chose this ratio to match that from our deployment. MP and HP workloads each constituted 30-50% of the total workloads since (1) users tended to employ the same few model variants from a limited set of popular families, and (2) each user typically used the same architecture (but not weights) for different feeds in a region. LP workloads were less common (<20%), and arose from different users opting for different model families.

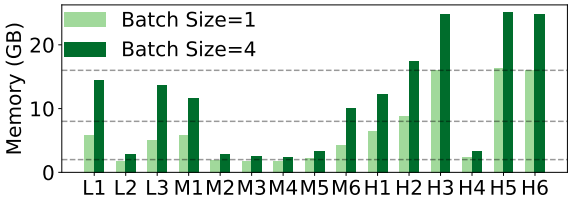


Figure 2: Per-workload memory requirements for two popular batch sizes used in video analytics [94]. Dashed lines represent the available GPU memory on several commercial edge boxes.

Each workload was randomly assigned to one of the cities, with the constituent models being randomly paired with the available videos. The extended version [82] details the workloads, each of which exhibits heterogeneity in terms of the families, tasks, videos, and (combinations of) target objects. In summary, the workloads contain 3-42 queries (avg: 15) across 3-7 video feeds (avg: 5), featuring 2-10 unique models (avg: 6) and 2-5 different objects (avg: 4). We consider additional workloads, models, objects, and videos in §6.3.

Result presentation. End-to-end accuracy depends on the available GPU memory. However, each workload requires a different minimum amount of memory to run, i.e., the GPU should be able to load/run the most memory-intensive model in isolation for a batch size of 1. Further, the memory needed to avoid swapping (i.e., to load all models and run one at a time) also varies per workload; we call this no_swap. To ensure comparability across all presented accuracy results and to focus on memory-bottlenecked scenarios, we assign each workload three memory settings to be evaluated on (listed in [82]): (1) the minimum value (*min*), (2) 50% of the no_swap value (*50%*), and (3) 75% of the no_swap value (*75%*).

3 Motivation

3.1 Memory Pressure in Edge Video Analytics

To run inference with a given model, that model's layers and parameters must be loaded into the GPU's memory, with sufficient space reserved to house intermediate data generated while running, e.g., activations. The amount of data generated (and thus, memory consumed) during inference depends on both the model architecture and the batch size used; a higher batch size typically requires more memory.

Figure 2 shows the total amount of memory (i.e., for both loading and running) required for each of our workloads and two batch sizes; the listed numbers exclude the fixed memory that ML frameworks reserve for operation, e.g., 0.8 GB for PyTorch [18]. As shown, many workloads do not directly fit into edge box GPUs, and the number of edge boxes necessary to support a given workload can quickly escalate. For instance, even with a batch size of 1 frame, 73% of our workloads need more than one edge box possessing 2 GB of GPU memory; with a batch size of 4, 60% and 27% require more than one edge box with 8 GB and 16 GB of memory.

Model	Load Memory (Time)	Run Memory (Time)		
		BS=1	BS=2	BS=4
YOLOv3	0.24 (49.5)	0.52 (17.0)	0.73 (24.0)	1.22 (39.9)
ResNet152	0.24 (73.3)	0.65 (24.8)	0.98 (26.3)	1.71 (26.7)
ResNet50	0.12 (27.1)	0.35 (8.4)	0.50 (8.5)	0.84 (8.5)
VGG16	0.54 (72.2)	0.74 (2.1)	0.89 (2.4)	1.18 (2.4)
Tiny YOLOv3	0.04 (6.7)	0.15 (3.0)	0.18 (5.2)	0.24 (5.2)
Faster RCNN	0.73 (117.3)	3.70 (115.4)	6.96 (210.1)	12.47 (379.4)
Inceptionv3	0.12 (11.8)	0.19 (9.1)	0.23 (9.1)	0.34 (9.1)
SSD-VGG	0.11 (16.1)	0.23 (16.5)	0.33 (25.7)	0.51 (44.6)

Table 1: Memory (GB) and time (ms) requirements for loading/running inference with 3 different batch sizes (in frames). Run memory values include load values, but exclude memory needs of serving frameworks. Results use a Tesla P100 GPU.

Table 1 breaks this memory pressure down further by listing the amount of loading and running memory required for representative models in our workloads. When analyzed in the context of the scale of edge video analytics workloads, the picture is bleak, even with a batch size of 1. For example, a 2 GB edge box can support only 1, 2, or 3 VGG16, YOLOv3, or ResNet50 models, respectively, after accounting for the memory needs of the serving framework. Moving up to 8 and 16 GB edge boxes (of course) helps, but not enough, e.g., an 8 GB box can support 13 YOLOv3 or 2 Faster RCNN models, both of which are a drastic drop from the 10s of models that workloads already involve (§2).

3.2 Limitations of Existing GPU Memory Management

Space and time sharing. Existing learning frameworks recommend model allocation at the granularity of an entire GPU [56]. Space-sharing techniques [14, 17] eschew this exclusivity and partition GPU memory per model. Although space-sharing approaches are effective when a workload's models can fit together in GPU memory, they are insufficient when that does not hold, which is common at the edge (§3.1)

There are two natural solutions when a workload's models cannot fit together in the target GPU's memory. The first is to place models on *different* GPUs [39, 94], which resource-constrained edge settings cannot afford. The second is to *time share* the models' execution in the GPU by *swapping* them in and out of GPU memory (from CPU, via a PCIe interface) [26, 39, 50, 94, 110]. However, as we will show next, time-sharing techniques are bottlenecked by frequent model swapping, which severely limits their utility. More recently, SwapAdvisor [50] and Antman [110] proposed swapping at finer granularities, e.g., individual or a few layers. However, even these approaches are limited in our case because a handful of layers in vision DNNs typically account for most memory usage (§5.2); edge boxes often lack the GPU memory to concurrently house even these expensive singular layers.

We evaluated time-sharing strategies in our setting by considering a hybrid version that *packs* models into GPU memory, and executes as many models as possible while ensuring that swapping costs for the next model to run are hidden. Concretely, we extend Nexus [94] to incorporate such pipelining. Our variant first organizes models in round-robin order (as Nexus does), and profiles the workload offline to determine the best global list of per-model batch sizes that maximizes the minimum achieved per-model throughput while adhering to an SLA (i.e., a per-frame processing deadline). Using those batch sizes, the scheduler traverses the round robin order with the goal of minimizing GPU idle time: when loading the next model, if there does not exist sufficient memory to load both parameters and intermediates, the most recently run model (i.e., the one whose next use is in the most distant future) is evicted to make space.

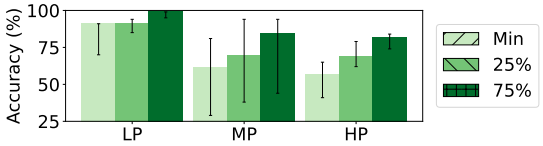


Figure 3: Achieved accuracy with time/space-sharing alone (i.e., using our Nexus variant) for different memory availability (following the definitions in §2). Bars list results for the median workload in each class, with error bars spanning min to max.

Figure 3 shows the accuracy of the Nexus variant on our workloads with an SLA of 100 ms; we saw similar trends for other common SLAs in video analytics [94]. As shown, accuracy drops are substantial, growing up to 43% relative to a setting when there exists sufficient memory to house all models at once. The root cause is the disproportionately high loading times of vision DNNs that must be incurred when swapping. As shown in Table 1, per-model loading delays are 0.98-34.4× larger than the corresponding inference times (for batch size 1). When facing the strict SLAs of video analytics, these loading costs result in the inability to keep pace with incoming frame rates, and thus, dropped (unprocessed) frames; the Nexus variant skipped 19-84% of frames.

Predicting workload characteristics. Another approach is to selectively preload models based on predictions of the target workload [115], e.g., deprioritizing inference on streams at night due to lack of activity. However, in edge video analytics, spatial correlation between streams results in model demands being highly correlated [55, 60, 71, 76].

Compression and quantization. These techniques generate lighter model variants that impose lower memory (and compute) footprints and deliver lower inference times. Some families offer off-the-shelf compressed variants (e.g., Tiny YOLOv3), and techniques such as neural architecture search can be used to develop cheaper variants that are amenable to deployment constraints [40]. Regardless, in reducing model complexity, these cheaper model variants typically sacrifice accuracy and are more susceptible to drift, relative to their more heavy-weight counterparts [21, 100]; consequently, determining the feasibility of using such models in a given setting requires careful tuning and analysis by domain experts.

We consider compression and quantization as orthogonal to merging for two reasons. First, in common workloads that involve a mix of models and tasks (§2), it may not be possible to compress all of the models while delivering sufficient

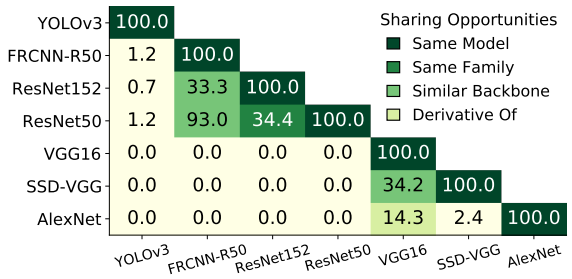


Figure 4: Percentage of architecturally identical layers across different model pairs. See Figure 20 for an extended version.

accuracy. However, even a handful of non-compressed models can exhaust the available GPU memory (§3.1). Second, compressed models exhibit sharing opportunities: our workloads include compressed and non-compressed models (§2), and our results show that Gemel is effective for both (§6).

4 Our Approach: Model Merging

To address the high model loading costs that plague existing memory management strategies when workloads cannot fit together in a GPU's memory (§3.2), we propose *model merging*. Merging is complementary to time/space sharing of GPU memory, and its goal is straightforward: share layers across models such that only one copy of each shared layer (i.e., layer definition and weights) must be loaded into GPU memory and can be used during inference for all of the models that include it. The benefits are two-fold: (1) reduce the overall memory footprint of a workload, thereby enabling edge boxes to house more models in parallel and perform fewer swaps (or equivalently, lower the number of edge boxes needed to run the workload), and (2) accelerate any remaining swaps by reducing the amount of extra memory that the next model to load requires. Note that merging does not involve sharing intermediates (i.e., layer outputs) for a common layer because models may run on different videos (and thus, inputs). We next highlight the promise for merging in edge video analytics (§4.1), and then lay out the challenges associated with realizing merging in practice (§4.2).

4.1 Opportunities

Commonality of layers. A layer is characterized by both its architecture and its weights. In ML frameworks (e.g., PyTorch, TensorFlow), the architecture is defined by first specifying a layer type (e.g., convolutional, linear, batch normalization), which in turn indicates how the layer transforms inputs, and dictates the set of defining parameters that must be specified (e.g., convolutional: kernel, stride, etc., linear: # of input features, bias, etc.). A layer's weights are a matrix of numbers whose dimensions match the layer structure. To successfully share a layer across a set of models, that layer must be *architecturally* identical in each model, but its weights need not be the same across appearances.

Architectural equivalence is determined directly from the model definition in the ML framework (i.e., no inference required): the layers must be of the same type, with identical values for type-specific properties. Using this approach, we studied pairs of 24 different models to identify and analyze layers with identical architectures; Figure 20 presents our comprehensive results. Below, we summarize our findings; Figure 4 lists results for representative model pairs.

Model pairs fall into one of three categories: (1) instances of the same model, (2) different models in the same family (e.g., ResNet variants), and (3) different models in different families. Multiple instances of the same model unsurprisingly match on every layer; this favorable scenario is not uncommon in edge video analytics, as several model architectures tend to dominate the landscape [20] and a given model can be employed on different video feeds or in search of different objects (§2). More interestingly, we also observe sharing opportunities across different models from the same (up to 84.6%) and divergent (up to 96.3%) families.

Models within the same family exhibit significant sharing opportunities as larger variants are typically extended versions of the original base model. For instance, ResNet models share ResNet blocks (groups of 2-3 convolutional layers) that are repeated at different frequencies, as well as the first convolutional layer and final fully-connected layer. As a result, all 41 layers of ResNet18 are shared with ResNet34 (Figure 19). Similarly, in the VGG family, models share the exact same base architecture and add different numbers of convolutional layers, e.g., VGG19 shares all 16 of VGG16's layers (13 convolutional, 3 fully-connected; Figure 5 (left)).

Sharing for models in different families comes in two main forms: (a) 'similar backbones' and (b) 'derivatives of.' Scenario (a) includes pairs of detectors that use the same (or similar) backbone networks for feature extraction, e.g., SSDs that use any VGG backbone, or FasterRCNNs that use any ResNet backbone. (a) also includes pairs of classifiers and detectors where the classifier (or a variant) is used as the detector's backbone. For instance, every layer in the ResNet50 backbone of FasterRCNN (which constitutes 51% of the detector's layers) appears in the ResNet101 classifier. Similar examples include SSD-VGG with any VGG variant, and SSD-MobileNet with MobileNet. Scenario (b) involves cases where one model family was derived directly from another. For example, VGG was developed by replacing AlexNet's large kernels with multiple smaller ones [96]; VGG16 and AlexNet share 3 out of 16 layers, including 2 fully-connected layers at the end (Figure 5 (right)). Other examples include InceptionNetV3 [102] with GoogLeNet [101].

In summary, 43% of all pairs of different models present sharing opportunities. Of those with substantial ($\geq$ 10%) common layers, 51% have models in the same family, while 49% involve models from different families; for the latter, 76% are 'similar backbones' and 24% are 'derivatives of.'

These layer similarities generally follow from the fact that the considered models are all vision processing DNNs. That

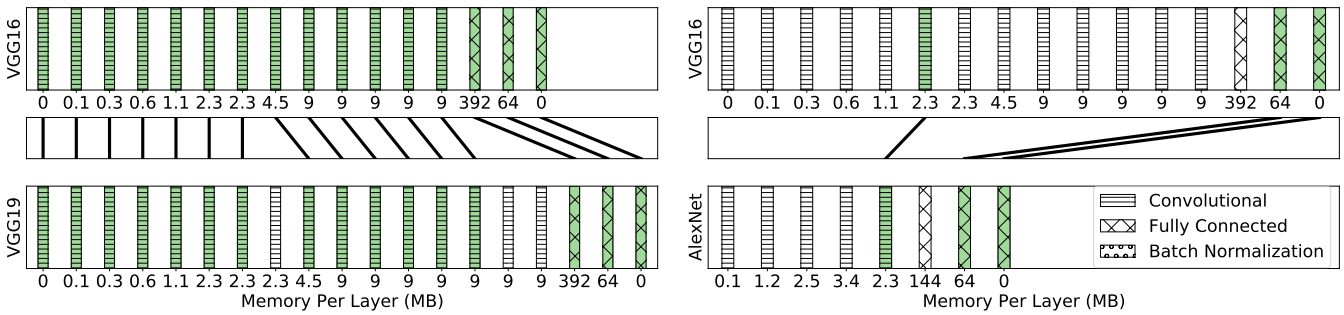


Figure 5: Sharing opportunities between VGG16 and VGG19 (left), and VGG16 and AlexNet (right).

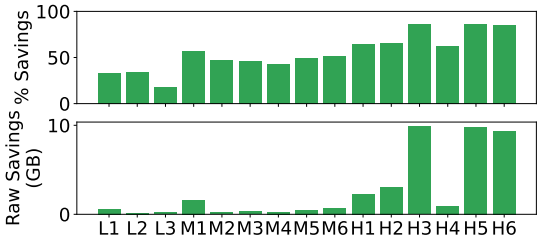


Figure 6: Potential memory savings when all architecturally identical layers are shared across the models in each workload.

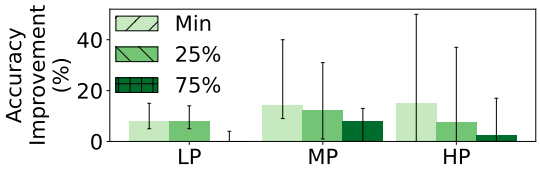


Figure 7: Potential accuracy improvements when sharing all architecturally identical layers. Memory availability is defined in §2, bars list medians, and error bars span min to max.

is, they all ingest pixel representations of raw images, and employ a series of traditional CV operations [22, 64], e.g., a convolutional layer is applying a learned filter to raw pixel values in preparation for downstream processing. Moreover, the target tasks are rooted in identifying and characterizing objects in the scene using low-level CV features such as detected edges and corners [27, 49, 65, 66, 71, 118, 119].

Prior work has capitalized on such similarities for efficient multi-task learning [30, 59, 112] and architecture search [75, 84]. Those efforts aim to reduce computation overheads by sharing "stems" of models, i.e., contiguous layers (and their generated intermediates) starting from the beginning of the models. In contrast, we aim to exploit architectural similarities to reduce memory overheads via merging. As a result, merging only requires layer definitions and weights to be shared, but not generated intermediate values. This distinction is paramount because, as we will discuss in §5.2, memory-heavy layers typically reside towards the end of vision DNNs. Consequently, stem sharing would require almost all model layers to be shared to reap substantial memory savings, which in turn brings unacceptable accuracy drops (§4.2 and §6). Merging, on the other hand, can share only those memory-heavy layers to simultaneously deliver substantial memory savings and preserve result accuracy.

Potential memory savings and accuracy improvements. Figure 6 shows the memory savings from sharing all of the common layers across the models in each of our workloads; this represents an *upper bound* on merging benefits as it disregards the challenge of identifying an acceptable set of weights per shared layer (§4.2). As shown, the memory savings are substantial: per-workload memory usage dropped by 17.9-86.4% relative to no merging, translating to raw savings of 0.2-9.9 GB. Importantly, these savings result in 2 and 4 new workloads fitting entirely (no swapping) on edge boxes with 2 GB and 8 GB of GPU memory (with batch size 1). Similarly, the number of 2 GB edge boxes needed to support each workload drops from 1-9 to 1-4. We further evaluated the resulting impact on end-to-end accuracy by comparing the performance of the Nexus variant from §3.2 when run on workloads with and without (maximal) merging. Models in both cases were ordered in the same way, to maximize the benefits of merging (§5.4). As shown in Figure 7, merging can boost accuracy by up to 50% across our workloads. These benefits are a direct result of lower swapping costs, and the resulting ability to run on 29-61% more frames.

4.2 Challenges

Merging layers for memory reductions requires using shared weights across the models in which those layers appear. However, those shared weights must not result in accuracy violations for any of the models, despite their potentially different architectures/tasks, target objects/videos, etc.; such accuracy drops would forego merging benefits from faster swapping. Concretely, there are two core challenges in practically exploiting the architectural commonalities from §4.1.

Challenge 1: sharing vs. accuracy tension. To maximize memory savings, merging seeks to share as many architecturally identical layers as possible across a workload's models. However, we observe that accuracy degradations steadily grow as the number of shared layers increases. Figure 8 illustrates this trend by sharing different numbers of identical layers across representative pairs of models that vary on the aforementioned properties, e.g., tar-

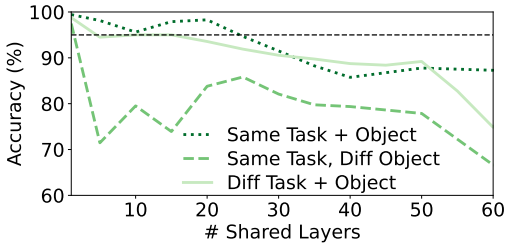


Figure 8: Accuracy after 5 hours of retraining when sharing additional architecturally-identical layers for different model pairs (starting from their origins). Tasks cover detection (Faster RCNN) and classification (ResNet50), and two objects: people, vehicles. Results list the lower per-model accuracies per pair.

get task. These results were obtained when we increase the number of shared layers by moving from start to end in the considered models, but similar trends are observed for other selection strategies (e.g., random) and models.

The reason for this behavior is intuitive: the retraining performed to assess the feasibility of a sharing configuration is *end-to-end* across the considered models. During this process, weights are being tuned for all of the layers in all of the models, with the constraint being that the shared layers each use a unified set of weights. Sharing more layers has three effects: (1) more constraints are being placed on the training, (2) it is harder to find weights for (the shrinking number of) unshared layers that simultaneously accommodate the growing constraints, and (3) learning each model's desired function becomes more difficult as there exist fewer overall parameters to tune [23, 70]. It is for these reasons that isolated merging strategies such as averaging weights across copies of each shared layer (while keeping other layers unchanged) do not suffice; we find that sharing even single layers in this way almost always results in unacceptable accuracy dips.

Digging deeper, the issue stems from non-convex optimization of DNNs, which leads to several equally good global minima [62, 63]. Thus, training even two identical models on the same dataset, and for the same task/object, often results in divergent weights across each layer, despite the resultant models exhibiting similar overall functionality.

Challenge 2: retraining costs. The retraining involved in determining whether a set of layers to share can meet an accuracy target, and if so, the weights to use, can be prohibitively expensive. For instance, each epoch when jointly retraining two Faster RCNN models that detect cars at nearby intersections (i.e., a simple scenario) took ≈35 mins, and different combinations of layer sharing required between 1-10 epochs to converge. These delays grow as more models are considered since training data must reflect the behavior of all of the unmerged models that are involved, e.g., by using the original training datasets for each of those models. Worse, it is difficult to know, a priori, which sharing configurations can meet accuracy targets (and which will not) in a reasonable time frame. For example, the model pairs in Figure 8 have largely different 'breaking points.' The result

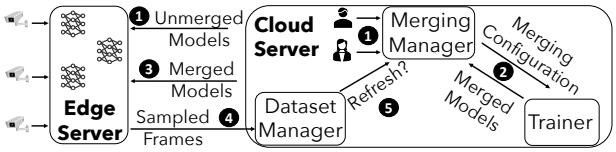


Figure 9: Gemel architecture.

also fails to support the use of intuitive trends to predict the success of sharing configurations: models targeting the same task or object do not exhibit any discernible advantage.

5 Gemel Design

Gemel is an end-to-end system that practically integrates model merging into edge video analytics pipelines by addressing the challenges in §4.2. We first provide an overview of Gemel's operation, and then describe the core observations (and resulting optimizations) that it leverages to enable timely merging without violating accuracy requirements.

5.1 Overview

Figure 9 shows Gemel's cloud merging and edge inference workflows. As in existing pipelines [16,60,71], users register inference tasks (or "queries") at Gemel's cloud component by providing a DNN, and specifying the input video feed(s) to run on as well as the required accuracy for the results. Upon receiving new queries, Gemel bootstraps edge inference by sending unaltered versions of the registered models to the appropriate edge box(es) ❶. When GPU memory is insufficient to house all of those models, edge boxes run the Nexus variant from §3.2 that pipelines inference and model loading to maximize the min per-model throughput.

After initiating edge inference, Gemel's cloud component begins the merging process, during which it *incrementally* searches through the space of potential merging configurations across the registered models, and evaluates the efficacy of each configuration in terms of both its potential memory savings and its ability to meet accuracy requirements ❷. The evaluation of each configuration involves joint retraining and validation of the models participating in merging. Since Gemel's goal is to ensure that the retrained models deliver sufficient accuracy (relative to the originals) on the target feeds, data for these tasks can be obtained in one of two ways: users can supply the data used to train the original models, or Gemel can automatically generate a dataset by running the supplied model (or a high-fidelity one [60, 116]) on sampled frames from the target feed.

At the end of each merging iteration, if the considered configuration was successfully retrained to meet the accuracy targets for all constituent models, Gemel shares the updated merged models with the appropriate edge boxes ❸. New merging results may result in altered edge inference schedules to maximize merging benefits for reducing swapping costs and boosting inference throughput. The iterative merging process for the current workload then continues until (1) the cloud resources dedicated to merging have been ex-

pended, (2) no configurations that can deliver superior memory savings are left to explore, or (3) models with sharing opportunities are either newly registered or deleted by users.

Gemel periodically assesses *data drift* for its merged models. As in prior systems [71, 100], edge servers periodically send sampled frames (and their inference results, if collected) to Gemel's cloud component ❹. These sampled frames are used to augment the datasets considered for retraining merged models, and to track the accuracy of recent results generated at the edge by deployed merged models. For the latter, Gemel runs the original user models on the sampled videos and compares the results to those from the merged models. If accuracy is below the target for any query, Gemel reverts edge inference to use the corresponding original (unmerged) models, and resumes merging and retraining, starting with the previously deployed weights ❺.

Implementation. Gemel uses PyTorch [18] to manage cloud merging and edge inference, and is implemented in ≈3500 LOC. More details are presented in A.1.

5.2 Guiding Observations

Two key empirical observations guide Gemel's approach to tackling the challenges in §4.2. We describe them in turn.

Observation 1: power-law memory distributions. We find that vision DNNs commonly exhibit power-law distributions in terms of memory usage, whereby a few "heavy-hitter" layers account for most of the overall model's memory consumption. Figure 10 illustrates this, showing that for 80% of considered models, 15% of the layers account for 60-91% of memory usage. For example, a single layer in VGG16 is responsible for 392 MB (the entire model is 536 MB) and corresponds to the steep slope around the x=80% mark. Similarly, Tiny YOLOv3 has three layers (around the 38%, 45%, and 65% marks) that together use 35 MB of its total 42 MB.

Heavy-hitter layers come in one of two forms. The first are the convolutional layers at the end of the feature extractor that condense the numerous low-level features extracted by prior layers (e.g., shapes, colors) into higher-level, more abstract features (e.g., eyes, nose). The second are the subsequent fully-connected layer(s) that learn more robust patterns from all possible combinations of those high-level features, e.g., eyes, nose, and fur might each suggest a dog, but the combination is a stronger indicator. Note that models generally include one such fully-connected layer per sub-task, e.g., detectors have one for finding bounding boxes and one for classifying objects. Memory-heavy fully-connected layers are spatially close to one another (within a few layers), and are usually followed by 1-2 cheap fully-connected layers that extract predictions from the final feature vector.

The main exception is ResNet, whose models use residual layers to address accuracy saturation limitations of prior deep models [47]. ResNet models have memory-heavy ResNet blocks (set of convolutional layers) that repeat at varying frequencies, thereby distributing memory more evenly across the models, e.g., ResNet101 and ResNet152 repeat the same ResNet block 23 and 36×, leading to gradual slopes in Figure 10. DenseNet has the same pattern [51].

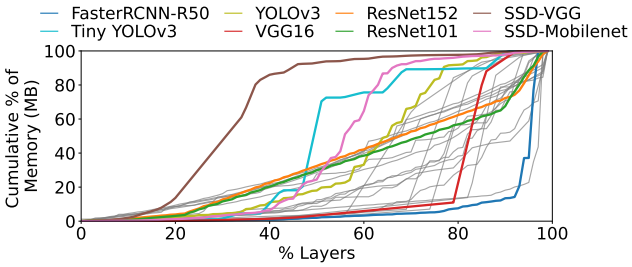


Figure 10: Cumulative memory consumed by each model's layers moving from start to end of the model. §A.4 has full legend.

Figure 10 also shows that heavy-hitter layers most often appear in the latter half of a model's architecture (since both forms involve condensing features from earlier layers), complicating the use of stem sharing for memory savings (§4.2). For example, Faster RCNN's expensive fully-connected layers fall at layers 101 and 104 out of 106, and together account for 76% of total memory. The few cases with heavy-hitters in the middle of a model (between the 20-60% marks) are "single-shot" detectors (SSD-VGG, SSD-Mobilenet, Tiny YOLOv3, YOLOv3) that find bounding boxes and classify objects at once, rather than as disparate subtasks. These models replace the few memory-heavy fully-connected layers (for those subtasks) with many cheaper convolutional layers; doing so extends model lengths and shifts the large jump from memory-heavy *feature extractor* layers to earlier.

These observations have two implications on merging. First, strategies can reap most potential memory benefits by targeting the few heavy-hitter layers in models. Thus, the tension between memory savings and accuracy is far more favorable than that between the number of shared layers and accuracy (Figure 8). Second, strategies should be agnostic to the position of heavy hitters in models, and must support the common case where heavy hitters appear towards the end.

Observation 2: independence of per-layer merging decisions. In DNNs, layers are configured based on input formats, target task, execution time, etc. Hence, a natural assumption is that the ability to share any one layer is dependent on sharing decisions for other layers, e.g., a layer may be shareable if and only if other layers are shared. Prior work has highlighted that inter-layer dependencies primarily arise between neighboring layers, e.g., with transfer learning, performance drops are largest when splitting neighboring layers [112]. Thus, to determine the existence of layer-wise dependencies as it pertains to merging, we focus our analysis on (potential) dependencies between neighboring layers; we also consider other layers via random selection. Using the 25% most memory-heavy layers for each model in our workloads, we test whether accuracy targets are met under different sharing configurations (described in Table 2).

	Only Alone	Only Alternate	Both	Neither
1 Each Side	1.1%	0.0%	97.6%	1.3%
2 Each Side	3.7%	0.0%	95.0%	1.3%
Random	8.5%	0.0%	90.2%	1.3%

Table 2: Sharing a layer alone vs. *alternate* approaches (sharing a layer with one or two neighbors on each side, or with 3 random sets of 1-10 layers). Results are % of runs that meet accuracy targets (aggregated across 80, 90, 95%), and list cases where the layer alone met but an alternate did not, an alternate met but the layer alone did not, both met, and neither met.

As shown, we *never* observe a case where a layer is unable to meet an accuracy target on its own, but it is able to meet the accuracy target when some other layers are also shared (shaded row in Table 2). This is consistent with our finding that sharing more layers leads to larger accuracy degradations (Figure 8) since additional constraints are placed on the weights for those layers, and fewer (unconstrained) non-shared layers exist to help satisfy the constraints. The implication is that layers can be considered independently during merging without harming their potential merging success.

Takeaway. Collectively, these observations motivate an incremental merging process (detailed in §5.3) that attempts to share one new layer at a time, and prioritizes heavy-hitter layers that consume the most memory (and are thus the most fruitful to share). In this manner, memory-heavy layers are considered in the most favorable settings (i.e., with the fewest other shared layers), and each increment only modestly adds to the likelihood of not meeting accuracy targets.

Note. Despite arising across our diverse workloads, these observations are not guarantees. Importantly, violation of these observations only results in merging delays (inefficiencies), but not accuracy breaches; accuracy is explicitly vetted prior to shipping merged models to the edge for inference.

5.3 Merging Heuristic

Gemel begins by enumerating the layers that appear in a workload, and annotating each with a listing of which models the layer appears in (and where) and the total memory it consumes across the workload; we refer to all appearances of a given layer as a 'group.' Gemel then sorts this list in descending order of memory consumption, e.g., a 100 MB layer that appears in 4 models would be earlier than a 120 MB layer that appears 3 times. Thus, memory-heavy groups, or those that would yield the largest memory savings if successfully merged, are towards the start of the list.

Gemel then maintains a running merging configuration, and simultaneously merges and trains layers across models in an *incremental* fashion. To begin, Gemel selects the first group from the sorted list (i.e., the one that consumes the most memory in the workload) and attempts to share it across all of the models in which it appears; this group is added to the running configuration. While a subset of models could be considered instead, Gemel aggressively opts to first try sharing across all models in the group, and then to selectively remove appearances of the layer when the resulting accuracy is insufficient. The reason is that we did not observe any model clustering strategies (e.g., based on task) that identified models consistently unable to share layers.

To retrain and merge the current running configuration, Gemel selects initial weights for the newly added group from a random model that includes that layer. We tried selecting weights from each model (including the one with the highest accuracy) but found no difference in the # of epochs needed to meet accuracy. We also tried default initialization techniques (e.g., Kaiming [48]), which led to lower accuracy. Retraining continues until the merged models each meet their accuracy targets, or a preset time budget elapses (10 epochs by default). If retraining is successful, Gemel adds the next group in the sorted list to the running configuration, and resumes retraining from the weights at the end of the previous iteration. The generated merged models are sent to the edge box and incorporated into edge inference (§5.4).

If retraining is not successful at the end of an iteration, Gemel must decide whether to prune layers from the current group and try again, or to discard the group altogether and move on to the next one in the sorted list. To do this, Gemel follows a strategy that aims to balance fast memory savings and avoidance of unsuccessful training rounds, with priority on the latter since failures can consume 3-10 epochs (each up to 30 min) and provide no new memory savings. Specifically, recall that each time a new group is considered, the number of shared layers in the merging configuration grows by the size of the group. To counter this 'additive increase,' upon unsuccessful retraining, Gemel halves the current group, eliminating half of the layer appearances. If the resulting layer appearances consume more memory than the next group, Gemel considers those layers; else, Gemel removes the current group from the running policy, and moves to the next one. In either case, retraining resumes from the weights at the end of the last successful iteration. We compare against alternate merging heuristics in §6.2.

Accelerating retraining. Each iteration requires Gemel to run retraining over many epochs, and validate the results accuracy-wise. To accelerate training and validation, Gemel takes an adaptive approach. During validation, as per-model accuracy values approach their targets, it is often unnecessary to train further on full epochs of data. Instead, Gemel reduces the training data once the accuracy is within a predefined threshold of the target. Specifically, Gemel reduces the amount of data so it is inversely proportional to the gap in accuracy normalized by the lift since the previous training. Reducing data on such *early success* directly translates to lower training times. Similarly, Gemel detects *early failures* by looking at the validation results and removing models that are not improving at the same pace as the others after some time (3 epochs by default). We empirically observe that early success and early failure detection drastically (28% on average) reduces retraining times.

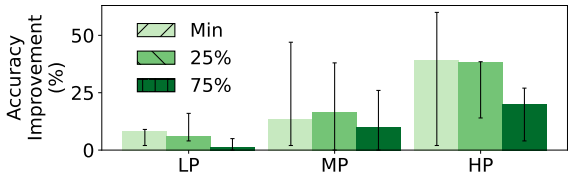


Figure 11: Accuracy improvements with Gemel compared to time/space-sharing alone for different GPU memories (defined in §2). Bars list median workloads, with error bars as min-max.

5.4 Edge Inference

Upon receiving a new set of merged models from Gemel's cloud component, an edge server quickly incorporates those models into its inference schedule. However, to ensure that merging benefits are maximized, the schedule is altered to reduce the amount of data that must be loaded across the anticipated swaps. During the offline profiling Nexus uses to select per-model batch sizes, Gemel estimates per-workload-iteration swapping delays based on per-model computation costs and swapping delays (both influenced by merging). The idea is that, when merging is used, in addition to ordering models to reduce the number of swaps, models that share the most layers should be placed next to one another in the load order. This lowers the cost of each swap by enabling finer-grained swapping, where only those layers in the next model that are not already in GPU memory must be loaded.

More generally, all schedulers will reap merging benefits in the event that Gemel enables a workload to entirely fit on an edge box (without swapping). Additional benefits depend on the specific scheduler. For schedulers that employ a statically-configured load order [81, 94], Gemel can directly modify the schedule as described above to maximize benefits. Other schedulers [39] dynamically select the load order to optimize for a certain metric. Such schedulers typically incorporate model loading times when estimating the efficacy of different orders, and thus would naturally factor in the effects of merging per swap. Note that merging benefits would be considered in the context of meeting the optimization metric(s) rather than minimizing global loading delays (as in Gemel's Nexus variant). Lastly, schedulers that ignore load times in favor of policies such as FIFO [105] or priority scheduling [111] will only see merging-induced reductions in loading costs if merged models are (by chance) neighbors in the order. Note that finer-grained [50, 110] and space-sharing [9, 14, 17, 21] schedulers follow the same principles: shared layers should be adjacent in the load orders for the former, while models with the most shared layers should be placed in the same GPU partition for the latter.

6 Evaluation

We primarily evaluated Gemel across the diverse workloads and settings from §2. Our key findings are:

- Gemel improves per-workload accuracies by 8-39% compared to time/space-sharing strategies alone; these improvements result from Gemel processing 13-44% more frames (while adhering to SLAs).
- Gemel lowers memory needs by 17.5-60.7% (0.2-5.1 GB); savings are 5.9-52.3% more than Mainstream [59] (stem sharing), and within 9.3-29.0% of an optimal that ignores weights (and accuracy drops) when sharing layers.
- More than 70% of Gemel's memory savings are achieved within the first 24-210 minutes of merging+retraining due to its incremental merging heuristic.

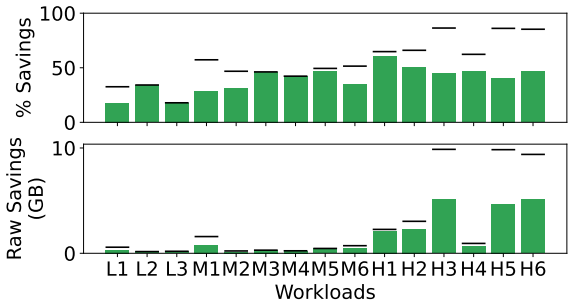


Figure 12: Gemel's per-workload memory savings. Lines above bars show the theoretical optimal savings from Figure 6.

6.1 Overall Performance

End-to-end Accuracy Improvements. We first compare Gemel with time/space-sharing solutions alone, i.e., the Nexus variant running with only unmerged (original) models. Our experiments consider all workloads and resource settings from §2, a per-frame processing SLA of 100 ms, and an accuracy target of 95%; trends hold for other accuracy targets and SLAs, which we consider in §6.2.

Figure 11 presents our results, showing that Gemel improves accuracy by 8.0%, 13.5%, and 39.1% for the median LP, MP, and HP workloads, respectively, when the edge box GPU's memory is just enough to load and run the largest model in each workload, i.e., the *min* setting. The origin of these benefits is Gemel's ability to reduce the time blocked on swapping delays by 17.9-84.0%, which enables processing on 13-44% more frames than without merging.

Our results highlight two other points. First, Gemel's benefits are highest for workloads that are most significantly bottlenecked by memory restrictions (and thus loading costs). For instance, workloads HP1 and LP1 exhibit largely different memory vs. computation profiles: loading costs are 66% of computation costs in the former, but only 15% in the latter. Accordingly, Gemel's accuracy wins across the available memory settings are 11-60% and 5-16% for workloads HP1 and LP1. Second, Figure 11 shows that, as expected, Gemel's benefits per workload decrease as the available GPU memory grows, e.g., accuracy improvements drop to 17.5% and 10.2% for the median MP workload when GPU memory grows to 50% and 75% of the total workload memory needs. The reason is straightforward: larger GPU memory yields fewer required swaps without merging.

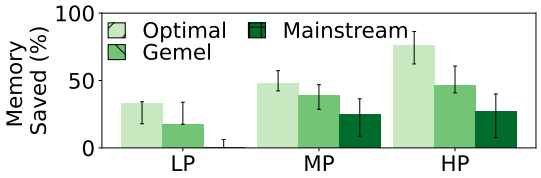


Figure 13: Memory savings with Gemel, an optimal that ignores accuracy, and Mainstream [59]. Bars list the median workload per class, with error bars spanning min to max.

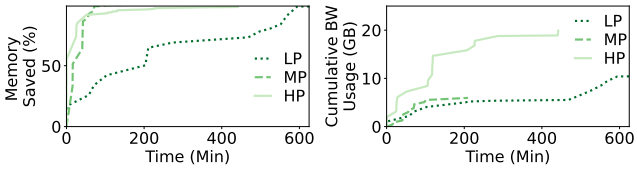


Figure 14: Gemel's memory savings (left) and cloud-to-edge bandwidth usage (right) over time during incremental merging. Results show the median workload per class.

Memory Reductions. Figure 12 lists the memory reductions that Gemel delivers for each considered workload by sharing model layers and the associated weights, i.e., parameter reductions. We note that reported values here are based on Gemel's final merging results and an accuracy target of 95%; we analyze the incremental nature of Gemel's merging heuristic in §6.2. As shown, parameter reductions are 17.5-33.9% for LP workloads, 28.6-46.9% for MP workloads, and 40.9-60.7% for HP workloads; the corresponding raw memory savings are 0.2-0.3 GB, 0.2-0.8 GB, and 0.7-5.1 GB, respectively. When analyzed in terms of overall memory usage during inference (i.e., including the parameters, inference framework, and intermediate data generated during model execution), reductions are 4.5-48.1% across the workloads. Wins are generally higher for workloads with larger parameter reductions, with the exception of Workloads LP1 and LP3 (reductions of 6.3% and 4.5%) whose intermediates are particularly large relative to the parameters.

To better contextualize the above memory savings, we compare Gemel with two alternatives. First, we consider a theoretical optimal (*Optimal*) that shares all layers that are architecturally identical across a workload's models, without considering accuracy (and the need to find shared weights for those layers). Thus, Optimal represents an *upper bound* on Gemel's potential memory savings. Second, we compare with *Mainstream* [59], a recent stem-sharing approach. To run Mainstream, we trained each model in our workloads several times, each time starting with pre-trained weights (based on ImageNet [90]) and freezing up to different points, e.g., freeze up to layer 10, freeze up to layer 15, etc. We selected the configuration for each model that kept the most layers frozen while meeting the accuracy target (95% relative to no freezing). Then, within each workload, we merged all layers that were shared across the frozen layer set of the constituent models (note that these layers have identical weights) and recorded the resultant memory savings.

Figure 13 shows our results, from which we draw two conclusions. First, Gemel's memory savings are within 9.3%, 15.0%, and 29.0% of Optimal for the median LP, MP, and HP workloads. Second, Gemel's memory reductions are 5.9-52.3% larger than Mainstream's across all workloads. This is a direct consequence of Gemel's prioritization of memory-heavy layers that routinely appear towards the end of models (§5.2). By requiring shared stems from the start of the models, Mainstream would have to share all layers up to the memory-heavy ones; we find that sharing nearly-entire models is rarely possible while meeting accuracy targets (Figure 8). The high variance in Mainstream's results are due to the fact that different models drop in accuracy at different rates when more layers are frozen. Classifiers drop relatively slowly (savings up to 70.1%), while detectors are a harder task with faster accuracy drops (Mainstream was unable to share many layers, with savings as low as 1.0%).

6.2 Analyzing Gemel

Incremental memory savings. Key to Gemel's practicality are its efficient merging heuristic and retraining optimizations that aim to reap memory savings early in the process; indeed, this is important not only to reap accuracy-friendly memory wins quickly, but also to quickly respond to workload changes. As shown in Figure 14 (left), 73% of Gemel's achieved memory savings for the median HP workload are realized within the first 24 minutes of merging. Similarly, 86% and 64% of the total memory savings are achieved in the first 42 and 210 minutes of merging for median MP and LP workloads, respectively.

Network bandwidth usage. After each successful merging iteration, Gemel ships weights to edge servers for all updated models. As shown in Figure 14 (right), cumulative bandwidth usage during merging is 6.0-19.4 GB for the three workloads. Importantly, bandwidth consumption largely grows after substantial memory savings are already reaped. For example, for the median MP workload, 86% of memory savings are achieved in 42 minutes, while only 2.1 GB (of the total 6.0 GB) of bandwidth is used during that time. The reason is that later merging iterations explore the larger number of lower-memory layers. Thus, Gemel can often deliver large memory savings even in constrained settings with bandwidth caps. Note that shipping weights uses cloud-to-edge (not precious edge-to-cloud) bandwidth.

Micro-benchmarks. We profile the time spent in each of Gemel's components. Training delays are configurable (Figure 14), but dominate cloud merging, with the remaining <2% of time spent on identifying shareable layers (0.7-1.4s per workload) and serializing/saving weights from successful training (9.1-19.5s per round). The majority of time spent at the edge steadily shifts from model loading to inference as Gemel's incremental merging results stream in; at the median, time spent blocked reduces from 32.8%, 48.3%, and 52.0% to 22.1%, 34.6%, and 27.9% for the LP, MP, and HP

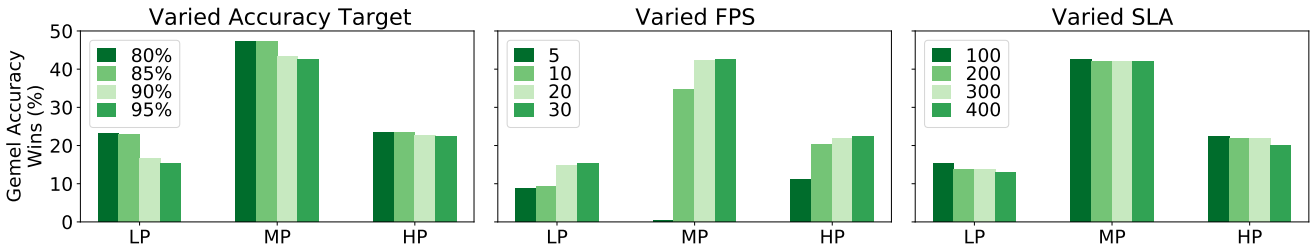


Figure 15: Gemel's accuracy wins (compared to time/space-sharing alone) with varied accuracy targets, FPS, and SLAs.

workloads respectively. Applying results takes <.15s and is not blocking.

Varying accuracy, FPS, and SLA. To evaluate the impact of each parameter, we conducted experiments using one randomly selected workload from each class. In each experiment, we only vary one parameter, while keeping the other two at the fixed values from above (95%, 30 FPS, 100 ms).

Figure 15 presents our results, which exhibit three trends. First, Gemel's accuracy wins over time/space-sharing alone grow (by 1.1-7.8% for the three workloads) as accuracy targets drop (from 95% to 80%). This is because certain layers failed to meet 95% during retraining, but did meet a lower accuracy target. Second, Gemel's accuracy wins drop as input video frame rates (FPS) drop, e.g., from 6.2-42% across the workloads when FPS drops from 30 fps to 5 fps. The reason is that lower FPS values reduce the amount of inference in any time window (assuming a fixed SLA), which in turn adds tolerance to high loading delays. Third, Gemel's benefits grow as SLAs become stricter: accuracy wins for the three workloads rise by 0.4-2.3% when SLA drops from 400 to 100 ms. This is because tighter SLAs imply more skipped frames for a given swapping delay.

Comparison to other merging heuristics. We consider variants that differ from Gemel in one of two ways: they choose layers to merge in a different order or they merge a different number of layers at a time. We describe the variants of each type below, along with the corresponding results. Our experiments use all workloads from §2, and we report memory saved over time. Figure 16 shows results for two representative workloads (HP3, MP2); the remaining workload results are in §A.4. In summary, no variant consistently outperforms Gemel, and the degradations (in saved memory or merging delays) that each brings to certain workloads (from being overly aggressive or cautious) are substantial.

Rather than merging layers in descending order of memory usage (irrespective of position) as Gemel does, the variants we consider start by merging the models' earliest layers (*Earliest*), latest layers (*Latest*), and three random layer orderings (*Random*). Across all workloads, these heuristics all resulted in significantly lower memory savings. Among the three, *Latest* performed the best (median of 13.5% of Gemel's savings), as memory-heavy layers often appear later in a model (but not necessarily the end). For the same reason, *Earliest* performed the worst (0.2% of Gemel's savings). *Random*'s performance varied dramatically (0.2% - 72.9%, median of 5.7% of Gemel's savings) based on whether a memory-heavy layer was selected.

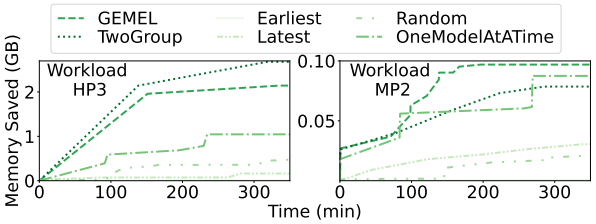


Figure 16: Comparing variants of Gemel's merging heuristic on two representative workloads.

We consider two variants to Gemel's approach of adding one layer group at a time across all models that layer appears in. First, *TwoGroup* more aggressively adds two groups at a time. This can result in faster memory savings than Gemel (3/15 workloads, including Figure 16 (left)), but most often (8/15 workloads) misses accuracy targets and results in substantial slowdowns (78 min longer to max savings for the median workload). The reason is that, on failure, *TwoGroup* restarts training with 1 group, adding long delay without memory savings, e.g., x=75-220 min in Figure 16 (right). Second, *OneModelAtATime* less aggressively shares the selected group's layer iteratively across the models it appears in. This reaches within 5% of Gemel's memory savings in 8/15 workloads, but is often unnecessarily slow, e.g., in Figure 16 (left), Gemel successfully considers 5 models at once, while *OneModelAtATime* individually adds models (some of which fail) leading to the flat stretch from 0-91 min.

6.3 Generalization Study

We evaluate Gemel on over 850 more workloads that extend our main ones by adding: (1) new scene types and the objects they bring (e.g., bags, hats, and people at a beach, boats in a canal), and (2) new models, including more variants in the same families (e.g., ResNet, VGG), and entirely new architectures (e.g., GoogLeNet [101], DenseNet [51]). In total, our analysis involves 17 videos (8 scene types), 13 objects, and 16 models; the extended version [82] lists the values.

Constructing workloads. Each query in a workload is parameterized by a set of knobs: *camera feed* (and corresponding *scene* type), *model*, and *object of interest*. To study the

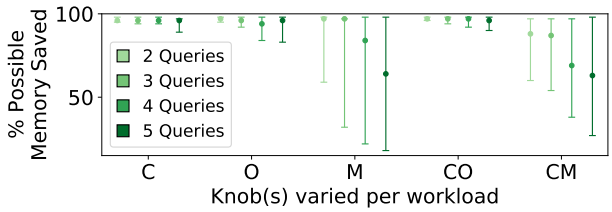


Figure 17: Memory savings across subset of generalization workloads, organized by workload size (color) and knobs varied (Camera, Object, Model). Distributions show median and 25-75%ile; accuracy target was 95%. Figure 22 has full graph.

impact of varying each knob (or combination of knobs) on Gemel's merging, we construct workloads as follows. For each set of target knobs to vary, we start with a random query and incrementally add new queries that only vary values for the target knobs to generate workloads with 2-5 queries each. We did this up to 30 times each for all target knob sets (as their values permit), excluding only (1) target knob sets that vary the *scene* but not *camera* knob, (2) queries for an *object* that never appears in a given *camera* feed, and (3) workloads with no possible memory sharing opportunities.

Findings. As shown in Figure 17, Gemel's memory savings are high for 2-query workloads (89-98% of optimal at medians), but steadily degrade as workloads grow. This is expected as increasing workload size is (by design, and unrealistically) increasing heterogeneity in this experiment. The nature of degradation depends on the knob(s) being varied. For all combinations of {*camera, object, scene*}, degradations are mild moving from 2- to 5-query workloads (0-8%), showing Gemel's robustness to variations on those properties. Since *model* is constant in these cases, degradations are because the same set of shareable layers must support more diverse scenarios (making it harder to find shared weights).

The *Model* knob (alone or with other knobs) presents a different picture, with larger drops in median memory savings (2-33%) and broader distributions. We can decompose this into two aspects as workload sizes increase:

- Previously-shared layers appear in the new model: the effect on memory savings heavily depends on where the shared layer appears in the new model; recall that layers can appear in different positions (and thus, serve different roles) across models (Figure 19). Cases where the new model introduces drastically different positions for shared layers (e.g., ResNet variants) account for the low-end of the resultant distributions, while memory savings largely persist when positions of shared layer(s) are similar in the new model (e.g., merging across VGG variants).
- New layers are shareable with the new model: the extra sharing opportunities increase potential savings, but are more challenging to realize as they reduce the number of non-shared layers whose weights help compensate for the constraints from sharing (§4.2).

7 Additional Related Work

Certain systems reuse model components [91], most relatedly via stem sharing for compute savings [59] or sharing operators with identical weights anywhere in models [68]; in contrast, Gemel targets memory savings, and enables sharing architecturally-identical layers anywhere in models even if they have different weights. Layer sharing in multi-task learning is often studied in the context of transfer learning, where models for a task with insufficient data leverage the dataset of a related task [30,99,103]; Gemel considers multiple sets of pretrained weights for sharing, each with different goals (e.g., detection vs. classification, different objects).

Other platforms optimize model serving either by tuning video analytics-specific knobs to lower compute footprints [29,35,49,55,60,61,87,109,116,117], or by identifying lightweight variants of individual models that match specific hardware resources [45, 89]; Gemel focuses on memory (not compute) bottlenecks, and optimizes across models. Some frameworks reuse results across frames [31,33,42,67,71], reducing frame rates for inference and alleviating the impact of model loading delays. Gemel provides benefits at lower FPS (§6.2), and also can alleviate memory pressure across spatially correlated feeds that exhibit limited reuse opportunities at the same time (§3.2).

There exist training optimizations that trade off memory usage for computation overheads [83, 86, 93]; we eschew such techniques given the holistic constraint on compute resources that edge boxes face (§1). Finally, another body of work develops metrics to quantify how similar models will behave [41, 58, 72]. While Gemel does not consider model similarity metrics in its heuristic (we quantitatively observe that 'model similarity' is not reflected in layer merging potential), we leave it to future work to explore the relationship between 'model similarity' and 'layer similarity' in improving Gemel's prediction of layer merging potential.

8 Conclusion

Model merging is a new memory management technique that exploits architectural similarities across vision DNNs by sharing their common layers (including parameters but not intermediates). Gemel efficiently carries out model merging by quickly finding and retraining accuracy-preserving layer sharing configurations, and scheduling edge inference to maximize merging benefits (8-39% accuracy boosts).

Acknowledgements. We thank Ramesh Govindan and Jennifer Rexford for their valuable feedback on earlier drafts of the paper. We thank our shepherd, Wenjun Hu, and the anonymous NSDI reviewers for their constructive comments. This work was supported in part by a Sloan Research Fellowship, research grants from Cisco, ONR grant N00014-18-1-2037, and NSF CNS grants 2152313, 2153449, 2147909, 2140552, 1703598, 1763172, 1907352, 2007737, 2006437, 2128653, and 2106838.

References

[1] Absolutely everywhere in beijing is now covered by police video surveillance. https://qz.com/518874/.

[2] Are we ready for ai-powered security cameras? https://thenewstack.io/are-we-ready-for-ai-powered-security-cameras/.

[3] AWS Outposts. https://aws.amazon.com/outposts/.

[4] Azure Stack Edge. https://azure.microsoft.com/en-us/products/azure-stack/edge/.

[5] British transport police: Cctv. http://www.btp.police.uk/advice_and_information/safety_on_and_near_the_railway/cctv.aspx.

[6] Can 30,000 cameras help solve chicago's crime problem? https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html.

[7] Edge computing at chick-fil-a. https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2.

[8] NVIDIA Jetson: The AI platform for edge computing. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/.

[9] NVIDIA Multi-Instance GPU . https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[10] Paris hospitals to get 1,500 cctv cameras to combat violence against staff. https://bit.ly/2OYiBz2.

[11] Powering the edge with ai in an iot world. https://www.forbes.com/sites/forbestechcouncil/2020/04/06/powering-the-edge-with-ai-in-an-iot-world/.

[12] Video analytics applications in retail - beyond security. https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html/.

[13] The vision zero initiative. http://www.visionzeroinitiative.com/.

[14] Cuda multi-process service, April 2021.

[15] Live Video Analytics with Microsoft Rocket for reducing edge compute costs, May 2021.

[16] Microsoft rocket video analytics platform, April 2021.

[17] NVIDIA TensorRT, April 2021.

[18] Pytorch, April 2021.

[19] Pytorch-yolov3. https://github.com/eriklindernoren/PyTorch-YOLOv3, 2021.

[20] Traffic Video Analytics – Case Study Report, May 2021.

[21] R. B. , Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, V. Bahl, and I. Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *USENIX NSDI*, April 2022.

[22] M. Alam, M. Samad, L. Vidyaratne, A. Glandon, and K. Iftekharuddin. Survey on deep neural networks in speech and vision systems. *Neurocomputing*, 417:302–321, 2020.

[23] Z. Allen-Zhu, Y. Li, and Y. Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. *CoRR*, abs/1811.04918, 2018.

[24] Amazon. Rekognition. https://aws.amazon.com/rekognition/.

[25] G. Ananthanarayanan, V. Bahl, L. Cox, A. Crown, S. Nogbahi, and Y. Shu. Video analytics - killer app for edge computing. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 695–696, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, Nov. 2020.

[27] S. Brutzer, B. Hoferlin, and G. Heidemann. Evaluation of background subtraction techniques for video surveillance. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pages 1937–1944, Washington, DC, USA, 2011. IEEE Computer Society.

[28] Z. Cai, M. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 3361–3369, Washington, DC, USA, 2015. IEEE Computer Society.

[29] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor. Scaling video analytics on constrained edge nodes. In *2nd SysML Conference*, 2019.

[30] R. Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[31] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[32] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. OSDI, 2014.

[33] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, Mar. 2017. USENIX Association.

[34] S. R. E. Datondji, Y. Dupuis, P. Subirats, and P. Vasseur. A survey of vision-based traffic monitoring of road intersections. *Trans. Intell. Transport. Sys.*, 17(10):2681–2698, Oct. 2016.

[35] K. Du, A. Pervaiz, X. Yuan, A. Chowdhery, Q. Zhang, H. Hoffmann, and J. Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 557–570, New York, NY, USA, 2020. Association for Computing Machinery.

[36] M. Everingham, L. Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010.

[37] Google. Google edge network. https://peering.google.com/#/infrastructure, 2016.

[38] Google. Cloud vision api. https://cloud.google.com/vision, 2021.

[39] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, Nov. 2020.

[40] P. Guo, B. Hu, and W. Hu. Mistify: Automating DNN model porting for on-device inference at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 705–719. USENIX Association, Apr. 2021.

[41] P. Guo, B. Hu, and W. Hu. Sommelier: Curating dnn models for the masses. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1876–1890, 2022.

[42] P. Guo and W. Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. *SIGPLAN Not.*, 53(2):271–284, mar 2018.

[43] HAILO. Edge AI Box. https://hailo.ai/reference-platform/edge-ai-box/, 2021.

[44] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 129–143, New York, NY, USA, 2016. ACM.

[45] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 123–136, New York, NY, USA, 2016. Association for Computing Machinery.

[46] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[47] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[48] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[49] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, Oct. 2018. USENIX Association.

[50] C.-C. Huang, G. Jin, and J. Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[51] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks, 2016.

[52] J. Hui. Object detection: speed and accuracy comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet and YOLOv3). https://jonathan-hui.medium.com/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359, 2018.

[53] C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131, Oct 2018.

[54] IBM. Maximo remote monitoring. https://www.ibm.com/products/maximo/remote-monitoring, 2021.

[55] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, V. Bahl, and J. Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing (SEC 2020)*, November 2020.

[56] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[57] M. Jeon, S. Venkataraman, J. Qian, A. Phanishayee, W. Xiao, and F. Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research*, 2018.

[58] H. Jia, H. Chen, J. Guan, A. S. Shamsabadi, and N. Papernot. A zest of LIME: Towards architecture-independent model distances. In *International Conference on Learning Representations*, 2022.

[59] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, Boston, MA, July 2018. USENIX Association.

[60] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 253–266, New York, NY, USA, 2018. Association for Computing Machinery.

[61] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, Aug. 2017.

[62] K. Kawaguchi, J. Huang, and L. P. Kaelbling. Every local minimum value is the global minimum value of induced model in nonconvex machine learning. *Neural Computation*, 31(12):2293–2323, Dec 2019.

[63] K. Kawaguchi and L. P. Kaelbling. Elimination of all bad local minima in deep learning. *CoRR*, abs/1901.00279, 2019.

[64] S. H. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah. Transformers in vision: A survey. *CoRR*, abs/2101.01169, 2021.

[65] H. Kim, S. Leutenegger, and A. J. Davison. Real-time 3D reconstruction and 6-DoF tracking with an event camera. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI*, pages 349–364, 2016.

[66] B. Kueng, E. Mueggler, G. Gallego, and D. Scaramuzza. Low-latency visual odometry using event-based feature tracks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 16–23, Oct 2016.

[67] A. Kumar, A. Balasubramanian, S. Venkataraman, and A. Akella. Accelerating deep learning inference via freezing. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[68] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, Oct. 2018. USENIX Association.

[69] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, June 2015.

[70] Y. Li and Y. Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 8168–8177, Red Hook, NY, USA, 2018. Curran Associates Inc.

[71] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 359–376, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Y. Li, Z. Zhang, B. Liu, Z. Yang, and Y. Liu. ModelDiff: testing-based DNN similarity comparison for model reuse detection. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2021.

[73] Z. Li, Y. Shu, G. Ananthanarayanan, L. Shangguan, K. Jamieson, and V. Bahl. Spider: A multihop millimeter-wave network for live video analytics. In *ACM/IEEE Symposium on Edge Computing*. ACM/IEEE, December 2021.

[74] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, July 2017.

[75] H. Liu, K. Simonyan, and Y. Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[76] X. Liu, P. Ghosh, O. Ulutan, B. S. Manjunath, K. Chan, and R. Govindan. Caesar: Cross-camera complex activity recognition. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 232–244. Association for Computing Machinery, 2019.

[77] Microsoft. Enabling Data Residency and Data Protection in Microsoft Azure Regions. https://azure.microsoft.com/en-us/resources/achieving-compliant-data-residency-and-security-with-azure/, 2021.

[78] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Comp. and Comm.*, 23(4):11–20, May 2020.

[79] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS*, 2010.

[80] OfCom. Residential landline and fixed broadband services. https://www.ofcom.org.uk/__data/assets/pdf_file/0015/113640/landline-broadband.pdf, 2017.

[81] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ml serving, 2017.

[82] A. Padmanabhan, N. Agarwal, A. Iyer, G. Ananthanarayanan, Y. Shu, N. Karianakis, G. H. Xu, and R. Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge, 2022.

[83] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 891–905. Association for Computing Machinery, 2020.

[84] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018.

[85] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-preserving video analytics as a cloud service. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1039–1056. USENIX Association, Aug. 2020.

[86] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. Zero-infinity: Breaking the GPU memory wall for extreme scale deep learning. *CoRR*, abs/2104.07857, 2021.

[87] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1421–1429, 2018.

[88] H. Rebecq, T. Horstschaefer, and D. Scaramuzza. Real-time visual-inertial odometry for event cameras using keyframe-based nonlinear optimization. In *British Machine Vision Conference 2017, BMVC 2017, London, UK, September 4-7, 2017*, 2017.

[89] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[90] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[91] S. S. Sarwar, A. Ankit, and K. Roy. Incremental learning in deep convolutional neural networks using partial network sharing. *IEEE Access*, 8:4615–4628, 2019.

[92] J. Sevilla, P. Villalobos, and J. Cerón. Parameter counts in Machine Learning. https://www.lesswrong.com/posts/GzoWcYibWYwJva8aL/parameter-counts-in-machine-learning, 2021.

[93] A. Shah, C. Wu, J. Mohan, V. Chidambaram, and P. Krähenbühl. Memory optimization for deep networks. *CoRR*, abs/2010.14501, 2020.

[94] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[95] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[96] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[97] Sony. REA-C1000 Edge Analytics Appliance. https://pro.sony/ue_US/products/ptz-cameras/rea-c1000-edge-analytics-appliance, 2021.

[98] F. Sultana, A. Sufian, and P. Dutta. Evolution of image segmentation using deep convolutional neural network: A survey. *Knowledge-Based Systems*, 201-202:106062, 2020.

[99] X. Sun, R. Panda, R. Feris, and K. Saenko. Adashare: Learning what to share for efficient deep multi-task learning. *arXiv preprint arXiv:1911.12423*, 2019.

[100] A. Suprem, J. Arulraj, C. Pu, and J. Ferreira. Odin: Automated drift detection and recovery in video analytics. *Proc. VLDB Endow.*, 13(12):2453–2465, July 2020.

[101] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions, 2014.

[102] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision, 2015.

[103] S. Vandenhende, S. Georgoulis, B. De Brabandere, and L. Van Gool. Branched multi-task networks: deciding what layers to share. *arXiv preprint arXiv:1904.02920*, 2019.

[104] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *CCR*, 44(5):27–32, Oct. 2014.

[105] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[106] A. R. Vidal, H. Rebecq, T. Horstschaefer, and D. Scaramuzza. Ultimate slam? combining events, images, and IMU for robust visual SLAM in HDR and high-speed scenarios. *IEEE Robotics and Automation Letters*, 3(2):994–1001, 2018.

[107] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, and M. Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 152–165, New York, NY, USA, 2019. Association for Computing Machinery.

[108] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia. Characterizing deep learning training workloads on alibaba-pai, 2019.

[109] Y. Wang, W. Wang, J. Zhang, J. Jiang, and K. Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[110] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[111] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *JSSPP*, 2003.

[112] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3320–3328, Cambridge, MA, USA, 2014. MIT Press.

[113] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[114] A. R. Zamani, M. Zou, J. Diaz-Montes, I. Petri, O. Rana, A. Anjum, and M. Parashar. Deadline constrained video analysis via in-transit computational environments. *IEEE Transactions on Services Computing*, 13(1):59–72, 2020.

[115] X. Zeng, B. Fang, H. Shen, and M. Zhang. Distream: Scaling live video analytics with workload-adaptive distributed edge intelligence. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 409–421, New York, NY, USA, 2020. Association for Computing Machinery.

[116] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, Mar. 2017. USENIX Association.

[117] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 426–438, New York, NY, USA, 2015. ACM.

[118] A. Z. Zhu, N. Atanasov, and K. Daniilidis. Event-based feature tracking with probabilistic data association. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*, pages 4465–4470, 2017.

[119] A. Z. Zhu, N. Atanasov, and K. Daniilidis. Event-based visual inertial odometry. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5816–5824, July 2017.

A Appendix

A.1 Implementation Details

Gemel's main components are training models at the cloud server and running the scheduler at the edge. During training, a single optimizer manages the weights across all considered models; the optimizer holds a single copy of weights for each layer that is shared across the models. Aside from this, Gemel's training process mirrors classic multi-task training [30]: it forms a collective pool of an equal number of data samples from all models and randomly selects batches from this pool. Samples are run through their respective models, each model calculates its loss individually, and losses are summed over all models. In this way, layers that are shared are updated by the concurrent training of multiple models within a single batch.

The Nexus-variant scheduler chooses when to load and evict models as described in §5.4. To load a model into GPU memory, the scheduler simply calls ".cuda()" on that model's PyTorch object. PyTorch automatically only loads layer weights not already in GPU memory. However, when evicting a model, PyTorch, by default, removes all of the layers' weights from GPU memory. This poses a problem if some of those weights are needed by models still in GPU memory (i.e., they are shared). To avoid this, the scheduler: (1) maintains a running list of shared layers that are needed by models currently in GPU memory or next in line to be loaded, and (2) when a model needs to be evicted, only evicts weights corresponding to layers not in the list. Overall, Gemel is implemented in ≈3500 LOC: 500 for finding shared layers and sharing them according to the heuristic, 2500 for dataset management and retraining, and 500 for scheduling models at the edge.

A.2 Generalization Workload Query Knobs

Knob	Values
Object	Truck, Person, Bus, Boat, Shoe, Skateboard, Car, Hat, Backpack, Wine Glass, Traffic Light, Parking Meter, Surfboard
Camera	A0, A1, A2, A3, B0, B1, B2, B3, B4, B5, B6, Restaurant, Mall, Beach, Canal, Parking Lot, Street
Model	SSD-VGG, AlexNet, YOLOv3, Tiny-YOLOv3, DenseNet, SqueezeNet, GoogLeNet, ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-152, VGG-11, VGG-13, VGG-16, VGG-19
Scene	CityA Traffic, CityB Traffic, Restaurant, Beach, Mall, Canal, Parking Lot, Street

Table 3: Knob values considered in generalization study.

A.3 Workload Memory Settings

Workload	L1	L2	L3
Min	4.50	1.45	4.50
50%	5.12	1.59	4.72
75%	5.43	1.66	4.83

Table 4: Edge box memory settings for LP workloads (in GB).

Workload	M1	M2	M3	M4	M5	M6
Min	3.35	1.45	1.32	1.32	1.45	3.35
50%	4.56	1.62	1.55	1.45	1.83	3.77
75%	5.16	1.70	1.65	1.52	2.02	3.99

Table 5: Edge box memory settings for MP workloads (in GB).

Workload	H1	H2	H3	H4	H5	H6
Min	3.35	4.50	4.50	1.45	4.50	4.50
50%	4.87	6.60	10.25	2.17	10.41	10.26
75%	5.63	7.66	13.13	2.53	13.36	13.14

Table 6: Edge box memory settings for HP workloads (in GB).

A.4 Additional Figures

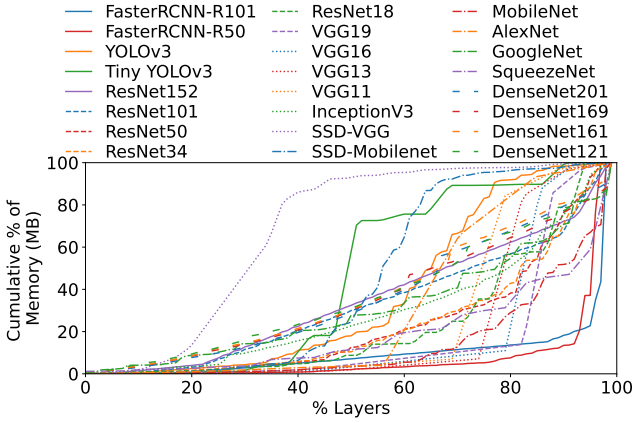


Figure 18: Extended version of Figure 10. Cumulative memory consumed by each model's layer groups moving from start to end of the model.

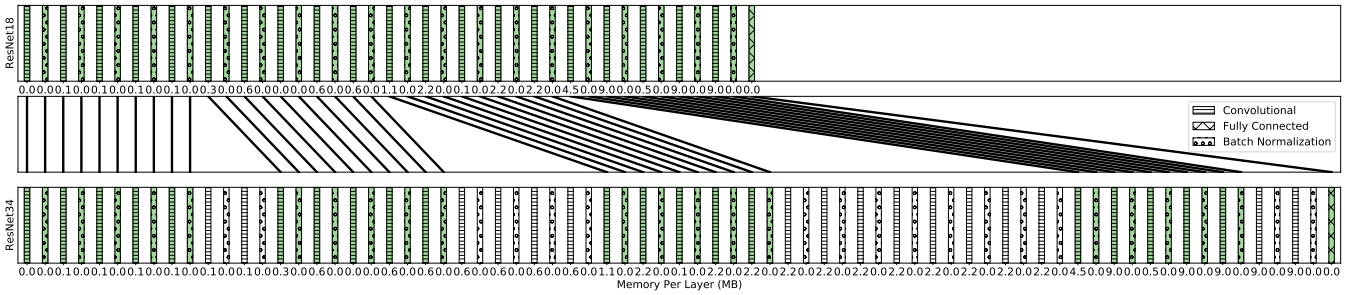


Figure 19: ResNet18 and ResNet34 are variants within the ResNet model family [47]. They share 41/73 layers (20 convolutional, 1 fully-connected and 20 batch normalization).

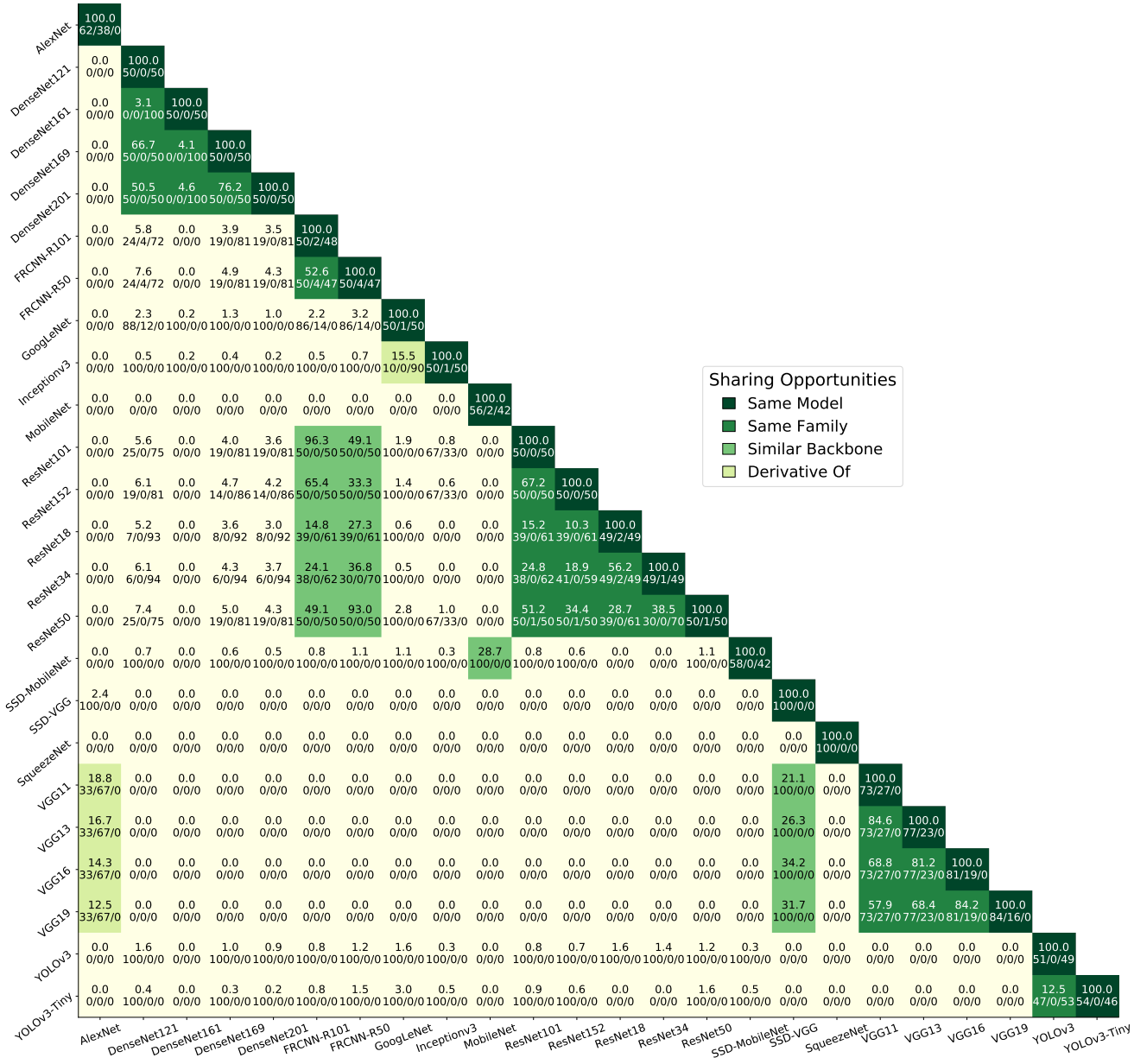


Figure 20: Extended version of Figure 4. For each unique pair of models, we show the percentage of architecturally identical layers and of those layers, the percent breakdown across layer types (%Convolutional / %Linear / %BatchNorm).

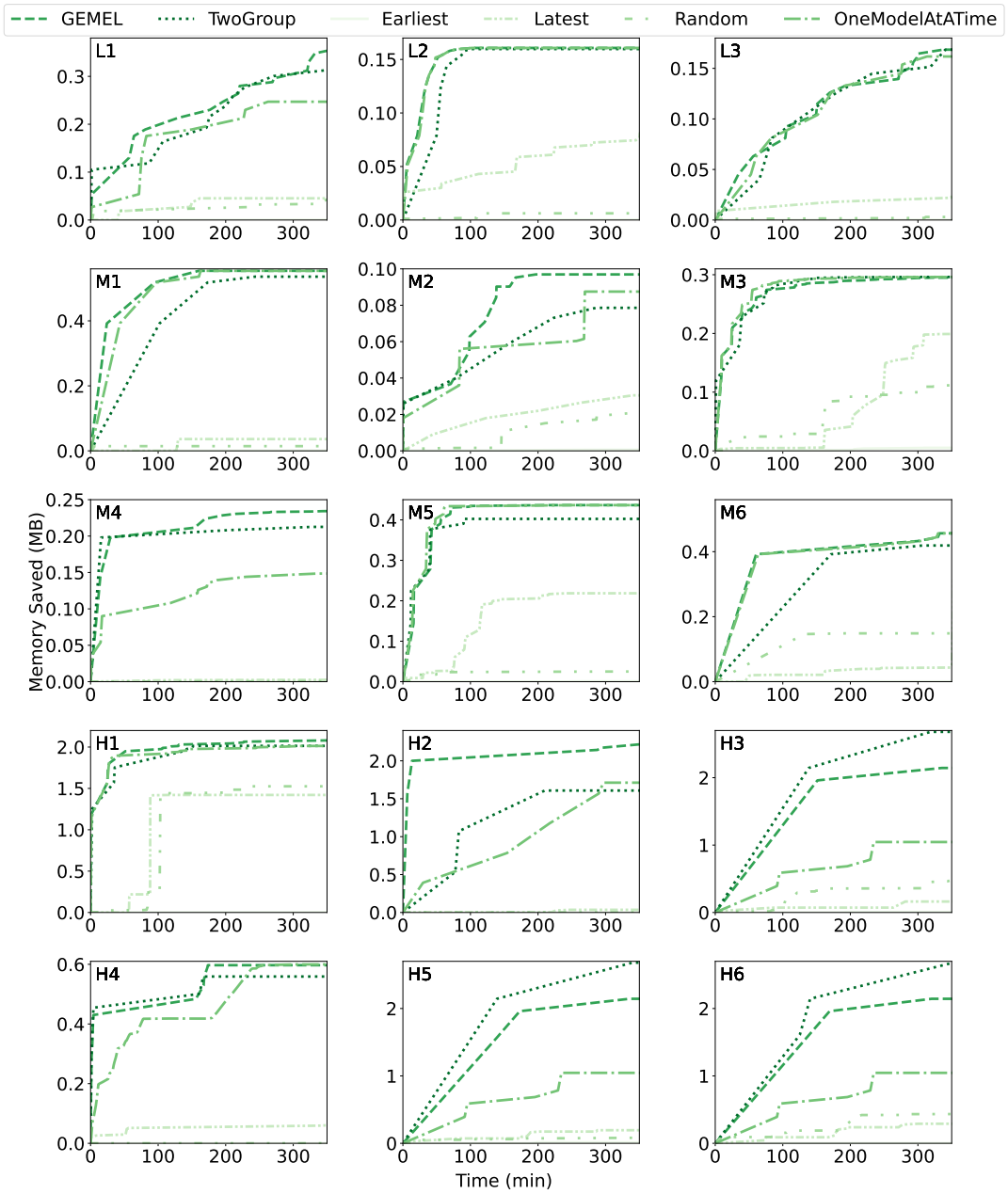


Figure 21: Complete version of Figure 16. Comparison of Gemel with other merging heuristics.

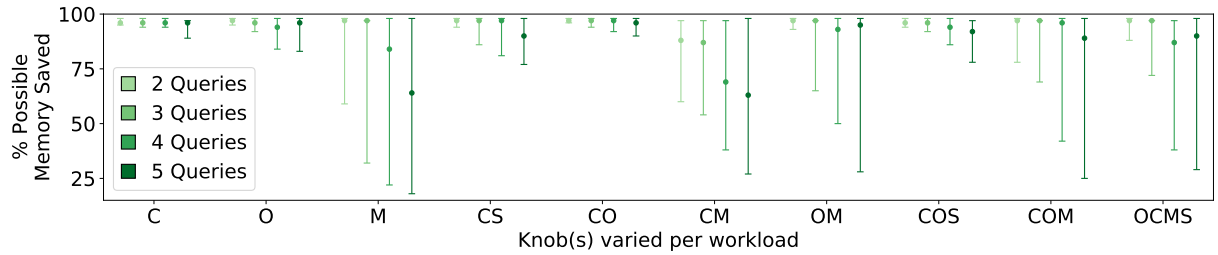


Figure 22: Extended version of Figure 17. Memory savings across 872 workloads, organized by workload size (color) and knobs varied (x-axis). We plot the median of each distribution (error bars spanning 25-75P). Knobs are labeled as follows: C:Camera, O:Object, M:Model, S:Scene.

Fast, Approximate Vector Queries on Very Large Unstructured Datasets

Zili Zhang* Chao Jin* Linpeng Tang† Xuanzhe Liu* Xin Jin*

*Peking University †Moqi

Abstract

The breakthroughs in deep learning enable unstructured data to be represented as high-dimensional feature vectors for serving a wide range of applications. Processing vector queries (i.e., finding the nearest neighbor vectors for an input vector) for large unstructured datasets (with billions of items) is challenging, especially for applications with strict service level objectives (SLOs). Existing solutions trade query accuracy for latency, but without any guarantees, causing SLO violations.

This paper presents Auncel, a vector query engine for large unstructured datasets that provides bounded query errors and bounded query latencies. The core idea of Auncel is to exploit local geometric properties of individual query vectors to build a precise error-latency profile (ELP) for each query. This profile enables Auncel to sample the right amount of data to process a given query while satisfying its error or latency requirements. Auncel is a distributed solution that can scale out with multiple workers. We evaluate Auncel with a variety of benchmarking datasets. The experimental results show that Auncel outperforms state-of-the-art approximate solutions by up to 10× on query latency with the same error bound ($\leq$ 10%). In particular, Auncel only takes 25 ms to process a vector query on the DEEP1B dataset that contains one billion items with four c5.metal EC2 instances.

1 Introduction

Vector query engines for unstructured datasets (e.g., images, videos and texts) are a key building block for modern applications including recommendation [1–4], recognition [5–8] and biological information retrieval [9–11]. This is enabled by the breakthroughs in deep learning [12] that allow unstructured data to be represented as high-dimensional feature vectors. A vector query is to find the top-k nearest neighbor vectors in a dataset for an input vector.

With the explosive growth of unstructured data [13, 14], a central challenge for vector query processing is to satisfy strict service level objectives (SLOs) for applications on large unstructured datasets that contain millions and even billions of items. For instance, a face recognition task is to match a human face from an input image against a database of faces. With deep convolutional neural networks [6], each face image is converted into an embedding vector. Consequently, the recognition task becomes a top-k nearest neighbor (KNN) search problem, i.e., finding the nearest neighbor vector of the query vector among the database vectors. The person corresponding to the nearest neighbor vector is the recognition result. Performing exact KNN search (e.g., through pairwise comparison between query vector and each stored vector) is costly in terms of computation resources, and more importantly, is hard to achieve low query latency. As a result, approximate top-k nearest neighbor (ANN) search [15–18] is widely used by vector query engines to tradeoff query accuracy for latency. The basic idea of ANN search is to sample a subset of the dataset for finding the top-k, and the sampling size affects the query accuracy and latency.

A key requirement for approximate query processing is to provide *performance guarantees* in order to meet SLOs [19–21]. Performance guarantees are defined in terms of error bounds (e.g., $\leq$10% query error) or latency bounds (e.g., $\leq$25 ms query latency). Existing systems [22–29] exploit various ANN algorithms [15–18] and system optimizations to optimize query accuracy and latency. However, these systems do not provide any performance guarantees.

Faiss [22] and AnalyticDB-V [24] are widely-used open-source and commercial vector query engines, respectively. Unfortunately, they do not provide any performance (error or latency) bounds. They build a profile to map query errors to sampling sizes for a given dataset. It is possible to leverage the profile to pick an appropriate sampling size to meet an error bound. But the problem is that the profile is *query-agnostic*: it ignores the characteristics of individual query vectors, and uses a *fixed* sampling size for all query vectors under the same error bound. Consequently, the sampling size is too pessimistic—the maximum sampling size among all query vectors has to be used to meet the error bound. This leads to excessive redundant computation.

Learned Adaptive Early Termination (LAET) [30] is a recent work that leverages machine learning to optimize vector query processing. It trains a gradient boosting decision tree model to predict when to stop searching for a given query in order to reduce query latency. It focuses on average query accuracy, and does not provide any error or latency bounds. LAET includes a heuristic to adapt the decision tree model by multiplying a hyperparameter to the prediction result to meet

a given error bound. But the model treats the entire structured ANN index as a blackbox, which performs poorly on query latency for bounded errors.

More importantly, existing systems focus on a *single-node* setup and use a *single* worker to process each query. Distributed processing is critical for vector queries over large unstructured datasets with billions of items. Conceivably, one can replicate a dataset to multiple workers, and process multiple queries in parallel—one query by each worker. This naive solution has high memory footprint for billion-scale datasets (e.g., 360 GB for DEEP1B [31] dataset). Moreover, it cannot reduce query latency with more workers, making it hard to achieve latency bounds for billion-scale datasets.

We present Auncel, a vector query engine for large unstructured datasets with performance guarantees. Different from existing systems, Auncel allows users to specify an error bound or latency bound for an input vector. The core of Auncel is a query-aware and error-aware *error-latency profile* (ELP) that enables Auncel to minimize the query latency for an error bound and maximize the query accuracy for a latency bound. Auncel is a distributed solution that can reduce the query latency with more workers. To the best of our knowledge, Auncel is the first distributed vector engine that provides bounded errors and bounded latencies.

There are two primary challenges in realizing Auncel. The first challenge is to decide the appropriate sampling size for an individual query vector under a particular error or latency bound. Auncel uses a *whitebox* approach that exploits the geometric properties in the high-dimensional space to explicitly model the relationship between sampling sizes and query errors. This enables Auncel to build more accurate ELPs than existing query-agnostic or blackbox approaches. Auncel immediately terminates the search process when the error bound can be guaranteed based on the ELP to minimize query latency. In terms of the latency bound, Auncel exploits the nature of vector query processing and uses a runtime approach to maximize query accuracy.

The second challenge is to scale Auncel out to multiple workers in order to reduce query latency. A natural approach is to shard a dataset among workers and aggregate workers' results for query processing. The nuance is to correctly set the local error bound for each worker. Naively setting the local error bound to be the target error bound would magnify the global error after aggregation. Auncel applies probability theory to calibrate the local error bound for each worker in order to bound the global error. We theoretically prove that Auncel is able to bound the global error with high probability.

We implement a prototype of Auncel, and extensively evaluate it with a variety of benchmarking datasets. The results show that Auncel outperforms Faiss [22] by 1.3–10× and LAET [30] by 1.4–3.6× on the single-node setup. For the distributed setup, Auncel is able to process a vector query under 25 ms for the DEEP1B dataset which contains one billion vectors of 96 dimensions with 128 workers (using four c5.metal EC2 instances).

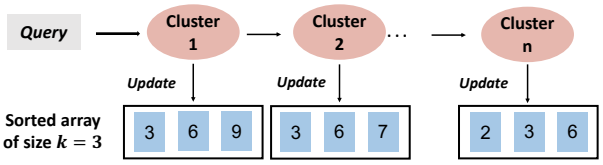


Figure 1: IVF workflow.

In summary, we make the following contributions.

- We present Auncel, to the best of our knowledge, the first distributed vector query engine that provides bounded query errors and bounded query latencies.
- We propose a whitebox approach that leverages high-dimensional geometry to build accurate ELPs, and apply probability theory to calibrate each worker to scale out.
- We implement an Auncel prototype. The evaluation shows that Auncel outperforms Faiss by up to 10× and LAET by up to 3.6×, and processes a vector query within 25 ms for billion-scale datasets with 128 workers.

2 Background and Motivation

In this section, we begin by introducing the background of vector queries on unstructured datasets. We then describe current solutions and their limitations to support vector queries on large unstructured datasets, which motivates the design of Auncel. Finally, we describe the challenge to scale out vector query processing.

2.1 Vector Queries on Unstructured Datasets

The common practice for managing and querying unstructured datasets is to use deep neural networks (DNNs) to process each item and represent each item as a high-dimensional feature vector [32–34]. A vector query on an unstructured dataset is to find the top-k vectors in the dataset that are most similar to the query vector. The most widely-used similarity metrics between two vectors are Euclidean distance and Angular distance. KNN search returns the top-k most similar vectors (i.e., nearest neighbors), and ANN search returns the approximate top-k nearest neighbors. KNN becomes impractical for large datasets with millions or billions of items due to long query latency. ANN trades off accuracy for latency, and is the de facto solution for vector query processing on large unstructured datasets. Another reason for the wide adoption of ANN is that it is unnecessary to output the exact top-k items for many vector query processing tasks, as DNN models themselves are not perfect when generating these vectors.

The basic idea of ANN search is to use an indexing structure to sample a subset of the dataset to find the top-k neighbors. Inverted file index (IVF) [15, 25, 35] is a state-of-the-art ANN algorithm. While IVF has many variants, it has the following general workflow. It trains a list of cluster centroids by k-means clustering [36] offline. These cluster centroids form the index of the dataset; each vector is assigned to the closest

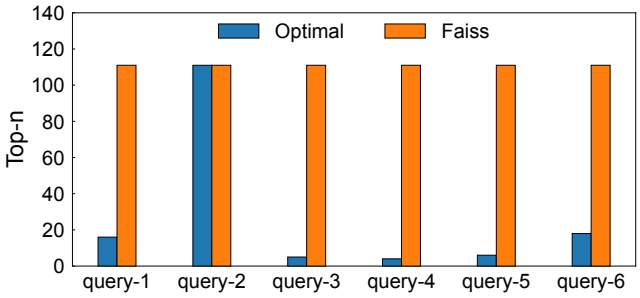


Figure 2: Redundant computation in Faiss.

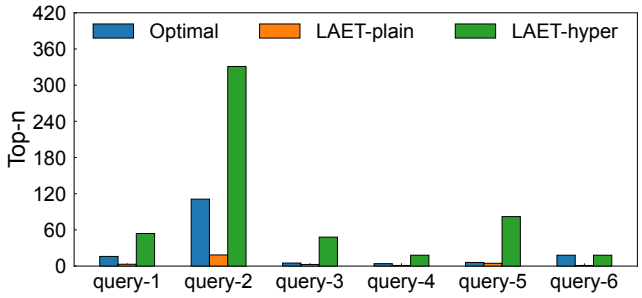


Figure 3: Redundant computation in LAET.

cluster. Given a query vector, IVF first computes the distance between each centroid and the query vector. Then it chooses the top-n nearest centroids, and processes the corresponding clusters of these centroids one by one. It maintains a sorted array of size k, and updates the array after performing pairwise distance calculation in each cluster as Figure 1. The k vectors remained in the array at the end are returned as the query result. The vectors in the array are called *intermediate result* during the processing. In this figure, each value in the array represents the vector's index in the ground truth result of exact search. The vectors in the top-n clusters are the sampling vectors. n determines the sampling size, and thus controls the tradeoff between accuracy and latency.

2.2 Bounding Performance for Vector Queries

Providing bounded performance for query processing is a key requirement for meeting SLOs of applications [19–21]. There are two typical types of performance bounds: error bounds and latency bounds. The query processing engine is expected to minimize query latency when given an error bound, and maximize query accuracy when given a latency bound.

Limitations of existing solutions. Existing systems [22–29] do not provide bounded performance, and leave the choice of the sampling size to users. While it is possible to adapt the mechanisms in existing systems to provide bounded performance, simply doing so yields undesirable results. Faiss [22] and AnalyticDB-V [24] build a profile by sampling some queries to map query errors to different n values (exponential power of two in practice to save the map building time) after building an IVF index for a dataset. To guarantee bounded errors, they use the n whose worst-case error is no bigger than the bound. This pessimistic choice of n has poor performance, because it is *query-agnostic*. It ignores the characteristics of individual queries and uses a fixed n for all queries under a given error bound. Some queries may use a smaller n (and thus achieve better latency) without violating the error bound.

To illustrate the problem, we randomly select six query vectors in DEEP10M [31] and assign them the same error bound (10%). In Figure 2, the optimal bars are the minimal values of n to reach the error bound for each query vector, and they are calculated through grid search of parameter n for the six queries respectively. Since Faiss uses the same value of n for all queries, the value is dominated by query-2; other queries do not need such a large n to meet the error bound. A larger n means searching more clusters, i.e., longer query latency. The naive solution of using a query-agnostic fixed value for n has a 10× gap from the optimal for some queries in this example.

To alleviate this problem, LAET [30] leverages machine learning to adaptively decide n among different query vectors. However, LAET is designed to reduce latency under *average* query error and is incapable to guarantee *bounded* error. It trains a decision tree model with LightBGM [37] and treats IVF index as a blackbox. The model takes the query vector, the intermediate result and some features of the clusters as input and outputs n for a given query. Due to the overhead of running a machine learning model for each query, LAET cannot use complex models and it also simplifies the model input. For instance, it only considers a small portion of intermediate result and cluster centroids. Therefore, the model cannot accurately predict n with blackbox fitting. To guarantee an error bound, LAET includes a heuristic to adapt the model by multiplying a hyperparameter to the prediction result—a tighter bound requires a larger hyperparameter. However, applying such a hyperparameter to all queries given the same error bound induces severely redundant computation with the inaccurate model. This is because this inaccurate blackbox model needs a very large hyperparameter to guarantee the error bound for all queries, but most queries only require a small one. Consequently, the values of n generated by LAET are also far from the optimal. Besides, tuning the hyperparameter for different error bounds is error-agnostic.

We continue with the previous DEEP10M example to show the problem of LAET. As shown in Figure 3, the bars of LAET-plain are lower than those of the optimal, indicating that the plain LAET solution with blackbox fitting is inaccurate to predict top-n and cannot provide bounded errors. LAET-hyper, which is LAET with the aforementioned heuristic, sets the hyperparameter large enough to ensure the bounded error for all the six queries. The hyperparameter is dominated by query-6; other queries can use a smaller hyperparameter, i.e., just enough to match the optimal. Therefore, LAET has the similar problem as Faiss and AnalyticDB-V. The inaccurate blackbox model introduces a 5× gap from the optimal for some queries in this example.

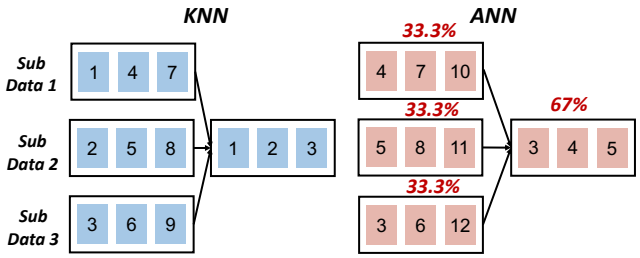


Figure 4: Problem when scaling error bounds.

Opportunity: Geometric structures in ANN indexes. The key to address the problem is building an accurate, lightweight ELP. Our core idea is to exploit the geometric structure and computation pattern of ANN indexes to establish the relationship between sampling sizes and query errors with high-dimensional geometry. The geometric intuition is that a query vector needs a large value of n if the vector is located at the *boundary* of clusters. Similarly, searching a small number of clusters, i.e., a small value of n, is sufficient, if the query vector falls close to a centroid. Thus, we can formulate the entire search procedure with geometric formulas, and use whitebox approach to explicitly model the relationship between sampling sizes and query errors.

2.3 Applying to Distributed Settings

Existing systems focus on a single-node setup and use a single worker to process each query. Replicating the dataset to each worker and processing multiple queries in parallel only increases throughput. It does not help with per-query latency, and has high memory footprint for each worker, both of which are undesirable for billion-scale datasets. A common approach is sharding, i.e., partitioning the dataset among multiple workers. Each worker finds the local top-k in its own shard (i.e., a *map* operation), and then a leader aggregates the local results to the global top-k (i.e., a *reduce operation*). This works well for exact search (KNN), as the aggregated result is identical to the ground truth. However, for approximate search (ANN), the error of the aggregated result is not bounded, even if the error of the local top-k on each worker is bounded.

To see why this is the case, consider the example in Figure 4. The example includes three workers, the value of k is 3, and the error bound is 35%. We show the local top-k at each worker and the aggregated global top-k. Each value represents the corresponding top-k vector's index in the global ground truth result of exact search. The results of KNN is on the left and that of ANN is on the right. In KNN, each worker returns the exact local top-k, and the aggregated top-k vectors are the true top-k (i.e., the ground truth). In ANN, the error of the local top-k at each worker is 33.3%, which satisfies the error bound. However, after aggregating the local top-k, the error of the global top-k is 66.7%; only one vector (with index 3) is in the true global top-k vectors.

To address the problem, we need to calibrate the local error bounds when finding the local top-k at each worker; we cannot directly use the global error bound. We apply probability theory to calibrate the local error bounds to ensure that the global error is bounded with high probability.

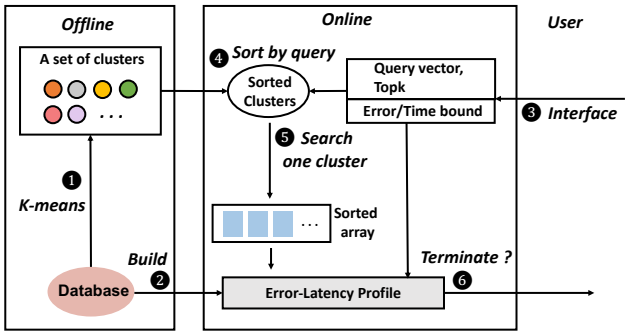


Figure 5: Auncel architecture.

3 Auncel Overview

We present Auncel, a vector query processing engine for large unstructured datasets with performance guarantees. Auncel exploits the geometric properties of ANN index structures to build accurate, lightweight ELPs. Such ELPs enable Auncel to sample just enough data to answer vector queries within their error or latency bounds. To scale out vector query processing for large datasets with billions of items, Auncel adopts map-reduce style dataflow operations, and applies probability theory to calibrate the local error bounds at each worker. Figure 5 shows the overall architecture of Auncel. We provide a brief overview of Auncel in this section.

User interface. Auncel allows users to tradeoff between accuracy and latency with user-defined performance bounds ❸. Specially, a user can specify an error bound or a latency bound for a query vector and a value of k (i.e., how many nearest neighbors to return) as follows.

- **Error bound ε.** The user specifies an error bound ε, and Auncel returns a result within ε error as soon as possible.
- **Time bound t.** The user gives a time bound t, and Auncel returns the most accurate result within t time.

Offline. Similar to all query processing engines for unstructured datasets, Auncel first uses IVF to build an ANN index for a given dataset offline ❶. IVF divides the vectors in the database into a few clusters with k-means clustering. It maintains the centroids of each cluster; each vector in the dataset is assigned to the closest cluster. In addition to the ANN index, Auncel samples synthetic or example queries to build an ELP for the dataset ❷. ELP building techniques include fitting geometric formulas and substituting some complex operators with pre-calculated key-value pairs to reduce overhead.

Online. At runtime, vector queries are issued to Auncel for processing. Each query includes a vector, a value of k, and an error/time bound. To process an incoming query, Auncel first evaluates the distance between the query vector and each

Symbol	Description
q	Query vector
t	Latency bound
ε	Error bound
l	Number of clusters
C_i	The i_{th} closest cluster to q (a hyper polyhedron)
ε_i	Error after processing $\{C_1 ... C_i\}$
S	Set of database vectors
S_i	Intermediate result after processing $\{C_1 ... C_i\}$
S_{gt}	Ground truth result of exact search ($S_{gt} = S_l$)
j^*	$\lvert S_i \cap S_{gt} \rvert$ after processing $\{C_1 ... C_i\}$
$\varphi_i(j)$	Scaling factor of the j_{th} element in S_i
$\lambda_i(j)$	Distance between q and the j_{th} element in S_i
$B(r)$	Sphere (Ball) with center q and radius r
$P_j(m)$	$B(\lambda_i(j)) \cap C_m$
$N_j(m)$	Number of the vectors within $\bigcup_{\eta=1}^{m} P_j(\eta)$
$V(G)$	Volume of geometric body G

Table 1: Key notations in problem formulation.

centroid, and then sorts the clusters by distance in ascending order ❹. According to the sorted clusters, Auncel performs pairwise distance calculation between the query vector and each stored vector cluster by cluster, and updates the sorted array (i.e., the intermediate result) ❺. After processing each cluster, Auncel uses the intermediate result and the centroids to predict the current error based on the ELP ❻. If the error or time bound can be satisfied, Auncel terminates the search process in ❺, and returns the vectors in the array as the result.

4 Auncel Design

In this section, we present the design of Auncel. We first describe the problem formulation (§4.1) and the key ideas (§4.2). Then we show how to build error profiles (§4.3) and latency profiles (§4.4). Finally, we describe how to apply our solution to distributed settings (§4.5). Some key notations in the design are listed in Table 1.

4.1 Problem Formulation

We first mathematically formulate the problem of vector query processing on unstructured datasets. Let $S = \{v_1, v_2, ...v_N\} \in \mathbb{R}^d$, where S is an unstructured dataset, N is the number of vectors in S, and v_i is a d-dimensional vector in S. Let $q \in \mathbb{R}^d$ be a query vector. Given a value $k \in \mathbb{N}_+$ and $k \leq N$, a vector query with q is to find the top-k nearest neighbors of q in S, according to a pairwise distance function $d(q, v_i)$. The distance function typically computes Euclidean distance or Angular distance between two vectors. The ground truth of the top-k nearest neighbors S_{gt} is obtained when searching in the entire dataset S. S_{gt} are often sorted according to $d(q, v)$.

$$S_{gt} = argmin^k_{v \in S} \quad d(q, v) \tag{1}$$

Finding the exact top-k nearest neighbors has high query latency for large datasets, which may violate latency SLOs

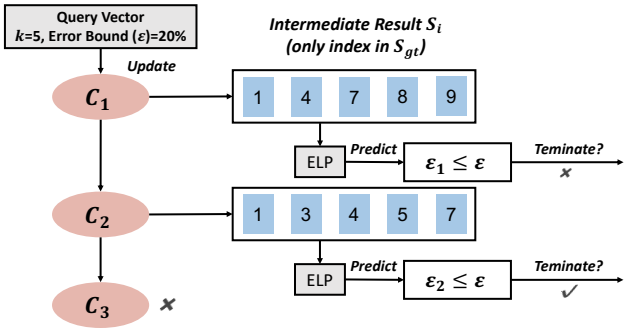


Figure 6: Example to show key idea and workflow of Auncel.

for applications. To trade query accuracy for query latency, a subset $S_a \subset S$ can be sampled to find the approximate top-k nearest neighbors S_r for lower latency.

$$S_r = argmin^k_{v \in S_a} \quad d(q, v). \tag{2}$$

The accuracy (recall) and error of S_r are defined as follows.

$$Accuracy \triangleq \frac{|S_{gt} \cap S_r|}{k} \tag{3}$$

$$Error \triangleq 1 - Accuracy \tag{4}$$

4.2 Workflow and Key Idea

Auncel allows users to specify an error bound or a latency bound for a query. When an error bound is given, Auncel minimizes query latency; when a latency bound is given, Auncel maximizes query accuracy. As we have described in §2, the basic approach for processing a vector query with an ANN index is to first compute the distances between the query vector and the centroids of the clusters in the index and then search cluster by cluster based on the ascending order of the distances. The key idea of Auncel is to build an accurate ELP so that after searching each cluster, Auncel can consult the ELP to decide whether to terminate the search.

Workflow of Auncel. We use a concrete example in Figure 6 to illustrate the workflow of Auncel. In this example, the ANN index partitions the dataset into three clusters. Each cluster (C_i) is a polyhedron in d-dimensional space. The value of k in top-k is 5, and the error bound is 20%. The three clusters are sorted in ascending order by their centroids' distances to the query vector. Auncel searches the three clusters based on the order one by one. The three clusters may have different numbers of vectors due to the imbalance property of k-means. It does not impact the error profile since Auncel terminates the search as soon as the current error is guaranteed. However, the imbalance property leads to inaccurate latency profile that we will discuss later in §4.4.

After processing C_1, Auncel updates the intermediate result (a sorted array) which represents the query vector's top-k

Algorithm 1 Error Profile

```
1: ε ← Error Bound, i ← 1
2: while i <= l do
3:     Perform search computation in cluster C_i
4:     geometric properties ← S_i , cluster centroids
5:     ε_i ← ELP (geometric properties)
6:     if ε_i ≤ ε then
7:         break
8:     i ← i + 1
9: Return intermediate result
```

nearest neighbors in C_1. Each element in the sorted array is a pair of a database vector's index and its distance to the query vector. For simplicity, the figure only shows the element's index in ground truth result. Auncel uses the intermediate result as the input of the ELP to predict the current error, and decides whether to terminate the search. The ground truth vectors have top-k indexes [1, 2, 3, 4, 5] in this example, which is obtained after searching all three clusters. As the intermediate result after searching C_1 is $[1,4,7,8,9]$, only two vectors in the intermediate result (the two with indexes 1 and 4) belong to the true top-k. Therefore, the current error is 60%, which is still above the error bound 20%. Auncel continues to search C_2. The intermediate result after searching C_2 contains four vectors in the true top-k (the four with indexes 1, 3, 4 and 5), and the current error decreases to 20%, which satisfies the error bound. Therefore, it is safe for Auncel to terminate the search and return the sorted array as the query result.

The workflow is summarized in Algorithm 1. Predicting the error with ELP in line 5 is the key to guarantee bounded errors and minimize query latency. If the predicted error is smaller than the actual error, the system terminates search too early, and fails to meet the error bound; if the predicted error is larger than the actual error, the system unnecessarily continues to search more clusters, which increases query latency. Thus, the main challenge is to build an accurate ELP to accurately predict the current error after searching each cluster.

Key idea of Auncel. To understand how Auncel addresses this challenge, consider the intermediate result after searching C_1. The first element in the intermediate result (1) is also the first element in ground truth (sorted array after searching all three clusters). But, the second element (4) is the fourth element in ground truth, and the the third element (7) is not in the top-k (k=5). We define the scaling factor of the j_{th} element in the intermediate result after processing C_i as follows, where $index_{gt}$ is the element's index in ground truth.

$$\varphi_i(j) \triangleq index_{gt}/j \quad (\geq 1) \tag{5}$$

If $\varphi_i(j)$ is known, then the current error of the intermediate result after searching C_i can be calculated. Specifically, we compute j^* such that

$$j^* = argmax_j\{j \cdot \varphi_i(j) \leq k\}. \tag{6}$$

The elements from 1 to j^* in the intermediate result belong to the true top-k. j^* is often calculated through binary search. The current error ε_i after processing C_i is

$$\varepsilon_i = 1 - j^*/k \tag{7}$$

For any j, $\varphi_i(j)$ converges to 1 when i increases to the number of clusters (l). Correspondingly, j^* converges to k and ε_i converges to 0. When searching the clusters one by one, Auncel terminates the process immediately when ε_i becomes no bigger than the error bound.

Therefore, we convert the problem of building an accurate ELP to accurately estimating $\varphi_i(j)$. We exploit the geometric properties of ANN indexes in high-dimensional space to estimate $\varphi_i(j)$. Since Auncel is designed to provide bounded errors, it is sufficient to estimate the upper bound of scaling factor $\varphi_i(j)$, which we describe next.

4.3 Handling Error Bounds

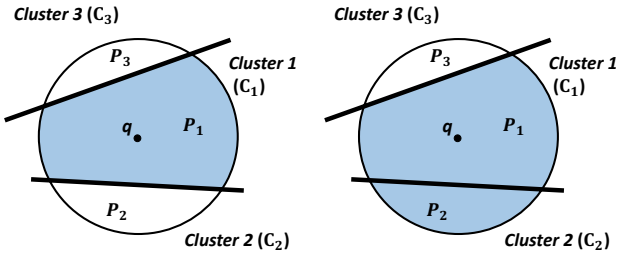
Auncel minimizes query latency under a given error bound using scaling factors in the i_{th} error prediction (φ_i). We perform a detailed analysis of $\varphi_i(j)$ from a geometric perspective, and design a formula to calculate the upper bound of $\varphi_i(j)$ under the two most prevalent distance metrics.

4.3.1 *Scaling Factor* under Euclidean Distance

We first focus on Euclidean distance, the most widely-used and intuitive distance metric, which measures the length of the line segment between two anchor vectors in geometry.

We have two key insights. Our first insight is to leverage the geometric structure of the IVF index. IVF shards the entire dataset into l clusters (C) by k-means clustering, and the clusters are sorted as $C = \{C_1, C_2 ... C_l\}$. Due to the rule of k-means, C_i is a d-dimensional *polyhedron* and the boundary between C_i and C_j is the $(d-1)$-dimensional mid-vertical plane of the line segment connecting the two clusters' centroids. Thus, we can divide the entire space into several parts based on these boundaries. Our second insight is to exploit the local geometric properties of q which belongs to C_1. The top-k ground truth vectors gather around q and form a sphere, $B(\lambda_l(k))$ with radius $\lambda_l(k)$ and center q in d-dimensional space. For any vector within $B(\lambda_l(k))$, it is a member of S_{gt}. For instance, if $B(\lambda_l(k))$ locates within C_1, it is sufficient to search in the first cluster to get ground truth.

The combination of these two insights allows us to compute $\varphi_i(j)$ with high-dimensional geometry. We know that the k ground truth vectors are distributed in sphere $B(\lambda_l(k))$, and the sphere is divided into many parts by the boundaries between C_1 and other clusters. For example, we have three clusters in total and query vector q in Figure 7. The sorted clusters are $\{C_1, C_2, C_3\}$, and all vectors within the sphere belong to S_{gt}. This sphere is cut into three parts—P_1, P_2, P_3—by the two boundaries, and N_i is the number of database vectors in the scope of $\bigcup\{P_1 ... P_i\}$ ($i = 1,2,3$). In Figure 7, the numbers of the vectors within the shaded areas are N_1 and N_2,



(a) Search in one cluster. (b) Search in two cluster.

Figure 7: Geometric demo for calculating scaling factor.

respectively. In Figure 7(a), scaling factor $\varphi_1(N_1) = \frac{k}{N_1}$ since the $N_{1_{th}}$ element in S_1 is the k_{th} element in S_{gt}. Figure 7(b) shows the scaling factor $\varphi_2(N_2) = \frac{k}{N_2}$. To extend to general settings, we define $P_j(m)$ as the intersection of $B(\lambda_i(j))$ and C_m after processing C_i. Consequently, $N_j(m)$ is the number of vectors within $\bigcup_{\eta=1}^{m} P_j(\eta)$ which gives that:

$$\varphi_i(j) = N_j(l)/N_j(i). \tag{8}$$

l is the number of clusters and i represents the current cluster.

We observe that the vectors of real-world datasets conform to *local uniform distribution*, and k in most query workloads is no bigger than 100, which means the vectors within the scope of $B(\lambda_l(k))$ nearly conform to uniform distribution. We provide a measurement in Appendix A.1 to confirm this observation. Let *den* represents the local density of $B(\lambda_l(k))$. We get that $N_j(m) \approx V(\bigcup_{\eta=1}^{m} P_j(\eta)) \times den$. Hence,

$$\varphi_i(j) = V(B(\lambda_i(j)))/V(\bigcup_{m=1}^{i} P_j(m)). \tag{9}$$

V is the volume function. Since $P_j(1)$ has different geometric meaning with $P_j(m)$ $(m \geq 2)$ (spherical cap) and is complex to calculate, we use the following inequation:

$$\begin{aligned}\varphi_i(j) &= \frac{1}{1 - \frac{V(\bigcup_{m=i+1}^{l} P_j(m))}{V(B(\lambda_i(j)))}} \\ &\leq \frac{1}{1 - \frac{\sum_{m=i+1}^{l} V(P_j(m))}{V(B(\lambda_i(j)))}} \leq \frac{1}{b - a \times U}.\end{aligned} \tag{10}$$

Different spherical caps (e.g., P_1, P_2 in Figure 7) may intersect with each other. The union of all spherical caps has a smaller volume than the sum of the volumes of all spherical caps, which leads to the first $\leq$. We apply $b - a \times U$ to substitute such complicated volume calculation in d-dimensional space, where a, b are parameters to fit offline and $a \times U$ is the geometric *upper bound* of the volume ratio. Appendix A.2 contains the detailed analysis of $a \times U$, and we conclude one of the upper bound functions is

$$U = \sum_{m=i+1}^{l} \arccos(x_m) \quad (0 \leq x_m \leq 1). \tag{11}$$

Algorithm 2 Latency Profile

```
t ← Time Bound, i ← 1
t_0 ← CurrentTime()
while i ≤ l do
    Perform search computation in cluster C_i
    t_c ← CurrentTime()
    if t_c − t_0 ≥ t − δ then
        break
    i ← i + 1
Return intermediate result
```

where $x_m = \frac{db_m}{\lambda_i(j)}$ and db_m is the distance between query vector and the boundary of C_1, C_m. We do not consider the circumstance when $x_m > 1$.

In Formula 11, the time complexity of calculating the upper bound of $\varphi_i(j)$ is $O(l)$ since $\arccos(x_m)$ only costs constant time. We use binary search to calculate j^* with Formula 6, which concludes that the time complexity to predict ε_i is $O(l \times \log(k))$ while the space complexity is $O(1)$.

4.3.2 *Scaling Factor* under Angular Distance

Another widely-used vector distance metric is Angular distance, which evaluates the angle between two anchor vectors. Its geometric meaning allows us to transform Angular distance into Euclidean distance. Specifically, we project all database vectors onto the unit sphere in d-dimensional space through *vector normalization*, while maintaining the Angular distance between any vectors. Thus, we substitute angle with the line segment on such unit sphere, and the theoretical analysis in §4.3.1 holds under Angular distance.

4.4 Handling Latency Bounds

Given a latency bound, Auncel maximizes query accuracy for a query vector. Conceivably, one can build a latency profile to capture the relationship between sampling sizes (i.e., the number of clusters to search) and query latencies. Then Auncel can consult the latency profile to get how many clusters to search based on a given latency bound. However, building such a profile is difficult, because the clusters have different sizes and the order of the set of clusters to search vary between different query vectors.

We design a *runtime* solution that exploits the *monotonicity* property of vector query processing to handle latency bounds, obviating the need of building a latency profile offline. Specifically, vector query processing searches the clusters in the ANN index one by one. The search is based on ascending order of the distances between the clusters' centroids and the query vector. The accuracy of the intermediate result increases when searching more clusters. As such, Auncel tracks the used time when searching the clusters, and terminates the search when the used time is close to the time bound. This ensures that Auncel uses as much time as possible in processing to maximize query accuracy. Algorithm 2 shows

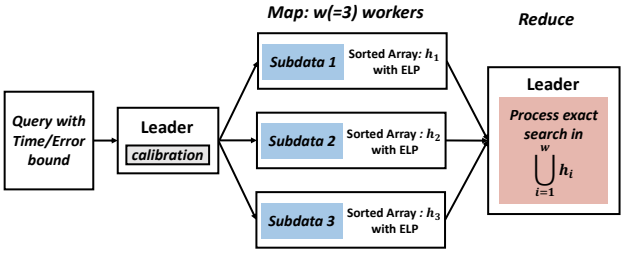


Figure 8: Auncel runtime with map-reduce.

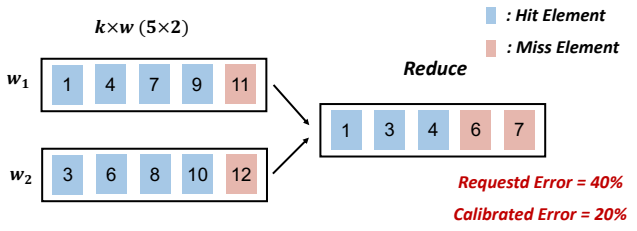


Figure 9: Error calibration of Auncel.

the pseudocode of processing vector queries under latency bounds. To guarantee the search terminates before the time bound, the algorithm stops the search when the used time is a (configurable) δ before the time bound. δ is influenced by the first cluster after Auncel terminates the search, i.e., C_i if the termination condition is triggered after processing C_{i-1}. The larger δ is required if C_i has more vectors and costs more time to process. To guarantee the latency bounds in any circumstances, δ is tuned according to the cluster with the largest number of vectors.

4.5 Applying to Distributed Settings

We emphasize that the distributed solutions mentioned in §2.3 either cannot reduce query latency (i.e., only improve throughput) or leads to error amplification. To scale out, Auncel uses map-reduce style processing to process vector queries with multiple workers while preserving the error and latency bound. Auncel divides distributed vector processing into two phases—a `map` phase and a `reduce` phase. As shown in Figure 8, Auncel randomly and uniformly shards the dataset into multiple partitions. Each worker owns one partition, and builds a local ANN index and a local ELP. One of the workers is elected as the leader, which controls the global query processing.

When a vector query with an error/time bound (which we call global bound) comes, the leader calibrates the bound to get an error/time bound to be used by each worker (which we call local bound). In the `map` phase, each worker uses its local ANN index and ELP to process the query on its own partition. In the `reduce` phase, the leader collects the local results from the workers, and performs exact top-k search on these collected results to produce the final result.

Calibration of error/latency bounds. As we have shown in §2.3, the local errors can be amplified in the `reduce` phase when the local results are aggregated to the final result. Directly using the global error bound as the local error bound in the `map` phase would cause the final error obtained by the `reduce` phase to be larger than the global error bound. To address this problem, we design an error bound calibration mechanism based on probability theory. The leader calibrates the local error bound before distributing the work to the workers in the `map` phase.

Figure 9 shows an example with two workers to illustrate the calibration mechanism. The global error bound ε is 40%, and the local error bound after calibration ε_c is 20%. Hit elements are the intersection of S_r and S_{gt}. The `map` phase is done by the workers individually, and the local error at each worker is 20%, which meets the local error bound ε_c. The `reduce` phase aggregates the local results and the global error is 40%. While the global error is bigger than 20%, it still meets the global error bound ($\varepsilon = 40\%$). The probability of the example in the figure is: $\binom{5}{3} * \binom{5}{5} / \binom{10}{8} = \frac{2}{9}$, where $\binom{n}{m}$ means the *combination number*. From classical models of probability, it means the probability of the random event that selects eight hit elements of all ten ground truth vectors and three elements are situated in the first five ground truth.

Formally, we define the number of hit elements of the first n elements in the ground truth as H_n. The worst case is $H_{k\times w} = k \times w \times (1-\varepsilon_c)$, which means each worker just meets the local error bound. The global error can be represented by $1 - H_k/k$. The probability of a random event $H_k = i$ is

$$P(H_k = i) = \frac{\binom{k}{i} \times \binom{k\times(w-1)}{k\times w\times(1-\varepsilon_c)-i}}{\binom{k\times w}{k\times w\times(1-\varepsilon_c)}}. \tag{12}$$

Thus, the probability of the random event to meet the global error bound is calculated by:

$$P(H_k \geq (k \times (1-\varepsilon))) = \sum_{i=k\times(1-\varepsilon)}^{k} P(H_k = i). \tag{13}$$

For instance in Figure 9, $P(H_k \geq (k \times (1-\varepsilon))) = P(3) + P(4) + P(5) = \frac{2}{9} + \frac{5}{9} + \frac{2}{9} = 1$, which means the calibrated local error bound (20%) can guarantee the global error (40%) under any circumstances. Auncel starts calibrating ε_c from ε and decreases ε_c by $\frac{1}{k}$ each time. We choose $\frac{1}{k}$ as the loop decrement because the error, which is defined in Formula 3, is a multiple of $\frac{1}{k}$. The probability to guarantee the global error bound ε is calculated through Formula 13. If the probability is greater than γ (99.9% used in our prototype), Auncel stops the calibration and distributes the ε_c to each worker. Therefore, the probability of failing to guarantee the error bound is less than 1-γ (<0.1% when $\gamma = 99.9\%$), i.e., Auncel guarantees the error bound with high probability. As for the convergence

time of the calibration algorithm, the calculation time of Formula 12 is constant since the values of combinatorial numbers are pre-calculated offline. The time complexity of Formula 13 is $O(k \times \varepsilon)$. Consequently, the convergence time of calibration is $O((k \times \varepsilon)^2)$. Since k in real query workloads is not large, the time to calibrate the error bound is relatively small.

Calibrating the latency bound is straightforward since the overhead of the `reduce` phase is negligible compared to the `map` phase. We slightly enlarge δ to include the reduce overhead as the latency calibration.

5 Implementation

We have implemented a system prototype of Auncel with ~3000 lines of code in C++ based on Faiss. We use Faiss for building IVF indexes and similarity search, and extend Faiss with building and applying ELPs for performance bounds. Because Faiss does not support distributed processing, we also implement data sharding, and map-reduce operations for distributed query processing. In theory, Auncel can be integrated with any vector query processing engines. We choose Faiss because it is a widely-used open-source vector query processing engine, and is adopted in production like Meta/Facebook. The code of Auncel is open-source and is publicly available at https://github.com/pkusys/Auncel.

ELP & Distributed setting. We implement the ELP component of Auncel with C++ Standard Library (STL) and Intel oneMKL [38]. ELP works in the query process and is treated as the process controller (monitor). For the distributed setting, Auncel spawns a worker process for each CPU core, and a server machine contains multiple workers. The entire dataset is sharded among the workers, and each worker processes queries on its own shard. Auncel randomly chooses a machine to spawn a leader process to receive the query and distribute the query with calibrated configurations (§4.5) to each worker. After all workers finishing their own query processing, the leader aggregates local results and returns the final results to the user. Each machine contains a daemon process that manages the local workers on the machine and communicates with the leader using TCP sockets. For leaders, Auncel handles query failures by re-executing the query. When receiving a local result from a worker, the leader creates a consistent backup of the result. Users are able to resume the query with the existing backup files, which is similar with the ideas of traditional primary-backup mechanisms [39, 40].

System optimizations. The ELP component imposes computation overhead to the system because of the complex geometric operations for high-dimensional vectors. We follow Faiss to implement Euclidean distance calculation, Angular distance calculation and vector normalization through SIMD instructions of oneMKL. Since SIMD is designed for vector operations such as inner-product and element-wise addition, it significantly reduces the computation overhead of the ELP component. In addition, we pre-calculate key-value pairs of some operations(e.g., arccos) offline and consult the key-value pairs online to improve the performance of these operations.

Dataset	Dimensions	Database Vectors	Query Vectors	Distance
SIFT10M [41]	128	10M	10K	Euclidean
DEEP10M [31]	96	10M	10K	Euclidean
DEEP1B [31]	96	1B	10K	Euclidean
GIST1M [42]	960	1M	1K	Euclidean
TEXT10M [31]	200	10M	10K	Angular

Table 2: Datasets used in the evaluation.

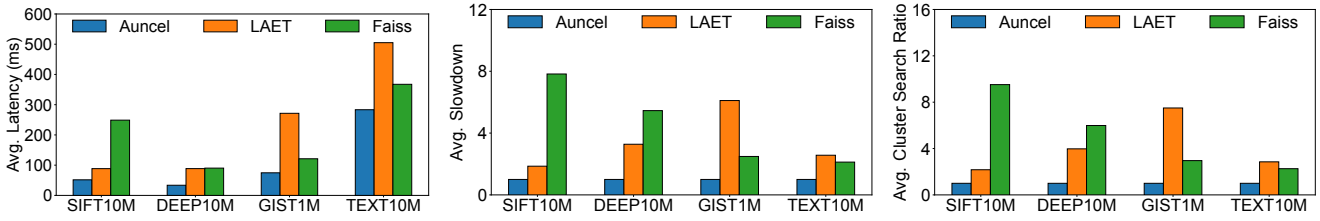
6 Evaluation

In this section, we empirically evaluate Auncel from the following aspects: (i) end-to-end performance improvement over state-of-the-art solutions; (ii) effectiveness of ELP; (iii) validation of the mathematical formulation; (iv) system overhead of Auncel; and (v) scalability. The summary of the experiments is as follows.

- Auncel outperforms LAET [30] and Faiss [22] by up to 3.6× and 10× on average query latency under the same error bound, respectively (§6.1).
- The ELPs built by Auncel are highly accurate across a range of datasets (§6.2).
- The mathematical formulation of Auncel fits well with real-world datasets (§6.3).
- The runtime overhead of Auncel is within 1%, and building an ELP offline can be done within ten minutes (§6.4).
- Auncel scales out near ideally, and only takes 25 ms to process a query on DEEP1B with 128 workers (§6.5).

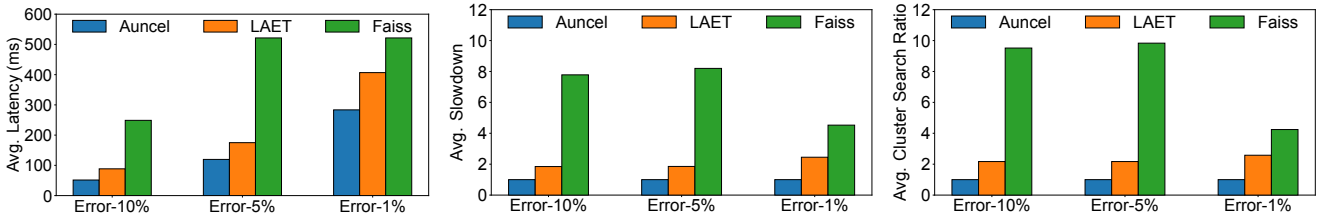
Setup. All experiments are conducted on AWS. We use two EC2 instance types, both configured with Ubuntu 18.04 LTS. For the single-node experiments, we use c5.4xlarge, which is configured with 16 vCPUs (Intel Xeon Platinum 8275CL) and 32 GB memory. For the scalability experiments, we use c5.metal, which is configured with 96 vCPUs (Intel Xeon Platinum 8275CL) and 192 GB memory. The reason of using c5.metal for the scalability experiments is the experiments aim to demonstrate the ability of Auncel to support very large datasets and we need large memory to host DEEP1B (nearly 400 GB memory footprint).

Datasets. Table 2 summarizes the preprocessed datasets used in our experiments. These datasets are widely-used benchmarking datasets for vector query processing in both academic and industry [43, 44]. Each dataset consists of database vectors, query vectors and ground truth. For each query vector, the ground truth records the indexes and distances of its top-100 neighbors. SIFT [41] is a dataset of local SIFT image descriptors [45] with ten million database vectors and ten thousand queries. DEEP [31] is a dataset of CNN [46] image embeddings with one billion database vectors and ten thousand queries. The single-node experiments only use ten million database vectors of DEEP, denoted by DEEP10M. The scala-



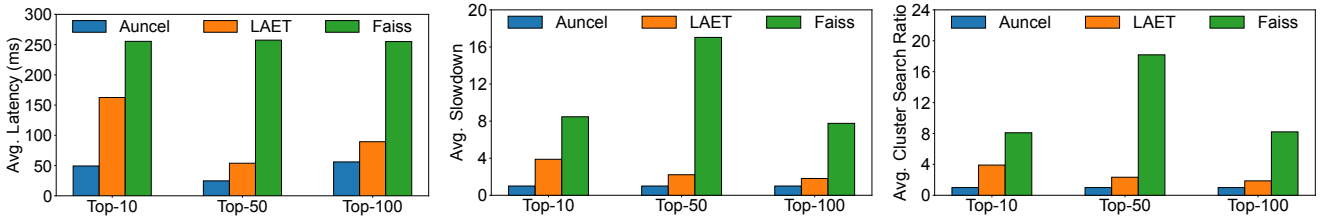
(a) Average latency. (b) Average slowdown of latency. (c) Average cluster search ratio.

Figure 10: Performance under different datasets.



(a) Average latency. (b) Average slowdown of latency. (c) Average cluster search ratio.

Figure 11: Performance under different error bounds.



(a) Average latency. (b) Average slowdown of latency. (c) Average cluster search ratio.

Figure 12: Performance under different values of k.

bility experiments use all database vectors of DEEP, denoted by DEEP1B. GIST [42] is a dataset of global color GIST descriptors [45] with one million database vectors and one thousand query vectors. TEXT [31] is a cross-model [47,48] dataset of texts and images, where the ten million database vectors and the ten thousand query vectors have different distributions in a shared representation space. The TEXT dataset adopts Angular distance as the distance metric, while the other three datasets use Euclidean distance.

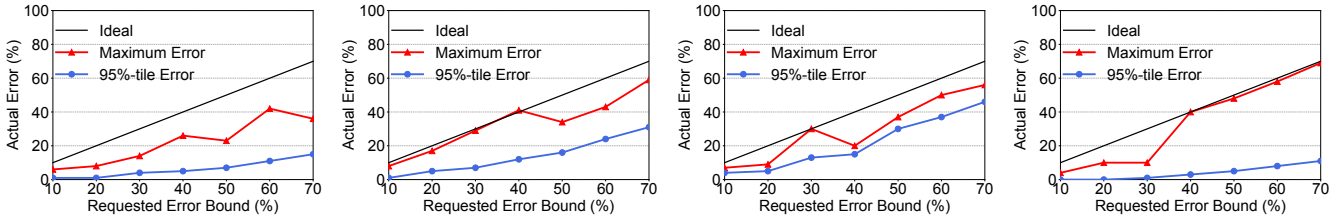
Baselines. We compare Auncel to two baselines.

- Faiss [22] is a widely-used solution for processing vector queries. It uses a fixed approach that picks the same sampling size for all queries under a given error bound.
- LAET [30] is a state-of-the-art solution that uses machine learning to adaptively determines search termination conditions for individual queries.

We emphasize that Faiss and LAET do not provide performance guarantees. To the best of our knowledge, Auncel is the first system that provides performance guarantees for vector query processing. In the experiments, we use the best configurations (e.g., the minimal n in the map for Faiss and the earliest search termination condition for LAET) to guarantee that all queries meet the given error bound. This allows us to fairly compare the query latency of Auncel, Faiss and LAET, while all three systems satisfy the given error bounds.

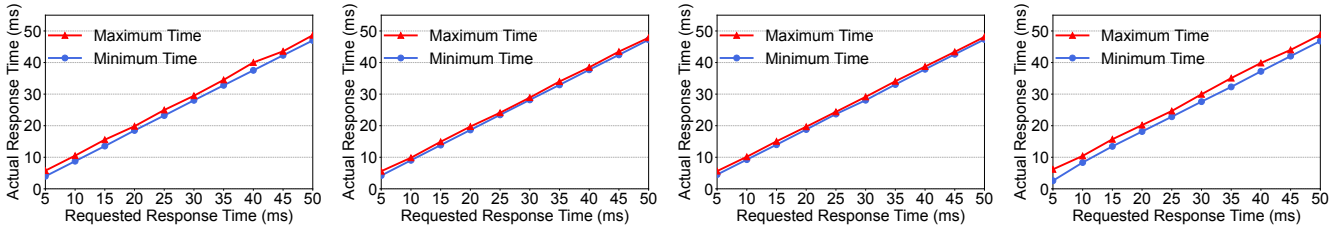
Note that it is necessary for all the three systems (Auncel, Faiss and LAET) to train their ELPs offline with some example queries. In the experiments, unless otherwise stated, we randomly split the query vectors into two parts of equal size, one for training and the other for testing. Because Auncel and LAET only perform search on the training queries once, the ELP building times of the two systems are almost the same. However, the grid search method of Faiss requires searching on the training queries for different top-n (exponential power of two in practice to save time), and the building time is tens of times longer than that of Auncel and LAET.

Metrics. We use average end-to-end query latency, $Ave(T_{system})$ as the main evaluation metric, where T_{system} represents the individual query latency of one of the three systems. In addition, we also report average slowdown of latency and average cluster search ratio. Average slowdown of latency is defined as $Ave(\frac{T_{baseline}}{T_{Auncel}})$. Average cluster search ratio is defined as the average ratio between the number of searched clusters by the baseline and that by Auncel, i.e., $Ave(\frac{N_{baseline}}{N_{Auncel}})$, where N represents the number of searched clusters when processing an individual query.



(a) SIFT10M. (b) DEEP10M. (c) GIST1M. (d) TEXT10M.

Figure 13: Effectiveness of error profiles.



(a) SIFT10M. (b) DEEP10M. (c) GIST1M. (d) TEXT10M.

Figure 14: Effectiveness of time profiles.

6.1 Overall Performance

We compare the end-to-end query latency between Auncel, Faiss and LAET under the same error bound. Auncel outperforms Faiss and LAET under different datasets, different error bounds, and different values of k.

Performance under different datasets. We compare Auncel against the baselines on SIFT10M, DEEP10M, GIST1M and TEXT10M. We use one c5.4xlarge EC2 instance. The error bound is 10% and k in top-k is 100 (i.e., returning top-100 nearest neighbors for a query). The results are shown in Figure 10, which we summarize as follows.

- Auncel achieves 1.8–3.6× lower average end-to-end latency than LAET and 1.3–4.8× lower of that than Faiss under the same error bound. This is because Auncel adaptively and accurately profiles the relationship between errors and sampling sizes, which allows Auncel to sample fewer clusters to generate query results.
- Compared to the gap of average latency between Auncel and the baselines, the gap of average slowdown is larger. This is because Auncel co-adapts individual queries and the error bounds, while the two baselines fail to adapt queries and error bounds at the same time.
- Auncel outperforms the two baselines by up to 9.6× in average cluster search ratio. This means Auncel can reduce the search cost by up to 9.6× in average while meeting the requested error bound, due to the use of the geometric properties when building ELPs.
- Auncel significantly outperforms Faiss and LAET on all the four datasets, which have different characteristics of data and different metrics of distance (Euclidean and Angular).

Performance under different error bounds. To show that Auncel consistently outperforms the baselines when the error bound changes, we vary the error bound from 1% to 10% and run the top-100 workload on the SIFT10M dataset. Figure 11 shows the performance under different error bounds. Auncel achieves 1.4–1.7× lower average end-to-end latency than LAET and 1.8–4.8× lower of that than Faiss.

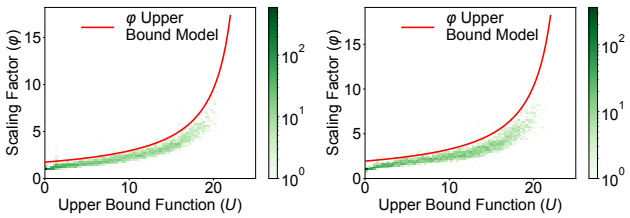
Performance under different values of k. We also vary the value of k from 10 to 100. We fix the error bound as 10% and run different top-k workloads on the SIFT10M dataset. From Figure 12, we observe that Auncel outperforms LAET and Faiss by 1.6–3.3× and 4.6–10× on average query latency, respectively. It confirms that Auncel can handle different values of k in top-k.

6.2 Effectiveness of ELP Techniques

In this set of experiments, we evaluate the effectiveness of the ELP building techniques in Auncel.

Error Profiles. To evaluate our error profiling technique, we run Auncel with the top-100 workload on the four datasets. We vary the error bound from 10% to 70%. Figure 13 illustrates the maximum and 95%-tile actual errors. We also plot the ideal straight lines (i.e., actual error equals to error bound) in Figure 13. Note that the measured actual error is always no bigger than the error bound, which demonstrates the ability of Auncel to guarantee bounded errors. As we increase the error bound, the measured actual error increases as well, indicating that Auncel adapts to the error bound. However, the maximum error does not increase monotonically with the error bound on SIFT10M, DEEP10M and GIST1M. This is because the geometric characteristics (e.g., dimension and distribution) of query vectors vary widely across different datasets.

Time Profiles. To evaluate our time profiling technique, we run Auncel with the top-100 workload on the four datasets and



(a) SIFT10M. (b) DEEP10M.

Figure 15: Validation of Formula 10.

report query latency of each query, with a time bound from 5 ms to 50 ms. Figure 14 shows the results of the maximum and minimum end-to-end query latencies. The results show that each query is terminated before the time bound.

6.3 Validation of Mathematical Formulation

This experiment validates that our theoretical model (Formula 10) fits well with real-world unstructured datasets. In ELP initialization, we compute the geometric upper bound function U and collect the corresponding real scaling factor φ according to the ground truth. We sample a portion of $U-\varphi$ pairs, where φ is the maximal in a small interval, to model the tight upper bound of φ. The larger interval apparently leads to a tighter upper bound. We then use least squares to fit Formula 10 for these $U-\varphi$ pairs. Figure 15 shows the results of the top-100 search workload on SIFT10M and DEEP10M. The results confirm that Formula 10 can well capture the relationship between U and φ, which allows Auncel to accurately predict the runtime errors.

6.4 System Overhead

Runtime overhead. To evaluate the runtime overhead of Auncel, we perform an experiment with top-k search workloads on different datasets. We configure Auncel to search fixed number of clusters and make an error prediction after searching each cluster. For comparison, we measure the ELP prediction time and the entire time of query processing. We also vary the value of k from 10 to 100, and run Auncel on SIFT10M. As shown in Table 3 and Table 4, the average latency can hardly be distinguished between Auncel with ELP and Auncel without ELP. The runtime overhead of using ELP in Auncel is within 1%. Note that the average latency almost stays the same from top-10 search to top-100 search. This is because distance computation dominates in the search process such that top-k relevant computation is negligible.

ELP Building Time. We evaluate the offline time taken for ELP building. We configure Auncel to train 50% query vectors with the top-100 workload on different datasets. Table 5 shows that the time to build ELP is within ten minutes, which is relatively small for datasets with ten million vectors.

6.5 Scalability

Auncel shards a dataset into several partitions. For a vector query, it performs local ANN search with ELP in the `map` phase and aggregates the local results in the `reduce` phase. We evaluate how configurations with different number of workers impact the average latency. We use four c5.metal EC2 instances to verify the scalability of Auncel. In this experiment, we adopt the calibration techniques described in §4.5 to guarantee 10% global bounded error for all queries. We vary the number of workers from 4 to 128, and run the top-100 workload on DEEP1B. Auncel assigns each machine an equal number of workers and spawns a leader worker on one of the machines. The leader worker receives all local top-k results and completes the reduce task. Figure 16 shows that the average latency is almost halved when the number of workers doubles each time, indicating that Auncel scales out *near ideally*. This is because Auncel only transmits the indexes of local top-100 vectors and their corresponding distances to the query vector from the workers to the leader; it does not transmit the raw data of the high-dimensional vectors. The `reduce` phase to aggregate local results takes only a small portion of the total time. Therefore, Auncel can fully leverage the advantages of data parallelism to scale out.

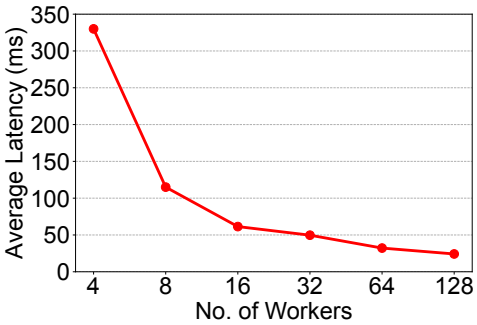


Figure 16: Scalability on multiple workers.

Dataset	Without ELP	With ELP
SIFT10M	68.02 ms	68.41 ms (+0.58%)
DEEP10M	48.16 ms	48.55 ms (+0.81%)
GIST1M	6.86 ms	6.91 ms (+0.70%)
TEXT10M	95.42 ms	96.00 ms (+0.61%)

Table 3: Runtime overhead under different datasets.

Top-K	Without ELP	With ELP
Top-10	68.09 ms	68.47 ms (+0.55%)
Top-50	68.12 ms	68.51 ms (+0.57%)
Top-100	68.02 ms	68.4 1ms (+0.58%)

Table 4: Runtime overhead under different values of k.

Dataset	Building Time
SIFT10M	6.20min
DEEP10M	4.44min
GIST1M	0.61min
TEXT10M	8.65min

Table 5: ELP building time on different datasets.

7 Discussion

Hardware acceleration. Some query engines [23, 49] leverage GPUs to accelerate vector query processing. GPU acceleration works well for small datasets, but is not suitable for large datasets because of limited GPU memory size. Thus, CPUs are widely used in production for large datasets. Also, GPUs are more expensive than CPUs. Therefore, we focus on CPUs for implementation and evaluation in this paper. We remark that the design of Auncel is orthogonal to the underlying hardware. Auncel can be applied to GPUs or other specialized hardware for vector query processing.

Vector compression. Vector compression [50] is proposed to reduce the memory footprint of large datasets. It compresses high-dimensional vectors into low-dimensional space while maintaining the distance property. The number of dimensions in each vector is reduced after vector compression. This technique is orthogonal to Auncel since we only consider the distance property between different vectors. Auncel processes vector queries on a dataset, no matter how many dimensions each vector in the dataset has.

8 Related Work

ANN indexes. Many ANN indexes are proposed to improve query accuracy and reduce query latency, such as IVF [15, 25, 35], graph index [16, 51, 52] and locality sensitive hashing [18, 53–55]. These algorithms perform search on sampling data which trade accuracy for query latency. They leave the sampling size (e.g., top-n in IVF) to users and do not provide bounded performance. These ANN algorithms are orthogonal and complementary to Auncel, and we follow one of the state-of-the-art solutions, IVF, to build Auncel. Besides, ANN algorithms also have different system characteristics [56]. For example, the graph index is more efficient than IVF, but it needs extra memory to hold large graphs. An interesting direction for future work is to build a unified ELP for more ANN indexes and provide bounded performance according to user preferences. Some recent works [30, 57] focus on early stopping conditions of nearest neighbor search to reduce average query latency at a high accuracy, but they do not provide any error or time bound guarantees. With the proliferation of unstructured data and machine learning, ANN on the embedding vectors of unstructured data becomes a key component in many AI applications, such as recommendation [1–4], recognition [5–8] and information retrieval [9–11]. Recent industrial vector data management systems [23, 24, 58] are developed to meet the rapidly increasing demand of these AI applications. They typically build their query processing engines on top of Faiss [22]. As we integrate the Auncel prototype into Faiss, it is convenient for these systems to leverage Auncel to improve vector query processing.

Approximate query processing. Approximate query processing systems [19, 20, 59–61] have gained a lot of popularity due to the long latency of exact search. These systems all provide time or latency guarantees through probability statistics. However, none of them pays attention to unstructured data represented by vectors. BlinkDB [19] and Quickr [60] focus on structured data and approximate aggregation jobs while ASAP [20] focuses on approximate graph pattern mining. The probability statistics method fails to produce good results on vector queries with performance guarantees. Thus, we introduce a novel high-dimensional geometry theory tailored for vector queries in Auncel. GRASS [21] is a scheduler for approximation jobs in data analytics clusters to alleviate the straggler problem in a map-reduce framework. It is complementary to the distributed design of Auncel since our error and latency calibration mechanism is easy to be integrated into such cluster schedulers. Big data warehouses [62–64] are prevalent in modern cloud services, which provide high-performance query processing. To manage large-scale unstructured data, Auncel can be integrated into these query systems to provide bounded performance. Auncel bridges the gap between approximate query processing and vector queries.

9 Conclusion

We present Auncel, a vector query processing engine that provides bounded errors and latencies on very large unstructured datasets. Auncel exploits the geometric properties of high-dimensional space and the nature of vector query processing to build precise and lightweight ELPs. Auncel is a distributed solution that leverages probability theory to scale out with multiple workers. We demonstrate the performance of Auncel on a variety of datasets. Auncel significantly reduces query latency while meeting error or latency bounds, and scales to billion-scale datasets with latency reduction.

Acknowledgments. We sincerely thank our shepherd Anurag Khandelwal and the anonymous reviewers for their valuable feedback on this paper. This work is supported by the National Key Research and Development Program of China under the grant number 2021YFB3300700, the National Natural Science Foundation of China under the grant number 62172008, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and a research gift from Moqi. Xin Jin and Xuanzhe Liu are the corresponding authors. Zili Zhang, Chao Jin, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *WWW*, 2007.

[2] J. Suchal and P. Návrat, "Full text search engine as scalable k-nearest neighbor recommendation system," in

IFIP International Conference on Artificial Intelligence in Theory and Practice, 2010.

[3] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Recsys*, 2016.

[4] M. Grbovic and H. Cheng, "Real-time personalization using embeddings for search ranking at airbnb," in *ACM SIGKDD*, 2018.

[5] R. He, Y. Cai, T. Tan, and L. Davis, "Learning predictable binary codes for face indexing," *Pattern recognition*, 2015.

[6] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[7] L. Zheng, L. Shen, L. Tian, S. Wang, J. Wang, and Q. Tian, "Scalable person re-identification: A benchmark," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[8] X. Liu, W. Liu, H. Ma, and H. Fu, "Large-scale vehicle re-identification in urban surveillance videos," in *ICME*, 2016.

[9] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, "The rise of deep learning in drug discovery," *Drug discovery today*, 2018.

[10] A. C. Mater and M. L. Coote, "Deep learning in chemistry," *Journal of chemical information and modeling*, 2019.

[11] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, 2015.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, 2015.

[13] R. Blumberg and S. Atre, "The problem with unstructured data," *Dm Review*, 2003.

[14] "Eighty Percent of Your Data Will Be Unstructured in Five Years." https://solutionsreview.com/data-management/.

[15] A. Babenko and V. Lempitsky, "The inverted multi-index," *TPAMI*, 2014.

[16] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *TPAMI*, 2018.

[17] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM TOG*, 2008.

[18] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004.

[19] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *EuroSys*, 2013.

[20] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "ASAP: Fast, approximate graph pattern mining at scale," in *USENIX OSDI*, 2018.

[21] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming stragglers in approximation analytics," in *USENIX NSDI*, 2014.

[22] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, 2019.

[23] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, *et al.*, "Milvus: A purpose-built vector data management system," in *ACM SIGMOD*, 2021.

[24] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data," in *Proceedings of the VLDB Endowment*, 2020.

[25] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "SPANN: Highly-efficient billion-scale approximate nearest neighbor search," in *Neural Information Processing Systems*, 2021.

[26] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Neural Information Processing Systems*, 2015.

[27] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *VISAPP*, 2009.

[28] "Annoy." https://github.com/spotify/annoy.

[29] L. Boytsov and B. Naidan, "Engineering efficient and effective non-metric space library," in *SISAP*, 2013.

[30] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving approximate nearest neighbor search through learned adaptive early termination," in *ACM SIGMOD*, 2020.

[31] "yandex billion-scale datasets." https://research.yandex.com/datasets/biganns.

[32] M. Grohe, "word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data," in *ACM SIGMOD*, 2020.

[33] H. Chen and H. Koga, "Gl2vec: Graph embedding enriched by line graphs with edge features," in *Neural Information Processing Systems*, 2019.

[34] G. C. Tomas Mikolov, Kai Chen and J. Dean, "Efficient estimation of word representations in vector space," in *International Conference on Learning Representations (ICLR)*, 2013.

[35] D. Baranchuk, A. Babenko, and Y. Malkov, "Revisiting the inverted indices for billion-scale approximate nearest neighbors," in *ECCV*, 2018.

[36] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, 1979.

[37] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Neural Information Processing Systems*, 2017.

[38] "Intel oneAPI Math Kernel Library (oneMKL)." https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/api-based-programming/intel-oneapi-math-kernel-library-onemkl.html.

[39] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *USENIX HotCloud Workshop*, 2010.

[41] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *ICASSP*, 2011.

[42] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *TPAMI*, 2010.

[43] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," in *SISAP*, 2017.

[44] S. Harsha, W. George, A. Martin, B. Artem, B. Dmitry, C. Qi, D. Matthijs, K. Ravishankar, S. Gopal, S. Suhas, and W. Jingdong, "Billion-scale approximate nearest neighbor search challenge," in *Neural Information Processing Systems Competition Track*, 2021.

[45] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *IJCV*, 2001.

[46] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[47] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," in *CIKM*, 2013.

[48] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[49] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 839–848, 2018.

[50] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[51] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," in *Proceedings of the VLDB Endowment*, 2019.

[52] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," 2019.

[53] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the VLDB Endowment*, 2007.

[54] Y. Zheng, Q. Guo, A. K. Tung, and S. Wu, "Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index," in *ACM SIGMOD*, 2016.

[55] L. Gong, H. Wang, M. Ogihara, and J. Xu, "idec: indexable distance estimating codes for approximate nearest neighbor search," in *Proceedings of the VLDB Endowment*, 2020.

[56] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, 2020.

[57] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos, "Progressive similarity search on time series data," in *BigVis*, 2019.

[58] W. Yang, T. Li, G. Fang, and H. Wei, "Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension," in *ACM SIGMOD*, 2020.

[59] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ACM ASPLOS*, 2015.

[60] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding, "Quickr: Lazily approximating complex adhoc queries in bigdata clusters," in *ACM SIGMOD*, 2016.

[61] A. P. Iyer, A. Panda, S. Venkataraman, M. Chowdhury, A. Akella, S. Shenker, and I. Stoica, "Bridging the gap: towards approximate graph analytics," in *SIGMOD GRADES-NDA*, 2018.

[62] "Google BigQuery." https://cloud.google.com/bigquery/.

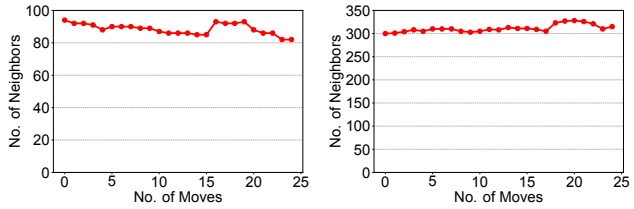
[63] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[64] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *ACM SIGMOD*, 2015.

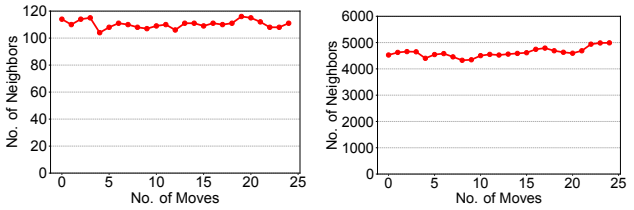
[65] "CMU CStheory-infoage chap1-high-dim-space." https://www.cs.cmu.edu/~venkatg/teaching/CStheory-infoage/chap1-high-dim-space.pdf.

A Appendix

A.1 Validation of Local Uniform Distribution



(a) cp_1 in SIFT10M. (b) cp_2 in SIFT10M.



(c) cp_3 in DEEP10M. (d) cp_4 in DEEP10M.

Figure 17: Validation of local uniform distribution.

We emphasize that the assumption about data distribution is *local* uniform distribution rather than uniform distribution. It is because the real-world items are distributed together smoothly. We perform a measurement to validate that vectors conform to local uniform distribution in widely-used unstructured datasets. We randomly pick a central point (cp) in the geometric space and let it do a random walk. Each time cp moves a small distance in either direction, the number of database vectors within a small radius (r) from cp is counted through a sweep. In this measurement, we pick two initial central points for SIFT10M and DEEP10M, respectively. Figure 17 shows the dynamics of the number of cp's neighbors and how it changes when cp moves. Within 25 moves (i.e., a relatively small local scope), the number of cp's neighbors fluctuates no more than 14% in the four examples. Consequently, it is reasonable to conclude the local uniform distribution.

A.2 Analysis of Formula 10

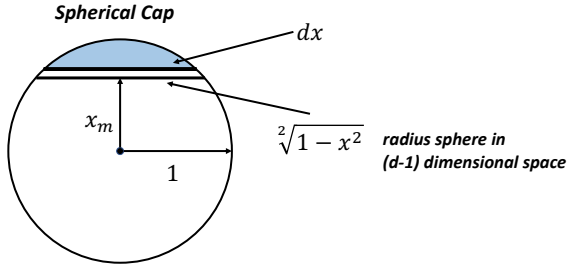


Figure 18: Unit spherical cap.

Let db_m represent the distance between the query vector q and the boundary of C_1, C_m. We leverage cosine theorem to calculate db_m by three anchor vectors, q and the centroid vectors of C_1, C_m, where the boundary is the mid-vertical plane of the line segment connecting the two centroids.

Since only the volume ratio is considered, we simplify the model into unit sphere, which substitutes $\lambda_i(j)$ with unit element (1) and db_m with $x_m = \frac{db_m}{\lambda_i(j)}$ after processing C_i. Thus, $\frac{V(P_j(m))}{V(B(\lambda_i(j)))}$ is identical to the ratio of spherical cap's volume, $V(sc)$ to the volume of the unit sphere as Figure 18 shows. According to properties of high-dimensional space [65], the volume of the unit sphere, $Un(d)$ in d-dimensional space is

$$V(Un(d)) = \frac{\pi^{\frac{d}{2}}}{\frac{d}{2}\Gamma(\frac{d}{2})}.$$

As for the spherical cap, the exact volume calculation is time-consuming through definite integral. To guarantee the bounded error, it is sufficient to provide an upper bound formula of the ratio. Thus, we derive the following lemma.

Lemma A.1 *For any $0 \le x_m \le 1$, the fraction of the volume of the hemisphere above the boundary (spherical cap, sc) is $V(sc)$. The unit sphere's volume is $V(Un(d))$ in d-dimensional space ($d \ge 3$). We conclude that $\frac{V(sc)}{V(Un(d))} \le a \times \arccos(x_m)$, where a is a number corresponding to d.*

Proof. The surface area of the intersection of the boundary and the sphere is

$$(1-x^2)^{\frac{d-1}{2}} V(Un(d-1)).$$

Thus, the volume of sc is:

$$V(sc) = V(Un(d-1)) \int_{x_m}^{1} (1-x^2)^{\frac{d-1}{2}} dx.$$

Now,

$$\begin{aligned} \frac{V(sc)}{V(Un(d))} &= \frac{V(Un(d-1))}{V(Un(d))} \int_{x_m}^{1} (1-x^2)^{\frac{d-1}{2}} dx \\ &\le a \times \int_{x_m}^{1} (1-x^2)^{\frac{d-1}{2}} dx. \end{aligned}$$

As for the upper bound, since $0 \le (1-x^2) \le 1$ and $-\frac{1}{2} < 0 < \frac{d-1}{2}$, we have $(1-x^2)^{\frac{d-1}{2}} \le (1-x^2)^{-\frac{1}{2}}$. Due to the properties of definite integral,

$$\int_{x_m}^{1} (1-x^2)^{\frac{d-1}{2}} dx \le \int_{x_m}^{1} (1-x^2)^{-\frac{1}{2}} dx$$

Thus,

$$\frac{V(sc)}{V(Un(d))} \le a \times \int_{x_m}^{1} (1-x^2)^{-\frac{1}{2}} dx = a \times \arccos(x_m).$$

The lemma is concluded by the above proof, and we fit a by sampling queries offline.

Above all, one of U is $\sum_{m=i+1}^{l} \arccos(x_m)$ in Formula 10 where $a \times U$ represents the upper bound of $\frac{\sum_{m=i+1}^{l} V(P_j(m))}{V(B(\lambda_i(j)))}$.

Arya: Arbitrary Graph Pattern Mining with Decomposition-based Sampling

Zeying Zhu[⋆*], *Kan Wu*[†*], *Zaoxing Liu*[⋆]
[⋆]*Boston University,* [†]*University of Wisconsin-Madison*

Abstract

Graph pattern mining is compute-intensive in processing massive amounts of graph-structured data. This paper presents Arya, an ultra-fast approximate graph pattern miner that can detect and count arbitrary patterns of a graph. Unlike all prior approximation systems, Arya combines novel graph decomposition theory with edge sampling-based approximation to reduce the complexity of mining complex patterns on graphs with up to tens of billions of edges, a scale that was only possible on supercomputers. Arya can run on a single machine or distributed machines with an Error-Latency Profile (ELP) for users to configure the running time of pattern mining tasks based on different error targets. Our evaluation demonstrates that Arya outperforms existing exact and approximate pattern mining solutions by up to five orders of magnitude. Arya supports graphs with 5 billion edges on a single machine and scales to 10-billion-edge graphs on a 32-server testbed.

1 Introduction

Graph-structured data have been used to represent relationships between entities in various domains, ranging from social networks [23, 39], financial transactions [46, 56], and knowledge bases [4]. There are two main categories of tasks in analyzing graphs: *graph computation* and *graph pattern mining*. Graph computation obtains various properties of a graph, such as PageRank [59] and connected components [41]. Graph pattern mining is more compute-intensive as it discovers structural patterns (i.e., subgraphs), such as motif finding [16, 57], frequent subgraph mining (FSM) [9, 73], and clique counting [21, 40]. These mining tasks are used in various applications, such as counting patterns of financial fraud [1] and detecting suspicious activities on social networks [8].

With graph data reaching multi-billion scales [7, 76], there is an increasing need to mine complex patterns to understand complicated internal relationships [22, 24, 29]. While many graph frameworks have been developed over the years based on various system and algorithmic optimizations, mining complex patterns (e.g., more than 5-vertex) in large graphs remains challenging. The fundamental reason is that pattern mining requires traversing and computing over large intermediate candidate sets, which grow exponentially with the graph size and pattern complexity. For instance, a recent high-performance mining engine, GraphPi [64], needs several hours to mine a six-vertex pattern in a graph of 1.2 billion edges, using the world's top-10 supercomputer with 1024 compute nodes (24,576 cores). Other general-purpose graph mining solutions, such as Peregrine [45] and Fractal [33], need more than a day to mine a six-vertex pattern even in small graphs (e.g., 1 million edges) on a 4-server testbed (see details in §7).

To reduce the underlying mining complexity, sampling-based approximation approaches have been proposed, e.g., ASAP [44] leverages a neighborhood sampling approach to approximate the pattern occurrences. Unfortunately, existing sampling-based approaches come with two common issues: (1) When pattern complexity (the numbers of vertices and edges) grows, the number of required sampling algorithm trials (called samplers) increases significantly (e.g., 10^{15} in mining 5-cliques in a billion-edge graph as shown in §2.1), making it infeasible to mine complex patterns in large graphs. (2) Systems like ASAP require developers to define distinct ways to sample a pattern to cover all possible occurrences. It might be easy for simple patterns like triangles: we can randomly pick the first edge, sample the second edge among the first edge's neighbors, and wait for the third edge to close the triangle. But it is challenging to figure out how to sample complex patterns as there are many distinct sampling ways.

In this paper, we present **Arya**, an approximate mining system that can scale to ultra-large graphs (e.g., 10 billion edges) and mine complex patterns (e.g., 11-vertex). In Arya, we tackle a general approximate mining problem: *Given an input graph, output the approximated occurrences of an arbitrary subgraph*. In many applications, an almost-correct result is sufficient, and the processing time is the key. For instance, a fintech company estimates the frequency of certain complex patterns to quantify the trends of online crime and fraud [24]. The chains of (money laundering) transactions form special patterns and estimating the occurrences of such patterns will help banks and companies to evaluate their operational risks.

Given that designing faster pattern sampling algorithms is theoretically difficult [35], Arya takes a new avenue to ***reduce the complexity of the pattern to be sampled***. Inspired by theory advances in graph sampling [14, 34, 36] and graph decomposition [18], our contribution is to bring theory into practice by an end-to-end system design that explores the algorithmic potential of approximate graph mining (e.g., systems

[*]Equal contribution.

design and optimizations, query accelerations) to meet user requirements (e.g. mining arbitrary patterns and error-latency profiles). Backed by decomposition theory, Arya significantly reduces the inherent approximation complexity of complex patterns in large dense graphs if the pattern can be properly decomposed. Arya has two main components: (1) a pattern decomposer that decomposes a complex pattern into a set of unique simple subpatterns and (2) a parallel estimation engine that generates a number of *samplers* for decomposed subpatterns and constructs an estimated frequency for the original complex pattern.

When designing our pattern decomposer, we need to determine an optimal decomposition of a complex pattern into simpler subgraphs such that it is sufficiently easier to sample the decomposed subpatterns and reconstruct the result for the original pattern. In Arya, we leverage the recent edge-cover-based graph decomposition [18]. The analysis shows that by computing the optimal fractional edge cover of a complex pattern (see §2.2), we can decompose the pattern into a unique collection of unique vertex-disjoint odd cycles and stars, which can be significantly easier to sample than the original pattern. With decomposition, Arya also has unique ways to sample patterns, alleviating the need to explore different sampling methods. Even if a pattern is too simple to be decomposed (e.g., 2-star), Arya performs no worse than existing sampling-based systems.

Once a pattern is decomposed, we build a parallel sampling engine to estimate the pattern occurrence by sampling cycles and stars separately. By extending the edge sampling theory from [15, 18], we build *odd cycle sampler* and *star sampler* with massive parallelism and construct the sampler for the original pattern. Each sampler is essentially a sampling trial aims to find *one instance* of the pattern with a fixed probability p: it merges the sampled odd cycles and stars and tests the remaining edges to find a potential pattern. If the sampled pattern can be formed, the sampler outputs $1/p$; otherwise zero. With a sufficient number of independent samplers, we can obtain an estimated pattern count by averaging the outputs from all samplers (linearity of expectation). To estimate the number of samplers required to achieve the desired accuracy, Arya introduces a heuristic inspired by ASAP [44] to build the Error-Latency Profile (ELP), which takes an error target as the input and infers relevant parameters (e.g., the number of samplers) to configure the graph miner based on bootstrapping (from a small sample of the graph).

With graph decomposition, edge sampling, and a series of system optimizations (e.g., probability-aware scheduling and sampler caching), Arya outperforms *any* existing graph mining systems in scalability. We implement Arya using Memcached to store graph data structures and optimize the most frequent queries to it (e.g., neighbor edges and vertices). We deploy Arya onto three computing scenarios: a single server, a single server with persistent memory, and a cluster with 4 to 32 servers. Our evaluation demonstrates that Arya outperforms GraphPi, a state-of-the-art supercomputer-based graph miner, by up to 5 orders of magnitude while incurring a less than 5% error. In addition, Arya outperforms the state-of-the-art approximate mining system [44] by up to 145× and scales to graphs with multi-billion edges. For example, Arya can mine a complex 7-vertex pattern in a 5-billion-edge synthetic graph in several seconds. We open-source Arya and datasets in https://github.com/Froot-NetSys/Arya.

In this paper, we make the following contributions.

- We present Arya, an approximate graph miner that scales to large graphs and complex patterns, leveraging advanced graph decomposition theory and edge-based sampling. (§3)
- We introduce techniques to quickly sample decomposed subpatterns and reconstruct the original pattern for approximation, based on the latest cycle/star sampling algorithms. (§4)
- We extend Arya to various distributed settings for heterogeneous graph processing scenarios. (§5)
- We show that Arya mines complex patterns in graphs with 5 billion edges using a single machine and 10 billion edges using multiple machines, a scale that even supercomputer-based systems failed to achieve. (§6,§7)

2 Background and Motivation

In this section, we discuss the background of graph pattern mining and approximate mining algorithms. We then describe the graph decomposition theory that we leverage.

2.1 Graph Pattern Mining

Problem Definition. Graph pattern mining is to find instances of a given pattern in a graph or set of graphs. A pattern is an arbitrary subgraph, which represents user-defined properties attached to the edges and vertices. Pattern mining algorithms aim to find all subgraph instances (called *embeddings*) that match a given pattern of interest. Such matching is usually done via iterating all subgraphs that are *isomorphic* to the input pattern, which is known to be NP-complete. At a high level, the compute complexity of an *exact* pattern mining algorithm is associated with the need to iterate over all possible embeddings in the graph.

Approximate Graph Pattern Mining. Given the search complexity of exact mining algorithms, approximation-based approaches become promising. Approximate analytics is widely used in solving complex big data [58], network telemetry [38, 63], and database problems [11], typically with significantly lower resource overheads. A common idea to perform approximation is to sample a *subset* of the input data uniformly at random and perform analytical tasks over the sampled data. For instance, uniform sampling (e.g., NetFlow and sFlow) has been widely used in monitoring network flows.

• **Advanced pattern sampling:** There is a large body of theoretical work on designing sampling algorithms [44, 47, 60] to mine specific patterns such as triangles and cliques. The

main difference between these algorithms is the way to sample a specific pattern. Intuitively, if a sampling approach can sample a pattern with a higher probability, a smaller number of samplers is required to achieve high accuracy (and thus a shorter completion time). For instance, *neighborhood sampling* is used in ASAP [44]. The main idea of neighborhood sampling is to continuously sample neighbor edges until the pattern can be formed. In mining triangles, each neighborhood sampler starts by sampling an edge uniformly at random and then sampling the second edge from the neighbor edges of the first edge. If there is a third edge in the remaining edges that can form a triangle with the existing two edges, this sampler successfully samples a triangle. Compared to the standard sampling approach that has a $1/m^3$ probability (sampling three edges uniformly at random), neighborhood sampling has a larger probability $1/m \cdot c$ to sample a triangle, where c is the number of neighbor edges of the first edge.

• **Limitations of existing sampling-based systems.** While sampling-based approximation is promising in reducing the computation in graph pattern mining compared to exact algorithms, two significant issues remain:

(1) Scalability remains an issue in mining complex patterns in (dense) large graphs for systems like ASAP [44]. In particular, the number of required samplers can be prohibitively large, leading to high computation and memory costs. Taking neighborhood sampling as an example, it requires $O(\frac{m^2}{f_p})$ to estimate 4-vertex patterns and $O(\frac{m^3\Delta}{f_p})$ for 5-vertex patterns, where Δ is the maximum degree in the graph and f_p is the occurrence of the pattern. From 4-vertex to 5-vertex, the computation complexity is increased by up to $O(m\Delta)$, where the number of edges m can be large (e.g., Twitter graph [50] has 1.2 billion edges). This complexity will be increased dramatically for more complex patterns, and this observation is confirmed by ASAP that they cannot scale to more than 5-vertex patterns in their large graphs evaluated in their 16-server testbed.

(2) Using ASAP, one needs to define how to sample a pattern using their neighborhood API [44]. While it is straightforward to sample simple patterns (e.g., triangle), sampling complex patterns is challenging (e.g., triangle-triangle). For instance, there is only one way to sample a triangle as described above, but there are multiple ways to sample a triangle-triangle (two triangles connected by an edge). It is challenging for developers to figure out all possible samplers when the pattern is even more complex. If samplers do not cover all possible ways to sample patterns, we can see severe underestimations in the final results.

2.2 Approximation with Graph Decomposition

The goal of our system, Arya, is to explore a scalable solution for mining complex patterns in large graphs. One potential way to improve the scalability is to continuously design and develop new sampling techniques that can sample patterns with higher probabilities. However, the fundamental limitation in this direction is that, we need to sample at least these many of edges and vertices to form a pattern, which implies an upper bound on how large the sampling probability can be on a specific pattern [35].

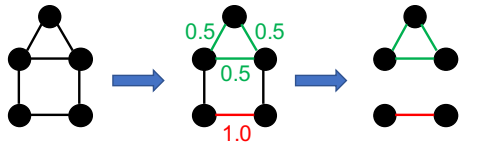


Figure 1: **An example of decomposing a pattern.**

Instead, Arya aims to take another road to improve the scalability of approximate pattern mining. The question we ask is that ***if we cannot improve the sampling technique further, can we instead reduce the complexity of the pattern to be sampled?*** We find that, graph decomposition theory [30], which is a powerful tool to reduce the complexity of graph matching and coloring problems, can also be applied jointly with existing sampling techniques to reduce the pattern mining complexity. Recent graph theory advances [18] made a contribution to proving that any subgraph can be decomposed as a collection of vertex-disjoint odd cycles and stars by solving an optimal fractional edge cover problem. This decomposition is a desired property for our purposes, as no matter how complex the patterns are, they can be decomposed into cycles and stars (tackling issue (2)), and these two subpatterns are "easier" cases to be sampled (tackling issue (1)).

Definition 1 ([18]). *Denote the fractional edge cover of a pattern P as $P(V_P, E_P)$, where V_P is the vertex set and E_P is the edge set. $P(V_P, E_P)$ is a mapping $\phi : E_P \rightarrow [0,1]$ such that for each v in V_P, $\sum_{e \in E_P, v \in e} \phi(e) \geq 1$. The fractional edge cover number is $\sum_{e \in E_P} \phi(e)$.*

The *optimal* fractional cover is to find a subset of edges in the pattern (covering all vertices) that minimum the fractional edge cover number $\rho(P)$ (i.e., min $\sum_{e \in E_P} \phi(e)$). Intuitively, the key insight (detailed proofs in [18]-A.2) is that for any pattern P, there always exists an optimal fractional cover that maps weights 0.5 to edges that can form odd cycles and maps weights 1.0 to edges that do not belong to any odd cycle (in the cover). This result is powerful because it ensures that we can decompose arbitrary patterns into a collection of odd cycles and stars. Moreover, the analysis further proves that this decomposition reaches optimal bounds for sampling arbitrary patterns, which strengthens our confidence in this decomposition. Thus, we need to calculate the following linear programming (LP), and construct odd cycles with edges of weights 0.5 and stars with edges of weights 1.0:

$$Minimize \sum_{e \in E_P} \phi(e)$$

$$s.t. \sum_{e \in E_P : v \in e} \phi(e) \geq 1, \forall v \in V_P$$

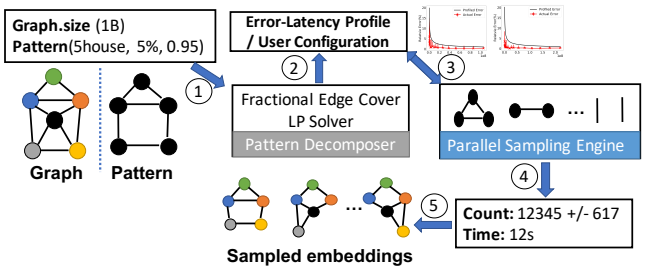


Figure 2: **Overall architecture of Arya.**

Figure 1 shows a 5-house example to find the optimal fraction cover as (b), and we then decompose 5-house into a three-cycle (if the weight ϕ of each edge is 0.5) and a 1-star (if the weight is 1.0).

Challenges. While the decomposition technique has provable theoretical guarantees, there are several challenges in building a general, distributed system for large-scale approximate graph mining. First, to be of practical use, once a pattern is decomposed into subpatterns, we need a distributed processing engine that optimizes the performance of running a large number of (subpattern) samplers, e.g., how to schedule the execution of the samplers and scale to distributed settings with optimized communication. Second, the graph theory we leverage assumes homogeneous edges and vertices while real-world graphs are often associated with properties. Therefore, the mining queries require *predicate matching* that envisions the technique to be property-aware. Finally, as an approximate processing system, we need to allow users to trade-off accuracy for running time. We need to understand the relationship between errors and actual running time in both single and distributed settings.

3 Arya Overview

We design Arya, an approximate graph pattern mining system leveraging decomposition-based graph sampling. Figure 2 demonstrates the overall architecture of Arya. Arya provides three operating modes to adjust to different compute scenarios: (1) *Single machine* mode that is optimized for local edge and vertex queries; (2) *distributed with replicated graphs* mode where the graph dataset is replicated entirely to multiple machines; (3) *distributed with partitioned graphs* mode that is developed with distributed KV-store (e.g., Memcached) to support arbitrarily partitioned graphs across machines. We provide an overview of different components in Arya and how users can leverage our system to perform approximate mining tasks for arbitrary patterns.

Arya workflow. Arya allows users to mine arbitrary patterns in a graph. As an approximate engine, a user can specify an input pattern and an error budget as follows:

- **Input pattern P:** The user defines an arbitrary subgraph P of the input graph as the pattern to mine in Arya. The user specifies P (in a text file) by adding a list of edges that form the pattern. This pattern will then be decomposed into a collection of odd cycles and stars via Arya decomposer. Unlike prior approximate mining systems, the user does not need to define the ways to sample the pattern as Arya will always sample stars and odd cycles for arbitrary patterns.
- **Error budget ε:** The user specifies an accuracy target by setting an error budget ε (e.g., 5%) with a confidence interval (e.g., 95%). Arya is expected to output an approximate result within ε error in time T.

After specifying the input pattern and the error budget, Arya first decomposes the pattern via the fractional edge cover LP solver as ①. For building an ELP, the ELP engine will return a required number of samplers (and time) for the error budget ε as ② using the parallel sampling engine ③; or the user directly specifies the number of samplers to run. Once the user approves the estimated time, the sampling engine will perform the approximation and return the estimated count with the actual run time as ④. The sampling engine also finds a set of sampled embeddings as ⑤.

Sampling vs. Enumeration. While Arya shows tremendous performance improvements on mining various complex patterns, we observe that Arya works best with the following two assumptions: (1) The graph is *dense* such that there are many pattern occurrences. (2) The decomposed subpatterns need only a few remaining edges to complete the pattern. For (1), it is a fundamental argument between deterministic enumeration-based approaches and sampling-based approaches. When the graph is sparse, it is challenging to find a pattern via random sampling (like "a needle in a haystack"), while it is a better case for enumeration. For (2), if the decomposition of a pattern breaks too many edges, each Arya's sampler spends extensive efforts on searching the remaining edges to complete the pattern, degrading the execution performance. If these two assumptions do not hold, Arya may experience many failed trials and thus requires more samplers. Therefore, while Arya supports arbitrary pattern mining, the actual runtime depends on the above two key conditions.

4 Basic Design

We now present how Arya enables ultra-fast graph pattern mining by combining pattern decomposition and edge-based sampling as a theoretical foundation. We focus on the single machine design in this section and extend it to distributed settings in the next section.

4.1 Pattern Sampling Algorithms

By leveraging graph decomposition theory (§2.2), a pattern will be represented as a set of vertex-disjoint odd cycles and stars. Our pattern sampler is to sample the relevant odd cycles and stars from the graph and check if there exist remaining edges that complete the pattern. Thus, we introduce two sampling algorithms based on [15, 18] to sample them separately, then use them to construct the pattern. In the algorithms, we denote an odd cycle with $2k+1$ edges as C_{2k+1} ($k \geq 1$) and a star with l petals as S_l.

Algorithm 1 Odd Cycle Sampler

1: **Input**:
2: Graph $G = (V, E)$ where $|E| = m$, an odd cycle C_{2k+1}
3: **Output**:
4: An instance of C_{2k+1} and sampling probability $Pr[C_{2k+1}]$ or 0
5: Sample an edge $e_1 = (u_1, v_1)$ from graph such that $d(u_1) \leq d(v_1)$. ▷ *sample first edge with an order*
6: Sample $k-1$ edges $e_2 = (v_2, u_2), \ldots, e_k = (v_k, u_k)$ with replacement from G. ▷ *sample rest edges as the cycle skeleton*
7: Sample a vertex n from the neighbors of u_1. ▷ *One more*
8: Check if there are remaining edges $(v_1, u_2), \ldots, (v_{k-1}, u_k), (v_k, n)$ in G. If succeeds, output C_{2k+1} and $Pr[C_{2k+1}]$; otherwise, 0.

Odd Cycle Sampler. At a high level, our goal here is to sample odd cycles and we can adopt *any* cycle sampling algorithms (e.g., neighborhood sampling [61] used in ASAP [44]). In Arya, we attempt to introduce the algorithm shown in [31] and [18]: first uniformly sample k edges (with the first edge having an order based on the degree), a neighbor edge of the first edge, and then test if there are remaining k edges in the graph to complete a cycle. Compared to ASAP, it is easier for Arya's sampler to sample an odd cycle using edge sampling since the probability of forming an odd cycle is higher. However, our empirical results show that the two algorithms are comparable for cycles (§7). We adopt this algorithm for two reasons: (1) *Easy to support longer odd cycles:* compared to neighborhood sampling, most of its sampling phase involves only random edge sampling, and thus no need to store nested neighborhood states (e.g. neighbors of neighbors); (2) *Easy to optimize performance* with the hash-based graph structures presented in §4.2 to accelerate queries.

Specially, we present the odd cycle sampler in Algorithm 1. First, we sample a special (directed) edge $e_1 = (u_1, v_1)$ whose first vertex does not have a larger degree than that of the second vertex (Line 5). Second, we sample another $k-1$ edges uniformly at random with replacement (Line 6). Third, we sample a neighbor edge (n, u_1) of u_1 as the last hoop of the cycle (Line 7). Finally, we need to test if there are remaining k edges in the graph to complete the odd cycle as $(u_1, v_1), (v_1, u_2), \ldots, (u_k, v_k), (v_k, n), (n, u_1)$. Since each sampling step is independent, the overall probability to sample this odd cycle is $Pr[C_{2k+1}] = Pr[e_1] \cdot Pr[e_2] \cdots Pr[e_k] \cdot Pr[n] = \frac{1}{m}(\frac{1}{2m})^{k-1}\frac{1}{d(u_1)}$.

Star Sampler. Intuitively, a star consists of a center vertex and a few petals, and sampling a star can be straightforward. There is a broad spectrum of theory work using star samplers as a main or subroutine in various applications (e.g., sparsification, clustering, and matching) [15, 18, 49]. Here, we adopt a common weighted star sampler as in Algorithm 2 (e.g., [15, 18]). We start by selecting a vertex v_1 with probability proportional to its degree $d(v_1)$ (i.e., $\frac{d(v_1)}{2m}$). This step is to sample centers that are more likely to form stars with multiple petals. In practice, we optimize the query performance by performing an edge sampling as the way to sample v_1. For instance, if a graph has 50 edges and a vertex v_1 has a degree of 10, randomly sampling an edge is equivalent to sampling a vertex that is v_1 with a probability of $1/10$ because there are 10 edges in the graph that are incident to v_1. This sampled vertex is used as the star center, and we will then sample l vertices from v_1's neighbors uniformly at random without replacement. We will either find such an l-star or return zero from this step. Overall, the probability to sample a star is $Pr[S_l] = Pr[v_1] \cdot Pr[petal_vertices] = \frac{d_{v_1}}{2m}\binom{d_{v_1}}{l}$.

Algorithm 2 Star Sampler

1: **Input**:
2: Graph $G = (V, E)$ where $|E| = m$, a star S_l with l pedals
3: **Output**:
4: An instance of S_l and sampling probability $Pr[S_l]$ or 0
5: Sample a vertex $v_1 \in V$ with probability $d_{v_1}/2m$ for any $v_1 \in V$. ▷ *weighted center vertex sampling*
6: Sample l petal vertices uniformly at random from the neighbors of v_1 without replacement . ▷ *sample pedals to complete*
7: Output S_l and $Pr[S_l]$ if succeeds; otherwise, 0.

Approximation for the Original Pattern. We can sample an embedding of the original input pattern if and only if all associated odd cycle samplers and star samplers find their instances. If any sampler does not successfully form their cycle/star embedding, we will terminate this sampler for the original pattern and return zero. Once all the decomposed pattern samplers finish, we need to *reconstruct* the original pattern P by *merging* the cycles/stars and *testing* if the remaining edges between the cycles and the stars do exist in the graph to form the pattern. This is the last step of the whole pattern sampler. During the testing, we list all the possible remaining edges and check if they exist in the graph until there are enough checked edges to complete the pattern. If a complete pattern can be formed with o odd cycles and s stars, the probability of the sampler is $Pr[P] = Pr[C^1] \cdots Pr[C^o] Pr[S^1] \cdots Pr[S^s]$. A sampler outputs $R[P] = \frac{1}{Pr[P]}$ if a pattern instance is found; otherwise, it outputs $R[P] = 0$.

In summary, Arya runs a number of such pattern samplers in parallel based on ELP. In the final "reduce" phase, if there are n samplers and sampler i returns result $R_i[P]$, Arya returns $\frac{\sum_{i=1}^{n} R_i[P]}{n}$ as the final result. This is because Arya's pattern sampler finds any possible embedding P_i with $Pr[P_i]$. Suppose there are $\#P$ embeddings of the pattern, the expected output of a sampler is $E = \sum_{i=1}^{\#P} \frac{1}{Pr[P_i]} \cdot Pr[P_i] + 0 \cdot (1 - Pr[P_i]) = \#P$. With more samplers, the average of the sampler outputs will be closer to the expected value $\#P$. Therefore, Arya trades more samplers for better accuracy.

4.2 Sampler-Friendly Graph Structures

We observe that both odd cycle and star samplers involve a number of specific queries to the graph data, which are the major computation bottlenecks in the sampler runtime. To improve sampler performance, we summarize the most frequent runtime queries and provide simple, yet effective data structures to accelerate them.

- **Edge sampling**: sample an edge e uniformly at random from the graph.
- **Neighbor sampling** (v,i): perform a neighborhood sampling on v to obtain the i-th neighbor edge ($i \leq d(v)$) and check what vertex is associated with this edge.
- **Degree checking** v: obtain the degree of vertex v.
- **Edge checking** (u,v): check if vertices u and v form an edge in the graph.

Given the nature of the queries above in randomized algorithms, we should optimize data stores to accelerate processing. For instance, an edge sampling query can be implemented unoptimized as drawing a random number from the edge list $[1 \ldots m]$ and taking a linear traversal to find the exact edge. Instead, we use two auxiliary data stores for performance improvements: (1) An *edge array* that is grouped by vertex with the requirement that all neighbor edges of a vertex are stored consecutively. We observe that many public graph datasets are already stored in this order [6]. (2) A *hash table* that maps vertices to their metadata. Specifically, each vertex has an integer as its ID and its metadata containing the vertex's degree and the starting index of the vertex in the edge array.

4.3 Advanced Pattern Mining Features

Beyond approximating the occurrences of a pattern in a graph, Arya provides users with several advanced features.

Predicate matching. A common way of representing the graph data is in the form of *property graph*, where user-defined properties are attached to the vertices and the edges. Thus, the real-world queries to a property graph may require to match patterns satisfying certain predicates. For instance, a predicate matching query can ask for the count of all 5-House patterns in the graph where all edges are associated with an organization or all vertices meet a certain type.

Arya supports three types of predicates—*at-least-one*, *at-least-percentage*, and *all*. For "at-least-one" or "at-least-percentage" predicates, users are asked to specify a predicate that is applied to at least one (or a percentage or all) of the edges or vertices. Arya can support these predicates since a new property "subgraph" can be maintained and the same sampling techniques can be applied. To perform a predicate matching task, we introduce a *conservative sampling* stage. We first create an auxiliary graph that contains only the edges or the vertices that satisfy the predicate. The odd cycle and star samplers will sample the first (or a percentage of) edges or vertices from the auxiliary graph and then perform the rest of the sampling in the original graph. Different from the non-predicate-case, we need to refine the sampling rates based on the number of matched edges or vertices stored in the auxiliary data store. In a simple example, the probability of sampling the first edge uniformly at random is not $1/m$ but $1/m^*$, where m^* is the number of edges that satisfy the predicate. More details can be seen in Appendix A.

Intermediate state caching (e.g., Motifs). We consider two scenarios when some intermediate states can be cached and reused. (a) First, when running multiple mining tasks on the same graph, different patterns may share one or more decomposed subpatterns (i.e., odd cycles and stars). For instance, the decomposed 5-house and triangle patterns share a 3-cycle subpattern. Thus, Arya automatically caches the previous subgraph samplers (3 cycles) to reuse across patterns. (b) Second, some patterns share the same sampling steps in their samplers to form the patterns, e.g., one can sample any 4-motif patterns (except for 3-star) using two 1-star samplers with different remaining edges to complete the patterns. Thus, Arya does not need to sample each 4-motif pattern separately.

Algorithm 3 Error-Latency Profile (δ,ε)

Input: Original graph G with M edges, sampled subgraph g with m edges (with probability r), pattern P with p edges, error target ε, and confidence $1-\delta$.
Output: Number of estimators N_e for G

1: $avg_{last} \leftarrow \inf$, $range_{last} \leftarrow \inf$, $N_c \leftarrow 10{,}000$
2: **while** `True` **do**
3: Run Arya 3 times with N_c samplers on subgraph g
4: $avg_{cur} \leftarrow$ the average count of the 3 trials.
5: $range_{cur} \leftarrow$ the range (max - min) of the 3 trials.
6: $\tilde{\varepsilon} \leftarrow |avg_{last} - avg_{cur}|/avg_{cur}$
7: **if** $\frac{range_{last}}{avg_{last}} < 10\%$ and $\tilde{\varepsilon} < \varepsilon$ and $\frac{range_{cur}}{avg_{cur}} < 10\%$ **then**
8: $C \leftarrow \frac{N\tilde{\varepsilon}^2 avg_{cur}}{m^{\rho(P)}}$, $h \leftarrow avg_{cur}$
9: `Break`
10: $N_c \leftarrow N_c \times 2$, $avg_{last} \leftarrow avg_{cur}$, $range_{last} \leftarrow range_{cur}$
11: $N_e \leftarrow \frac{C \cdot M^{\rho(P)}}{\frac{h\varepsilon^2}{r^p}\delta}$ ▷ $\rho(P)$ is known given P

4.4 Error Latency Profile (ELP)

Arya allows users to tradeoff accuracy for result latency. As an approximation system, Arya needs to determine the number of samplers for expected errors and running time. Arya uses a heuristic to build the ELP. Our experiments in §7.3 demonstrate the accuracy of the ELP.

According to the mathematical analysis in [18, 30] using Chebyshev's inequality, decomposition-based sampling requires $O(\frac{m^{\rho(P)}}{\#P})$ estimators to provide a $(1\pm\varepsilon)$-approximation to the ground truth ($\#P$) for any $\varepsilon > 0$, where $\rho(P)$ is the minimum fractional edge cover number of pattern P. Furthermore, the actual number of required samplers is lower bounded by $\frac{Cm^{\rho(P)}}{\#P \cdot \varepsilon^2 \delta}$ with probability $1-\delta$ for some constant C. Thus, our goal is to estimate C for a particular graph and a pattern. We

achieve this by using a "sparsified" input graph: we uniformly sample a subgraph from the original graph with probability r (e.g., 30%), and determine an approximate number of needed samplers by running varying numbers of samplers and converging to a stable pattern count. The pseudocode is shown in Algorithm 3. Line 1 gives an initial number of samplers N_c to start with. For a given N_c, the algorithm runs Arya 3 times to obtain the average and range of 3 trails in lines 3 to 5. If the last and current range difference is small enough and when using the current average result as the ground truth, the last average estimated result is within the error target (line 7), we can exit, calculate an estimated C , and treat the current average result as the estimated ground truth h (line 8). Otherwise, ELP exponentially increases N_c (line 10) and proceeds again. Line 11 calculates N_e, the number of samplers for G, based on C and δ, ε, M and scaled $\#P$.

5 Scaling Arya to Distributed Settings

In this section, we introduce how Arya is scaled to multiple machines to support larger graphs and more complex patterns. Naturally, a graph store can be distributed in the following three ways: (1) distributed replicated graphs, (2) randomly partitioned graphs, and (3) arbitrarily partitioned graphs. Arya is designed to support all these configurations and arbitrarily partitioned graph is the most challenging one. We introduce several optimizations to improve Arya's scalability by up to 4.7× over the basic design.

5.1 Distributed Replicated Graphs

Replicated graphs are a common approach for serving input data in distributed graph mining systems, such as Fractal [33] and GraphPi [64]. In Arya, the compute can be distributed directly across the machines if the graphs are replicated. This is because each sampler is independent and each machine will be assigned a subset of required samplers to run on its multiple CPU cores/threads. Each thread takes one sampler at a time. Once the samplers are assigned, there is no communication between machines or samplers until the final aggregation of the results from all the samplers.

5.2 Distributed Partitioned Graphs

The second scenario is when graphs are partitioned to multiple machines. G-thinker [72] and Kudu [52] are example mining systems that assume graphs are partitioned among compute nodes. Similarly, graphs can also be separated from compute nodes in real-world scenarios. Meta, for example, has its own cluster of graph store RIPQ [68]. Arya assumes an API (e.g., *getedge(edgeID)*, *getAdjList(vertexID))* to access partitioned graphs and can work with either locally partitioned graphs (as in G-thinker) or remote graph stores.

In practice, many partitioning strategies are possible. ASAP [44] requires to partition the graph edges uniformly at random. Graph partition strategies often depend on the workloads (e.g., PageRank [59]). In addition, the graphs may need to be partitioned based on strategies to be compliant with security and privacy requirements, such as GDPR [3] and GDPR-Neo4j [5]). Unlike ASAP, Arya makes no assumptions about partitioning strategies.

Arya extends its design from replicated graphs to partitioned graphs, with one major challenge to overcome. In contrast to the replicated graph scenario, a graph is partitioned into slices to compute nodes; each node's samplers will have a potentially large number of random accesses to the graph data stored in other nodes. This poses significant scaling challenges for Arya on partitioned graphs: Arya will be constrained by network communication overheads. A single triangle sampler, for example, entails six graph queries (1 edge sampling, 3 degree checkings, 1 neighbor sampling and 1 edge checking). Each triangle sampler in the Friendster graph [74] generates around 6KB network traffic. To count triangles in Friendster with a 5% error, we will need at least 4 million samplers, which translates to 20 million graph queries and 23GB of network traffic. The comptation-communication ratio is approximately $c\frac{p}{(p-1)}$, where c is a constant depending on the pattern and graph, and p is the number of partitions. The detailed analysis is defered to Appendix C. While the intermediate state caching technique can help reduce communication costs, Arya introduces two more techniques for communication reduction: (1) probability-aware sampler scheduling and (2) batched sampling and communication.

Technique 1: Probability-aware sampler scheduling. A key observation we have is that different decomposed subpattern samplers (e.g., triangle vs. 2-star) have different probabilities to fail (not finding one). According to our Mico graph profiling, a 2-star sampler has a 0.5% failure probability while a triangle sampler has a 92% probability to fail. This is because simpler structures are more likely to be sampled than complex structures in a graph. Based on this observation, we can save communication overheads if we sample these likely-to-fail subpatterns earlier. A lot of such samplers will fail early and we can prune them without running other subpattern samplers.We note that after decomposition, each subpattern sampling occurs independently, and thus the order of subpattern sampling has no effect on the original pattern sampling's success/failure probability and overall accuracy. Taking the triangle-2star pattern as an example: for each pattern sampler, if sampling the triangle first, it is more likely to fail (92%) and there is no need to sample 2-star in 92% of the cases. Hence, we schedule subpattern samplers in the order of their sampling failure probabilities to achieve better performances.

To do so, we must address an important question: how do we know which subpattern samplers are likely to fail? The answer depends on the pattern and graph. Given the static graph, Arya first offline profiles failure probabilities of popular subpatterns (such as 2-star, triangle) in a small number of trials. Then each pattern counting task can query the failure probability profile for any subpattern samplers. When the failure probability of a subpattern is not in the

profile, we perform an online profile by letting the first set (e.g., 10%) of the samplers collect the failure probabilities information without early pruning. These probabilities will be used to schedule the remaining samplers. This technique is applied to all Arya versions.

Now, we analyze the overheads of Arya's offline and online profiles. The cost of offline failure profiling is minimal because the overheads are amortized by all queries to the graph, and profiling is limited to simple common subpatterns. For example, profiling simple 2- to 5-stars and triangle subpatterns for the Friendster graph takes only 220ms even for less than 5% error results, whereas a single 5-node pattern query to Friendster can take tens of seconds as we will show in Figure 5(b).

Arya's online failure probability profiling trades off early pruning opportunities in the query's first 10% samplers for better subpattern sampling order in its remaining 90% samplers. This approach produces runtime comparable to Arya with perfect subpattern sampling order pre-knowledge and significant improvements over Arya with the worst subpattern sampling order. We use 10% as we found it to be adequate for accurate simple subpattern failure probability profiling.

If the profiled failure probability is inaccurate (which is uncommon as subpatterns are simple patterns that are easy to estimate), Arya may use suboptimal subpattern sampling orders. In this worst case, probability-aware sampling performs similarly to the case of no early pruning.

Technique 2: Batched sampling/communication. Arya reduces overheads from the network stack by using batched sampling and communication. One Arya thread advances a batch of samplers at the same time (vs. progress one sampler until it finishes). When a graph query is required in a sampler, the thread buffers the query and pauses the sampler before moving on to the next sampler in the batch. When all of the samplers in the batch are waiting for graph queries, the thread will begin its batch communication with the graph store (i.e., send out the queries we buffered, for example, with Memcached multi-get).

6 Implementation

We build Arya for both single-machine and distributed graph computing scenarios. The pattern decomposition logic is implemented with Python, and the core components of Arya are written in C++ with 11K LOC. We open-source Arya at [2].

Pattern Decomposition. Arya takes an arbitrary pattern as input and outputs a set of stars and odd-cycles via a pattern decomposition logic. As discussed in §2.2, Arya will find the optimal fractional cover of the input pattern. We use `scipy` linear programming package to find the optimal cover: it takes only 900ms on a single server to decompose complex 20-vertex patterns and less than 400ms for less complex patterns that have fewer than 10 vertices. This running time is negligible compared to the total mining time.

Graph Sampler. There are three versions of sampling logic written in C++: single-machine, distributed replicated graph, and distributed partitioned graph. The former two versions access in-process graph stores, while the distributed partitioned graph version accesses remote graph stores (e.g., Memcached) via TCP. To parallelize the samplers, all Arya versions employ multi-threading. Single-machine version and distributed replicated graph version use the work-stealing algorithm dynamically scheduling computations. A communication thread distributes tasks when the total number of samplers is smaller than required and uses asynchronous communication primitives for work stealing. Worker threads return the results from a batch of samplers to the communication thread when they finish a task. Worker threads then execute the next batch of samplers. We can configure the granularity of a sampling task. For distributed partitioned graph implementation, the master process will initiate samplers on each machine and collect results from each machine when the sampling phase is completed. For evaluating ASAP in a fair setting, we implement ASAP graph samplers using Arya's system API (which is faster than Spark used in ASAP), including accessing the graph structures and performing edge- and node-related queries.

7 Evaluation

We evaluate Arya on a variety of open-source and synthesized graphs and compare it to the state-of-the-art approximate mining system (ASAP [44]) and exact mining systems (*Single-machine:* Peregrine [45], DwarvesGraph [26], AutoMine [54]. *Distributed:* Fractal [33], GraphPi [64], G-Thinker [72], Kudu [52]). Our experiments demonstrate:

- Compared to single-machine exact mining systems, Arya is up to $105,365\times$ faster than Peregrine within a 5% loss of accuracy when counting complex patterns. To the best of our knowledge, Arya is the first system capable of mining complex patterns (>6 vertices) on giant graphs.
- Compared to distributed mining systems, Arya outperforms Fractal by $62\times$ to $56,842\times$ and GraphPi by up to $988\times$.
- Compared to ASAP, Arya can mine arbitrary patterns in both single-machine and distributed settings. Arya is up to $145\times$ times faster on a single machine and $55\times$ faster in distributed settings, with a 5% error target.
- Arya's probability-aware scheduling and batched sampling techniques are effective for speeding up Arya by up to $4.7\times$.

Datasets and baselines. We compare Arya to state-of-the-art systems using a set of representative graphs as in Table 1. We obtain the ground truth via running deterministic mining systems such as GraphPi [64] and Peregrine [45][2]. For datasets used in distributed partitioned graph experiments, the graph

[2]We are unable to get the mining results of P3 and P4 patterns [64] in the Twitter graph because all tested deterministic miners including GraphPi experienced system crashes or their running time exceeded 24 hours.

Size	Graph	Nodes	Edges	Degrees
Medium	Mico [37]	100,000	1,080,298	22
	Youtube [51]	1,134,890	2,987,624	8
Large	Twitter [50]	41.7 million	1.2 billion	36
	Friendster [75]	65.5 million	1.8 billion	28
Giant	RMAT-5B	500 million	5 billion	
	RMAT-10B	1 billion	10 billion	

Table 1: **Graph datasets used in the evaluation.** *We use the RMAT model [48] to generate small and giant synthetic graphs. In the RMAT model, we used default parameters (a,b,c,d) as (0.44,0.22,0.22,0.22).*

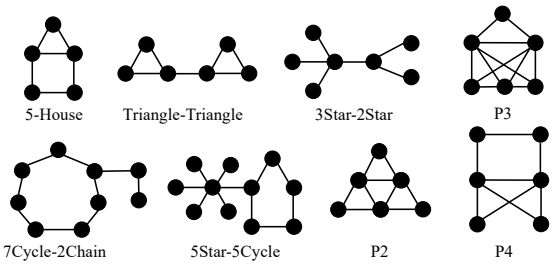


Figure 3: **Evaluated complex patterns.**

is partitioned based on node index hash into relatively similar sizes, and edges belonging to the same node are put into one partition. Since we do not have access to (single-machine) Automine, DwarvesGraph, and (distributed) Kudu, we refer to their performance numbers in a similar setup. Since ASAP is built on Spark, we reimplement its sampling approach using our API for a fair comparison.

Hardware testbed. Our experiments are carried out in three different hardware configurations: (1) Single-machine DRAM, which has 20 CPU cores and 188 GB of DRAM. (2) Single-machine PMEM (persistent memory) with additional 4 × 128GB Intel Optane DCPM. (3) Distributed settings with 4 to 32 machines in a cluster, each with the same configuration as the single-machine-DRAM. The testbed CPUs are Intel Xeon Silver 4114 clocked at 2.2Ghz per core.

Evaluated patterns. We evaluate both simple patterns (Triangle and 4-Motif) and complex patterns. Most prior systems did not evaluate patterns larger than 5-vertex while Arya can mine arbitrary patterns in large graphs. We describe the complex patterns we evaluate in Figure 3.

7.1 Single-Machine Performance

Overall Performance. We first compare Arya to Peregrine, Automine, Dwarvesgraph, and GraphPi. As shown in Table 2, we evaluate both simple patterns (triangle, 3-Motif) and complex patterns of up to 11 vertices on medium (Mico) and large graphs (Friendster). We also mine the extremely complex patterns such as 3Star-2Star. The results show that Arya significantly outperforms existing systems, particularly in complex patterns. On Mico, complex patterns (3Star-2Star, 7Cycle-2Star, 5Star-5Cycle) always take longer than 24 hours or crash. The long running time of Peregrine illustrates that

Pattern	Graph	System	Runtime	Error/Speedup
Triangle	Mico	**Arya**	**22ms**	0.74%
		Peregrine	46ms	2×
		GraphPi	3.5s	159×
	Friendster	**Arya**	**15ms**	1.24%
		Peregrine	11.3s	782×
		GraphPi	770.5s	51367×
3-Motif	Mico	**Arya**	**36ms**	0.09%
		Peregrine	67ms	1.8×
		DwarvesGraph	48ms	1.3×
		AutoMine	161ms	4.4×
		GraphPi	6.86s	190×
	Friendster	**Arya**	**59ms**	0.71%
		Peregrine	20.6s	349×
		GraphPi	804.4s	13634×
4-Motif	Mico	**Arya**	**1.0s**	0.42%
		Peregrine	5.2s	5.2×
		DwarvesGraph	1.3s	1.3×
		AutoMine	22s	22×
		GraphPi	21s	21×
	Friendster	**Arya**	**13248s**	0.76%
		Peregrine	2158s	**1/6**×
		DwarvesGraph	4369s	**1/3**×
		GraphPi	4399s	**1/3**×
3Star-2Star (7 vertices)	Mico	**Arya**	**0.8s**	N/A
		Peregrine	>24h	105365×
		GraphPi	2.33s	2.91×
	Friendster	**Arya**	**287s**	N/A
		Peregrine	Crashed	N/A
		GraphPi	924s	3.22×
7Cycle-2Chain (9 vertices)	Mico	**Arya**	**4s**	N/A
		Peregrine	Crashed	N/A
		GraphPi	Stuck	N/A
5Star-5Cycle (11 vertices)	Mico	**Arya**	**211s**	N/A
		Peregrine	>24h	409×
		GraphPi	Stuck	N/A
P3 [64]	Mico	**Arya**	**11s**	2.5%
		GraphPi	8.7s	**1/1.2**×
P4 [64]	Mico	**Arya**	**6.7s**	1.6%
		GraphPi	13.3s	2×

Table 2: **Single-machine DRAM: Arya vs. Peregrine, DwarvesGraph, Automine.** *This table summarizes runtime of Arya and other graph engines on various patterns (first column) and graphs (second column). Arya has a 5% error target.*

existing exact mining systems are fundamentally incapable of mining complex patterns. In contrast, Arya counts 3Star-2Star in Mico in 0.8s, outperforming Peregrine by 105,365×. We observe that while GraphPi completes mining star-related patterns, their results were incorrect, which prevents us to evaluate Arya's errors in some cases.

In this setting, we also explore an undesirable scenario for Arya (and any sampling-based approaches). In the Friendster graph, the occurrence of 4-Motif is relatively "sparse", making sampling-based approaches fundamentally more challenging to sample patterns. This is due to the "searching a needle in a haystack" effect and it is an ideal case for traversal-based solutions. Thus, in this scenario, Arya is running 3 to 6 times slower than exact mining solutions.

Table 3 shows results for Arya's **intermediate state caching** technique under the scenario when running three

Mico	Triangle-Triangle	5-House	Triangle
No Cache	13.3s	4.8s	0.079s
Cache	14.6s	3.0s	0.037s
Speed Up/Down	0.91×	1.6×	21.2×
Youtube	**Triangle-Triangle**	**5-House**	**Triangle**
No Cache	188.7s	297.9s	0.32s
Cache	198.7s	127.6s	0.011s
Speed Up/Down	0.95×	2.3×	27.9×

Table 3: **Arya's intermediate state caching technique.** *This table summarizes runtime and speedup of applying intermediate state caching technique when mining three patterns consecutively. Since these patterns share a common subpattern triangle, Arya caches the triangle samples in mining Triangle-Triangle and reuse them in the 5-House mining. Similarly, Arya also caches additional triangle samples when mining 5-House. These cached triangle samples accelerate the Triangle mining task significantly.*

Pattern	Graph	System	Runtime
Triangle	RMAT-5B	Arya (10%)	89s
	RMAT-5B	Arya (5%)	337s
	RMAT-5B	Peregrine	Crashed
3Star-2Star	RMAT-5B	Arya (10%)	395s
	RMAT-5B	Arya (5%)	1583s
	RMAT-5B	Peregrine	Crashed

Table 4: **Scaling single-machine Arya to giant graphs with PMEM.** *This table summarizes runtime of Arya (10% and 5% error rates) and Peregrine when mining on RMAT-5B.*

mining tasks (Triangle-Triangle, 5-House, and Triangle) on the same graph. Arya can mine multiple patterns one by one. Except for the last pattern Triangle, the sampled subpatterns and their actual sampling probabilities are cached; starting from the second pattern, we can reuse the cached subpatterns instead of sampling new ones and thus the running time is reduced. This experiment shows that when mining multiple patterns with shared subpatterns, Arya can achieve significant speedups (e.g., up to 27.9× for the last task) as the performance bottleneck is sampler computation and performing caching has negligible performance overheads.

We add **persistent memory** into the single machine to mimic large memory machines. On a giant 5-billion-edge graph (RMAT-5B), Arya counts triangles in 337 sec and mines a complex pattern of 7 vertices (3Star-2Star) in less than 30min while Peregrine fails to complete (Table 4).

Arya vs. ASAP. Figure 4 compares the running time of Arya and ASAP for different error rates. Both approximate systems, as expected, require more samplers to achieve lower error rates. However, the performance differences of the two approaches lie in two key factors: (1) the number of samplers needed and (2) the running time of each sampler. Compared to ASAP with the same error rate, Arya usually achieves better runtime because it requires fewer samplers (due to decomposition) and/or individual samplers run faster (due to Arya uses edge sampling while ASAP uses neighborhood sampling and Arya's system optimizations). For instance, when the graph is large (e.g., YouTube), the pattern is complex (e.g., 5-House, Triangle-Triangle), Arya requires fewer samplers, each of which is also faster than that of ASAP. Thus, for example, Youtube graph and 5-House pattern (Figure 4 (c)), Arya achieves less than a 5% error rate in 1.2s, whereas ASAP takes 3 min (145× slower). For small dense graphs (e.g., Mico, Figure 4 (b)), Arya and ASAP have comparable performances because they require comparable numbers of samplers, and their samplers have similar running times.

7.2 Scaling Arya on Distributed Settings

Arya can mine graphs that are (a) replicated across servers and (b) partitioned across servers (e.g., simulating geo-distributed graphs). We use a cluster with 4 to 32 servers.

7.2.1 Distributed Replicated Graphs

When graphs are replicated across nodes, Arya mines on each node in parallel and aggregates sampled results in a "reduce" phase. Many existing systems (e.g., Fractal and GraphPi) scale to multiple nodes using replicated graphs.

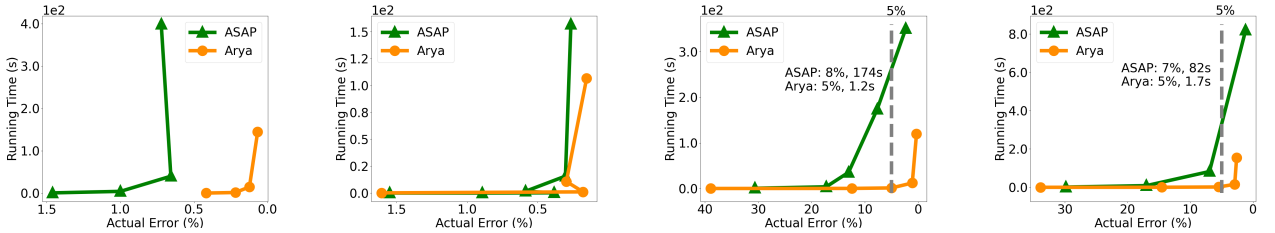
Overall performance. As depicted in Table 5, we compare Arya to Fractal and GraphPi on a 4-node cluster. In summary, Arya outperforms both Fractal and GraphPi, especially when graph is large and pattern is complex. For example, when mining triangles on the Twitter Graph, Arya achieves a 988× speedup over GraphPi. When the pattern is changed to Triangle-Triangle (a 6-vertex complex pattern), neither Fractal nor GraphPi can complete execution within a day, whereas Arya takes only 393 seconds. Overall, our results show that Arya has a significant advantage when mining complex patterns on large graphs.

Table 6 compares the performance of Arya, GraphPi, and ASAP (Spark version) on larger clusters. Arya outperforms both ASAP (by up to 55×) and GraphPi (by up to 1084×) on simple (e.g., 3-Motif) and complex patterns (e.g. 5-House). Referring to one of GraphPi's pattern (P2) mining results on a world-class supercomputer (up to 1024 nodes), we can see that approximate graph mining system Arya can achieve even better performance with only 16 nodes, demonstrating significantly better scalability.

Scalability in a Cluster. Figure 6 illustrates how Arya scales as more nodes (or cores) are added to the cluster. The runtime of Arya decreases as more machines are assigned to it. Arya can scale for both small and large graphs, whereas GraphPi cannot scale for larger Twitter graph with more than eight nodes. We find that the scaling of Arya is slightly worse than linear scaling. This is due to increased synchronization overheads of the final results when there are more nodes.

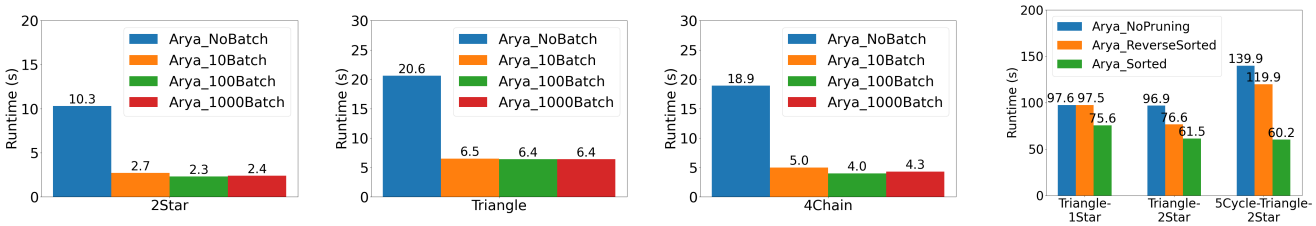
7.2.2 Distributed Partitioned Graphs

Unlike in a replicated graph setup, the graphs are partitioned across machines, and Arya samplers now require communi-



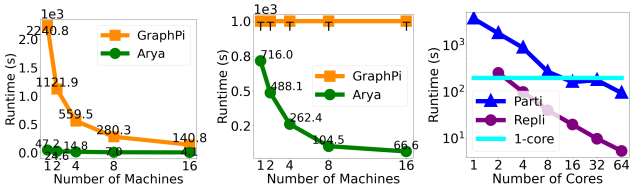
(a) Youtube, 4-Chain (b) Mico, 5-House (c) Youtube, 5-House (d) Youtube, Triangle-Triangle

Figure 4: **Comparing ASAP and Arya on running time vs. actual errors.** *This figure compares ASAP (our reimplementation for fairness) and Arya's running time (y axis) v.s. estimation errors (x axis, descending order). We report median absolute error rate \|%\| from 10 runs of each experiment. As shown, Arya requires fewer samplings and less runtime for the same error rate as ASAP, especially for large graphs and complex patterns.*



(a) Batching - Friendster, 8 Nodes (b) Probability-aware sampler scheduling

Figure 5: **Effectiveness of Arya batching and probability-aware sampler scheduling.** *This figure compares the performance of Arya with and without 1) batching and 2) probability-aware sampler scheduling techniques. In figure (a), Arya_NoBatch represents Arya without batching. Arya_KBatch represents Arya with K batched sampling and batched communication. We vary K between 10 and 1000. In figure (b), Arya_NoPruning represents the basic version of Arya, which samples all sampling blocks and then judges them all together. Arya_Sorted represents Arya when sampling according to the fail probability of each sampling block and terminating the estimator after any block fails. Arya_ReverseSorted is Arya that does sampling based on fail probability in a descending order.*



(a) Mico, Δ_Δ (b) Twitter, 5-House (c) Friendster, 4-Chain

Figure 6: **Scalability of Arya.** *This figure shows the performance of Arya when the number of nodes (cores) with replicated and partitioned graphs varies. We examine both small (Mico) and large (Twitter and Friendster) graphs, as well as various patterns. The letter 'T' indicates that the execution time exceeds 24 hours. Δ_Δ denotes Triangle-Triangle pattern. (a) and (b) shows the scalability of replicated-graph version Arya compared with GraphPi. (c) shows the replicated-graph and partitioned-graph versions of Arya compared with single-thread Arya showing the COST metric on Friendster graph and 4-Chain pattern, where "Parti" represents partitioned-graph version and "Repli" represents replicated-graph version.*

cations with remote nodes to obtain the necessary sampled edges or neighbors of vertices for testing. In this experiment, graphs are partitioned into machines based on their vertices and associated edges, and is stored in a Memcached instance on each machine. For simplicity, we evenly partition the graph to Memcached nodes.

Effectiveness of Arya Scaling Techniques. We begin by demonstrating the efficacy of batching and probability-aware scheduling techniques in improving Arya performance on partitioned graph setups. As shown in Figure 5 (a), batching can significantly improve Arya performance on partitioned graph setups. Batching improves Arya 2-Star, Triangle, and 4-Chain mining by 4.5×, 3.2×, and 4.7×, respectively, on eight nodes with Friendster graph. Because we've found that batching more than 100 samplers together yields minimal benefits, we set the default batching size to 100.

Figure 5 (b) shows how probability-aware sampler scheduling can help with Arya mining complex patterns. In this experiment, we use two nodes and the Mico graph. As an example, consider the following subpatterns: 2-Stars, triangle, and 5-Cycle. These subpatterns have very different sampling success probabilities: 2-Stars: 99.5%, Triangles: 8%, and 5-Cycles: 0.09%. When mining complex patterns containing these subpatterns, we can see that Arya samples with sorted likely-to-fail subpattern samplers and the early pruning achieves up to 2.3× (for 5Cycle-Triangle-2Star) better performance than no pruning. Arya's samplers with other orderings of subpatterns (e.g., *ReverseSorted*) cannot achieve comparable performance to fail probability sorted sampling.

McSherry's COST metric [55]. Figure 6 (c) shows the scalability of Arya's distributed replicated and partitioned versions compared with the runtime of a single thread Arya. The replicated version's COST is around 2.7 cores because the MPI implementation uses a master thread to poll results from worker threads, using at least 1 core. The partitioned version's

Pattern	Graph	System	Runtime	Error/Speedup
Triangle	Mico	**Arya**	**0.5s**	0.74%
		Fractal	145s	278×
		GraphPi	5.4s	10×
	Youtube	**Arya**	**0.55s**	0.78%
		Fractal	317s	576×
		GraphPi	38s	69×
	Twitter	**Arya**	**3.8s**	0.96%
		Fractal	>24h	22,736×
		GraphPi	3755s	988×
4-Motif	Mico	**Arya**	**3.3s**	0.42%
		Fractal	205s	62×
		GraphPi	33s	10×
	Youtube	**Arya**	**123s**	0.42%
		Fractal	29966s	243×
		GraphPi	219s	1.8×
	Twitter	**Arya**	**360s**	0.23%
		Fractal	failed	N/A
		GraphPi	>24h	240×
5-House	Mico	**Arya**	**0.8s**	0.63%
		Fractal	1822s	2366×
		GraphPi	6.3s	8×
	Youtube	**Arya**	**18s**	0.65%
		Fractal	2479s	142×
		GraphPi	36s	2×
	Twitter	**Arya**	**265s**	4.06%
		Fractal	failed	N/A
		GraphPi	>24h	326×
Δ_Δ	Mico	**Arya**	**1.5s**	0.71%
		Fractal	>24h	56,842×
		GraphPi	560s	368×
	Youtube	**Arya**	**15s**	1.13%
		Fractal	>24h	5760×
		GraphPi	11696s	779×
	Twitter	**Arya**	**393s**	N/A
		Fractal	failed	N/A
		GraphPi	>24h	220×

Table 5: **Distributed replicated graphs (4-nodes).**

COST is around 13 cores due to large communication costs in Memcached. In this version, scaling 1-core to 16-core experiments run on a single machine, and 32-core and 64-core experiments run on 2 and 4 machines of 16 cores.

Overall performance. Table 7 summarizes Arya's overall performance in comparison to G-thinker and Kudu. Kudu is a system that converts single-machine or distributed replicated graph mining systems (such as GraphPi and Automine) to partitioned graph setups. As shown in the table, Arya outperforms all existing exact graph mining systems on small (e.g., Mico) and large (Friendster) graphs, mining simple (e.g., 2-Star) and complex patterns (e.g., Triangle-2Star). The improvement is most noticeable on complex patterns. G-thinker, for example, fails to execute both Triangle-1Star and Triangle-2Star on a small Mico graph within a day; however, Arya can finish in seconds, yielding a speedup of more than 44000×.

Mining 10-billion edges graph on a large cluster. Table 8 shows the Arya runtime with 32 nodes mining patterns on a 10 billion edges graph (RMAT-10B). As shown in the table, Arya can mine huge graphs quickly. It completes triangle counting in 22 minutes for a 5% error rate and 358s for a 10% error rate. When mining 4-Chain, we see similar levels of speed. Arya can mine complex patterns even though it requires more time of sampling on a huge graph; for example, for a pattern with 7 vertices like 3Star-2Star, Arya finishes in 4.2h with a 10% error rate.

Pattern	Graph	System	Runtime	Error/Speedup
3-Motif	Twitter	**Arya, 16 × 8**	2.8s	0.34%
		ASAP, 16 × 8	150s	55×
		GraphPi, 16 × 8	2971s	1084×
5-House	Twitter	**Arya, 16 × 16**	60s	4.06%
		ASAP, 16 × 16	738s	12×
		GraphPi, 16 × 16	> 24h	1440×
Δ_Δ	Twitter	**Arya, 16 × 20**	100s	N/A
		GraphPi, 16 × 20	> 24h	864×
P2 [64]	Twitter	**Arya, 16 × 20**	856s	N/A
		GraphPi, 16 × 20	23.2h	98×
		GraphPi, 128 × 24	10000s	12×
		GraphPi, 1024 × 24	3000s	3.5×
P4 [64]	Twitter	**Arya, 16 × 20**	1600s	N/A
		GraphPi, 16 × 20	> 24h	54×

Table 6: **Comparing Arya, GraphPi, and ASAP on larger clusters.** *This table presents Arya/ASAP/GraphPi runtime on different clusters. The "system" column indicates system, the number of machines × the number of cores per machine. Arya is set to a 5% error target. GraphPi, 128 × 24 and 1024 × 24 results are picked from GraphPi paper [64].*

Pattern	Graph	System	Runtime	Error/Speedup
2-Star	Friendster	**Arya** 4 Nodes	0.58s	0.70%
		G-thinker 4 Nodes	52.4s	90×
		Arya 8 Nodes	0.64s	0.70%
		G-thinker 8 Nodes	30.8s	48×
Triangle	Friendster	**Arya** 4 Nodes	0.94s	1.24%
		G-thinker 4 Nodes	99s	105×
		Arya 8 Nodes	0.76s	1.24%
		G-thinker 8 Nodes	58s	76×
		Kudu-GraphPi 8 Nodes	79s	104×
		Kudu-Automine 8 Nodes	84s	110×
Triangle-1Star (5 vertices)	Mico	**Arya** 2 Nodes	1.93s	0.95%
		G-thinker 2 Nodes	>24h	44766×
Triangle-2Star (6 vertices)	Mico	**Arya** 2 Nodes	1.73s	0.40%
		G-thinker 2 Nodes	Crashed	N/A

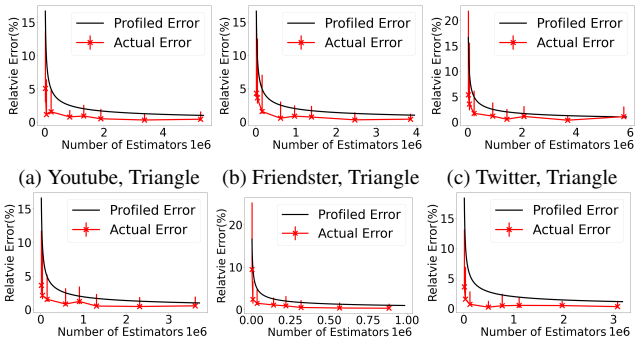
Table 7: **Distributed Partitioned Graphs: Arya vs. G-thinker vs. Kudu-GraphPi, Kudu-Automine.** *This table compares Arya and G-thinker and Kudu performance on partitioned graphs. The system column indicates both system name and how many nodes the graph is partitioned to.*

7.3 Effectiveness of Arya ELP

Finally, we evaluate the effectiveness of Arya Error-Latency Profiling. In this experiment, we compare the actual error vs. the predicted error from Arya ELP given a variety of amount of samplers. The Arya runtime is proportional to the number of samplers. As depicted in Figure 7, we investigate various patterns (Triangle and 3-Star) on various graphs (Youtube, Friendster, Twitter). We build the error profile by running

Pattern	Graph	System	Runtime
Triangle	RMAT-10B	Arya (10%)	358s
	RMAT-10B	Arya (5%)	1275s
4-Chain	RMAT-10B	Arya (10%)	171s
	RMAT-10B	Arya (5%)	688s
3Star-2Star	RMAT-10B	Arya (10%)	4.2h
	RMAT-10B	Arya (5%)	16.5h

Table 8: **Arya mining 10-billion edges huge graph.** *This table summarizes runtime of Arya (10% and 5% error rate) when mining on RMAT-10B on a 32-node cluster.*



(a) Youtube, Triangle (b) Friendster, Triangle (c) Twitter, Triangle

(d) Youtube, 3_Star (e) Friendster, 3_Star (f) Twitter, 3_Star

Figure 7: **Relative errors vs. number of estimators for YouTube, Friendster, and Twitter graphs.** *Actual error is obtained by compare Arya results with ground truth, and profiled error is the error estimated by ELP given a number of samplers.*

different numbers of samplers. We run ten trials for each x-axis value and report the *median* and *variance* error bars. As we can see, Arya ELP yields good upper bounds for the required error targets.

8 Related Work

Single-machine mining systems. A number of single machine exact mining systems have been proposed [26, 28, 43, 45, 54, 70]. These systems leverage a wide spectrum of system optimizations to prune the intermediate state and accelerate the subgraph exploration process. For instance, Peregrine [45] focuses on pattern-aware techniques to reduce the exploration of unnecessary subgraphs. Essentially, it prunes the incomplete subgraphs early if they cannot later form the pattern. Automine [54] and RStream [70] use guided exploration strategies but reduce memory usage. The fundamental performance bottleneck of these exact systems is that *regardless of how optimized the exploration techniques are, one must still explore all the patterns in the graph*. When the pattern occurrences are dense in the graph, this bottleneck will be significant. Arya leverages decomposition-based sampling, which significantly reduces exploration complexity. Another related work is DwarvesGraph [26], which also uses a type of pattern composition to count the decomposed subpatterns separately and thus reduces the overall computation. However, DwarvesGraph's decomposition technique cannot be applied with sampling techniques to further reduce search complexity. Some related architecture works leverage different architectures and hardware to accelerate enumerating graph patterns [27, 62, 65–67], while Arya and compared baselines run on general-purpose CPUs.

Distributed mining systems. To scale graph mining tasks on larger graphs, a wide range of distributed mining systems are proposed [10, 17, 25, 33, 53, 64, 69, 72]. Recent systems such as Fractal [33] and GraphPi [64] focused on supporting general-purpose mining tasks. Fractal extends the "embedding-aware" processing model by introducing the concept of *fractoids* and reduces the complexity of its depth-first search exploration. GraphPi is a high-performance graph miner that optimizes computation and communication overheads by introducing a 2-cycle-based automorphism elimination algorithm. GraphPi scales to supercomputers (up to 1024 compute nodes) to support complex pattern mining in large graphs. Arya's decomposition-based sampling technique further improves the scalability by several orders of magnitude.

Graph approximation theory. There have been rich efforts from theory community to analyze and propose approximate graph algorithms for various graph analytical tasks [12, 13, 18–20, 32, 42, 60, 71]. Among these efforts, only a small subset of them are used in graph systems. None of them are aimed at distributed scenarios, nor do they introduce methods to understand the real-world performance of the algorithms. To bring theory into practice, we entail careful understanding of the algorithmic tradeoffs and the actual computation scenarios. We leverage this rich theoretical foundation to further improve the sampling-based approximate systems and propose a series of sampling-friendly optimizations.

9 Conclusions

We observe that existing graph pattern mining systems cannot scale to complex pattern mining over large graphs as they fail to cope with the explosively growing mining complexity. We propose Arya as an approximate graph miner that combines graph decomposition theory with sampling techniques to achieve optimized mining complexity over arbitrary patterns. Arya can deal with large billion-level graphs even in a single machine and can scale to larger graphs in distributed settings. Our evaluation demonstrates that Arya outperforms state-of-the-art mining systems by up to five orders of magnitude. We posit that Arya can potentially be applied to extreme mining scenarios (e.g., trillion edges) on a small computing base, and we plan to explore this for future work.

Acknowledgments. We would like to thank the anonymous reviewers and our shepherd Peter Pietzuch for their helpful comments. This work was supported in part by NSF grants CNS-2107086, CNS-2106946, SaTC-2132643, and Red Hat Collaboratory at Boston University.

References

[1] Ant financial's innovations and practices in online graph computing. https://www.alibabacloud.com/blog/ant-financials-innovations-and-practices-in-online-graph-computing_595846.

[2] Arya graph pattern mining codebase. https://github.com/Froot-NetSys/Arya.

[3] General Data Protection Regulation. https://gdpr-info.eu/.

[4] Graph databases: Updates on their growing popularity. https://www.dataversity.net/graph-databases-updates-on-their-growing-popularity/.

[5] Neo4j Privacy Shield: The Graph Solution for GDPR. https://neo4j.com/use-cases/gdpr-compliance/.

[6] Stanford Large Network Dataset Collection. https://snap.stanford.edu/.

[7] Weg graph. http://webdatacommons.org/, 2014.

[8] A comparison of state-of-the-art graph processing systems. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/, 2016.

[9] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727. IEEE, 2016.

[10] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727. IEEE, 2016.

[11] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[12] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 459–467, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[13] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, pages 5–14, New York, NY, USA, 2012. ACM.

[14] M. Aliakbarpour, A. S. Biswas, T. Gouleakis, J. Peebles, R. Rubinfeld, and A. Yodpinyanee. Sublinear-time algorithms for counting star subgraphs via edge sampling. *Algorithmica*, 80(2):668–697, 2018.

[15] M. Aliakbarpour, A. S. Biswas, T. Gouleakis, J. Peebles, R. Rubinfeld, and A. Yodpinyanee. Sublinear-time algorithms for counting star subgraphs via edge sampling. *Algorithmica*, 80(2):668–697, 2018.

[16] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241–i249, 2008.

[17] Apache Giraph. http://giraph.apache.org.

[18] S. Assadi, M. Kapralov, and S. Khanna. A simple sublinear-time algorithm for counting arbitrary subgraphs via edge sampling. *Innovations in Theoretical Computer Science (ITCS)*, 2018.

[19] S. Assadi, S. Khanna, and Y. Li. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 1723–1742, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

[20] V. Braverman, R. Ostrovsky, and D. Vilenchik. *How Hard Is Counting Triangles in the Streaming Model?*, pages 244–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[21] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[22] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA. USENIX.

[23] J. Brynielsson, J. Högberg, L. Kaati, C. Mårtenson, and P. Svenson. Detecting social positions using simulation. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 48–55. IEEE, 2010.

[24] C. Chen, C. Liang, J. Lin, L. Wang, Z. Liu, X. Yang, J. Zhou, Y. Shuang, and Y. Qi. Infdetect: a large scale graph-based fraud detection system for e-commerce insurance. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1765–1773. IEEE, 2019.

[25] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[26] J. Chen and X. Qian. Dwarvesgraph: A high-performance graph mining system with pattern decomposition. *arXiv preprint arXiv:2008.09682*, 2020.

[27] Q. Chen, B. Tian, and M. Gao. Fingers: exploiting fine-grained parallelism in graph mining accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 43–55, 2022.

[28] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*, pages 378–391, 2021.

[29] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, Aug. 2015.

[30] W. H. Cunningham and J. Edmonds. A combinatorial decom-

position theory. *Canadian Journal of Mathematics*, 32(3):734–765, 1980.

[31] A. Czumaj and C. Sohler. Estimating the weight of metric minimum spanning trees in sublinear time. *SIAM Journal on Computing*, 2009.

[32] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 69–78, New York, NY, USA, 2008. ACM.

[33] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374, 2019.

[34] T. Eden, D. Ron, and C. Seshadhri. On approximating the number of k-cliques in sublinear time. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 722–734, 2018.

[35] T. Eden and W. Rosenbaum. Lower bounds for approximating graph parameters via communication complexity. *arXiv preprint arXiv:1709.04262*, 2017.

[36] T. Eden and W. Rosenbaum. On sampling edges almost uniformly. *arXiv preprint arXiv:1706.09748*, 2017.

[37] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, 2014.

[38] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.

[39] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *Proceedings of the VLDB Endowment*, 6(13):1510–1521, 2013.

[40] I. Finocchi, M. Finocchi, and E. G. Fusco. Clique counting in mapreduce: Algorithms and experiments. *Journal of Experimental Algorithmics (JEA)*, 20:1–20, 2015.

[41] S. Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

[42] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. In *Proceedings of the 2012 Internet Measurement Conference*, IMC '12, pages 131–144, New York, NY, USA, 2012. ACM.

[43] C. Gui, X. Liao, L. Zheng, P. Yao, Q. Wang, and H. Jin. Sumpa: Efficient pattern-centric graph mining with pattern abstraction. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–330. IEEE, 2021.

[44] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. {ASAP}: Fast, approximate graph pattern mining at scale. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 745–761, 2018.

[45] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[46] C. Jedrzejek, J. Bak, and M. Falkowski. Graph mining for detection of a large class of financial crimes. In *17th International Conference on Conceptual Structures, Moscow, Russia*, volume 46, 2009.

[47] M. Jha, C. Seshadhri, and A. Pinar. A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox. *ACM Trans. Knowl. Discov. Data*, 9(3):15:1–15:21, Feb. 2015.

[48] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[49] M. Kurant, M. Gjoka, Y. Wang, Z. W. Almquist, C. T. Butts, and A. Markopoulou. Coarse-grained topology estimation via graph sampling. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 25–30, 2012.

[50] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[51] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[52] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, et al. Kudu: Storage for fast analytics on fast data. *Cloudera, inc*, 28, 2015.

[53] D. Mawhirter, S. Reinehr, W. Han, N. Fields, M. Claver, C. Holmes, J. McClurg, T. Liu, and B. Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 289–303. IEEE, 2021.

[54] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523, 2019.

[55] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[56] K. Michalak and J. Korczak. Graph mining approach to suspicious transaction detection. In *2011 Federated conference on computer science and information systems (FedCSIS)*, pages 69–75. IEEE, 2011.

[57] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[58] J. Mondal and A. Deshpande. Stream querying and reasoning on social data. In *Encyclopedia of Social Network Analysis and Mining*, pages 2063–2075, 2014.

[59] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[60] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. *Proc.*

VLDB Endow., 6(14):1870–1881, Sept. 2013.

[61] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment*, 6(14):1870–1881, 2013.

[62] G. Rao, J. Chen, J. Yik, and X. Qian. Sparsecore: stream isa and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–199, 2022.

[63] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 328–341, 2010.

[64] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: high performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[65] J. Su, L. He, P. Jiang, and R. Wang. Exploring pim architecture for high-performance graph pattern mining. *IEEE Computer Architecture Letters*, 20(2):114–117, 2021.

[66] N. Talati, H. Ye, S. Vedula, K.-Y. Chen, Y. Chen, D. Liu, Y. Yuan, D. Blaauw, A. Bronstein, T. Mudge, et al. Mint: An accelerator for mining temporal motifs.

[67] N. Talati, H. Ye, Y. Yang, L. Belayneh, K.-Y. Chen, D. T. Blaauw, T. N. Mudge, and R. G. Dreslinski. Ndminer: accelerating graph pattern mining using near data processing. In *ISCA*, pages 146–159, 2022.

[68] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. {RIPQ}: Advanced photo caching on flash for facebook. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 373–386, 2015.

[69] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, New York, NY, USA, 2015. ACM.

[70] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. {RStream}: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 763–782, 2018.

[71] T. Wang, Y. Chen, Z. Zhang, T. Xu, L. Jin, P. Hui, B. Deng, and X. Li. Understanding graph sampling algorithms for social network analysis. In *2011 31st international conference on distributed computing systems workshops*, pages 123–128. IEEE, 2011.

[72] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W.-S. Ku, and J. C. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1369–1380, 2020.

[73] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[74] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[75] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[76] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. {FlashGraph}: Processing {Billion-Node} graphs on an array of commodity {SSDs}. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

A More Details on Predicate Matching

In predicate "all", queries specify a predicate that is applied to every edge or vertex. For instance, one can query "5-House patterns where all edges have the property NSDI". To support such a query, Arya introduces *a conservative sampling* stage to generate a new graph (of edges and their vertices) from the original graph, where the predicate condition is applied to every edge; and in the sampling phase, Arya runs only on this new property graph. This step ensures that "all" edges match the predicate.

In predicate "at-least-one", queries specify a predicate that is applied to at-least one edge or vertex. For example, one such query is "5-House patterns where at-least-one edge has a weight > 66". To support such predicate queries, we change the runtime workflow to take two passes on the graph. In the first pass, edges matching the predicate of weight > 66 are generated as a new graph. In the second pass, every sampler picks the first edge randomly from the new graph. This ensures that the pattern found by the sampler (if it does find one) meets the predicate. For the rest of the edges, the sampler continues sampling on the original graph, which can add zero or more edges that satisfy the predicate. If a duplicated edge is found, we disregard this sampler. This ensures that the probability analysis of Arya to estimate the error still holds.

Similarly, in the predicate "at-least-percentage", we use the two-pass approach as in the predicate "at-least-one". The only difference is we need to sample a percentage of edges from the newly generated graph.

B Detailed Explanation of Sampling Algorithms

In this section, we introduce the data structures and probability calculation used in the decomposition sampling. Algorithm 4 and Algorithm 2 describe the building blocks of decomposition sampling in our implementation as [18]. We use sampler trees as sampling data structures for maintaining the (inverse) probability. Figure 8 shows a $(2k+1)$-odd cycle sampler tree and an l-star sampler tree representation. Each subpattern (an odd cycle or a star) always has a two-layer sampler tree structure. The first layer is a root node, the second layer contains one or more nodes as leaves and we assign the inverse probability to each leaf.

Odd Cycle Sampler Tree. In Figure 8 (a), $e_1, ..., e_k$ in the cycle sampler tree root denotes first k edges sampled in Algorithm 4 (line 5 and 6) and $n_1, ..., n_b$ in b leaves denotes vertices sampled in line 7 and 8 where $b = \lceil d(u_1)/\sqrt{m} \rceil$. This set of nodes and edges can potentially form at most b $(2k+1)$-odd cycles in the original graph which are represented in b branches in the cycle sampler tree. The inverse probability of one leaf i is $Pr_i[C_{2k+1}] = Pr[e_1] \cdot Pr[e_2] \dots Pr[e_k] \cdot Pr[n_i] = \frac{1}{m}(\frac{1}{2m})^{k-1}\frac{1}{d(u_1)}$ if the root edges and the node in the leaf can form an odd cycle; otherwise, the inverse probability is defined as $Pr_i[C_{2k+1}] = 0$.

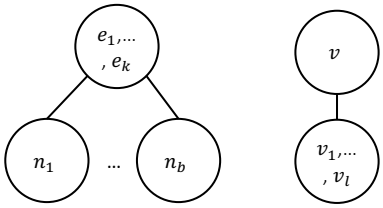
Algorithm 4 Odd Cycle Sampler Tree

1: **Input**:
2: Graph $G = (V, E)$ where $|E| = m$, an odd cycle C_{2k+1}
3: **Output**:
4: A set consisting of C_{2k+1} or 0
5: Sample an edge $e_1 = (u_1, v_1)$ from graph such that $d(u_1) \leq d(v_1)$. ▷ *sample first edge with an order*
6: Sample $k-1$ edges $e_2 \dots e_k$ with replacement from G. ▷ *sample rest edges as the cycle skeleton*
7: **for** $i = 1$ to $\lceil d(u_1)/\sqrt{m} \rceil$ **do** ▷ *Sample last hoop edge*
8: Sample a vertex n_i from the neighbors of u_1.
9: Test if there are edges in G to complete an odd cycle.

Star Sampler Tree. In Figure 8 (b), v in the root node is the central vertex of the star and $v_1, ..., v_l$ in the leaf node are l petals. The inverse probability of the leaf is $Pr[S_l] = Pr[v] \cdot Pr[petal_vertices] = \frac{d_v}{2m}\binom{d_v}{l}$, where m is the total number of edges in graph G.

Approximation for the Original Pattern. Supposing a pattern P consists of o odd cycles $C_{2k_1+1}, ..., C_{2k_o+1}$ and s stars $S_1, ..., S_s$, and $z = 2o + 2s$. A pattern-sampler tree will be a z-level tree which consists of odd cycle sampler subtrees and star sampler subtrees. To obtain the final pattern-sampler tree, we run subpattern samplers in some order. The pattern-sampler tree keeps extending two new layers by connecting each last-layer leaf-node to a new subpattern subtree root. A final sampler tree is shown in Figure 9 (a). We also show a 5-House sampler tree example in Figure 9 (b). As 5-House is decomposed into a triangle and an 1-star (see Figure 1), the first two layers of 5-House sampler tree represent a triangle sampler tree, and for the last two layers each branch represents a 1-star sampler tree.

A path from the root to a leaf in the pattern sampler tree forms a potential pattern. If path j passes connectivity test with remaining edges, the probability of the path is $Pr[P_j] = Pr[C_j^1] \dots Pr[C_j^o] Pr[S_j^1] \dots Pr[S_j^s]$ because subpatterns are sampled independently. The output of a sampler path is $R[P_j] = \frac{1}{Pr[C_j^1] \dots Pr[C_j^o] Pr[S_j^1] \dots Pr[S_j^s]}$ if it forms a pattern after testing; or $R[P_j] = 0$ if it's not. The estimated pattern number outputs by a pattern sampler tree is the average of each path's estimation output, which is $R[P] = \frac{\sum_{j=1}^{w} R[P_j]}{w}$ supposing we have w final-layer leaves. We aggregate results from all pattern sampler trees as their average number $\frac{\sum_{i=1}^{n} R_i[P]}{n}$, supposing there are n sampler trees and tree i outputs $R_i[P]$. [18] proves the expectation estimated by $R_i[P]$ is the number of pattern P in graph G (denoted as $\#P$) and variance is bounded.



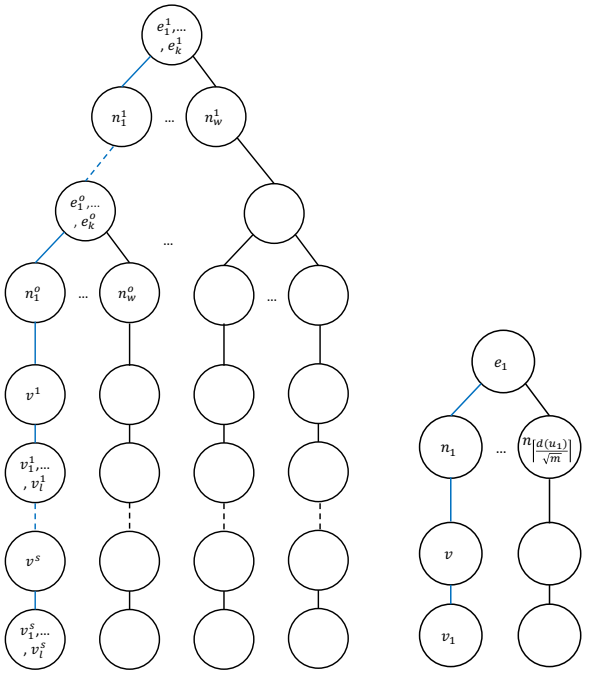
(a) $(2k+1)$-odd-cycle sampler tree (b) l-star sampler tree

Figure 8: **Subpattern sampler trees used in decomposition.** *This figure shows $(2k+1)$-odd-cycle sampler tree and l-star sampler tree each corresponding to Algorithm 4 and 2.*

C Computation-Communication Ratio Analysis in Partitioned-graph Setting

In this section, we calculate computation-communication ratio based on sampler tree implementation. In terms of computation cost, in an l-star sampler, we sample l neighbors randomly and thus the contribution to computation cost is $\Theta(l)$. In a $(2k+1)$-odd cycle sampler, we sample an edge from the graph first, which is $\Theta(1)$, and then sample $k-1$ edges with cost $\Theta(k-1)$. And then we sample $\lceil d(u_1)/\sqrt{m} \rceil$ vertices from the first node u_1's neighbors, which costs $\Theta(d(u_1)/\sqrt{m})$, where m is the number of edges in the large graph and $d(u_1)$ is the degree of the start vertex of the first sampled edge. Let $\overline{\Delta}$ denote the average degree of the large graph. Testing completeness of these $\lceil d(u_1)/\sqrt{m} \rceil$ cycles needs to test k edges, whose cost is $\Theta(k \cdot \overline{\Delta})$ after amortizing since our neighbor checking goes through all the neighbors of the start vertex. We also test the remaining edges of the entire pattern connecting each subpattern, and the computation cost is $\Theta(x \cdot \overline{\Delta})$ supposing there are x remaining edges. Supposing the pattern contains s stars and o odd cycles, the total computation cost for one sampler is $\sum_j^s l_j + \sum_i^o (k_i + \frac{\overline{\Delta}}{\sqrt{m}} + k_i \cdot \overline{\Delta}) + x \cdot \overline{\Delta}$ by amortizing the degrees among multiple sampling trials.

Supposing we have p partitions ($p \geq 2$). In our evaluation, we partition the vertices nearly uniformly by hashing the vertices to a machine, the probability of a sampler may not have a vertex's neighbors locally is $\frac{p-1}{p}$. For an l-star sampler, if the central vertex of this star is local to the machine running the sampler algorithm, the communication cost is 0 because we have all the neighbors of a vertex belonging to the partition stay in the same machine; Otherwise, the communication cost is $\Theta(d(u_{central}))$. Therefore, the l-star sampler communication cost is $\Theta(\frac{p-1}{p} d(u_{central}))$. For a $(2k+1)$-cycle sampler, we first sample an edge, if this edge is not local, we will retrieve all the neighbors for the start vertex. When sampling the next k edges, we may retrieve the neighbor list of a vertex for each edge. Thus the communication cost is $\Theta(\frac{p-1}{p} k \cdot \overline{\Delta}))$. To test these k edges and a neighbor of the first vertex form a cycle, we need to test the connectivity of k remaining edges in the $(2k+1)$-cycle, which cost is $\frac{p-1}{p} k \cdot \overline{\Delta}$.



(a) A pattern sampler tree (b) 5-House sampler tree

Figure 9: **Pattern sampler trees.** *Dotted lines represent there can be multiple other odd cycles or stars in the middle of the layers. Solid lines are connecting root and leaves in a subpattern sampler subtree or from a last cycle leaf node to a first star root node. The blue lines form a path from the root to a last-layer leaf. The labels of some nodes are omitted in the figure.*

Plus the x remaining-edge test of the entire pattern, the total communication cost is $\frac{p-1}{p}(\sum_i^o 2k_i \cdot \overline{\Delta} + \sum_j^s \overline{\Delta} + x \cdot \overline{\Delta})$.

Therefore, the computation-communication cost of partitioned Arya is $\frac{p}{p-1} \cdot \frac{\sum_j^s l_j + \sum_i^o k_i + \sum_i^o (\frac{1}{\sqrt{M}} + k_i) \cdot \overline{\Delta} + x \cdot \overline{\Delta}}{(\sum_i^o 2k_i + s + x) \cdot \overline{\Delta}}$, which is approximately $c\frac{p}{p-1}$ where c is a constant related to the pattern and the graph. This communication cost can be reduced by using batching technique mentioned in Section 5.2.

SECRECY: Secure collaborative analytics in untrusted clouds

John Liagouris *Vasiliki Kalavri* *Muhammad Faisal* *Mayank Varia*
Boston University
{liagos, vkalavri, mfaisal, varia}@bu.edu

Abstract

We present SECRECY, a system for privacy-preserving collaborative analytics as a service. SECRECY allows multiple data holders to contribute their data towards a joint analysis in the cloud, while keeping the data siloed even from the cloud providers. At the same time, it enables cloud providers to offer their services to clients who would have otherwise refused to perform a computation altogether or insisted that it be done on private infrastructure. SECRECY ensures no information leakage and provides provable security guarantees by employing cryptographically secure Multi-Party Computation (MPC).

In SECRECY we take a novel approach to optimizing MPC execution by co-designing multiple layers of the system stack and exposing the MPC costs to the query engine. To achieve practical performance, SECRECY applies physical optimizations that amortize the inherent MPC overheads along with logical optimizations that dramatically reduce the computation, communication, and space requirements during query execution. Our multi-cloud experiments demonstrate that SECRECY improves query performance by over 1000× compared to existing approaches and computes complex analytics on millions of data records with modest use of resources.

1 Introduction

Secure collaborative analytics [20,26,30,47,97] is a family of emerging applications, where multiple data holders are willing to allow certain computations on their collective private data (e.g., for profit, social good, improved services, etc.), provided that the data remain siloed from untrusted entities. For instance, some companies would agree to participate in a gender wage gap study [32] but only if no employee wages are revealed to other companies, as they may lose their competitive advantage. Similarly, researchers from different medical institutions may conduct a large-scale study on the union of their patient records, provided that the data analysis is end-to-end compliant with privacy regulations [2, 3]. Another example is private advertising: web users may subscribe to recommendations based on collaborative filtering as long as their online activity remains hidden from the service provider [85].

To realize the above use cases, we need systems capable of extracting value from sensitive or proprietary data, while protecting the data from untrusted or unauthorized entities. We identify four major requirements for such systems. First, they must ensure no information leakage so that they reveal nothing except the output of the computation the data holders have agreed on. At the same time, they must guarantee security in the absence of trusted resources, as the data holders may lack the expertise or infrastructure needed for secure computation and may need to outsource the analysis to untrusted third parties [45]. Another requirement is to support complex analytics beyond simple statistics, such as relational queries on multiple tables [95]. Lastly, while queries in these use cases are non-interactive, they must complete in reasonable time, e.g., within a few hours.

Enabling secure outsourced analytics with practical performance has been a long-standing research challenge [13]. So far, there exist three general approaches to secure computation with no leakage. The first one is Fully Homomorphic Encryption (FHE) [57] that provides "ideal" security by enabling computation directly on encrypted data. Although there are many implementations that support simple functions [4, 6, 8, 21], FHE is still prohibitively slow for the analytics we consider in this work. A more practical approach is to use secure hardware solutions, like Intel's SGX, which have been proposed as a faster alternative to cryptography but do not provide provable security [33, 74]. A third promising approach is cryptographically secure Multi-Party Computation (MPC) [77]. MPC refers to a family of cryptographic protocols that enable mutually distrusting parties to jointly compute functions on secret (encoded) data without relying on any single trusted entity. MPC is generally faster than FHE-based approaches but still challenging to scale to inputs with more than a few thousand records [22, 45, 73, 95, 105].

Recently, systems like Conclave [105], SMCQL [22], Senate [95], and others [23, 24, 111] have made MPC more accessible to data analysts by providing relational interfaces

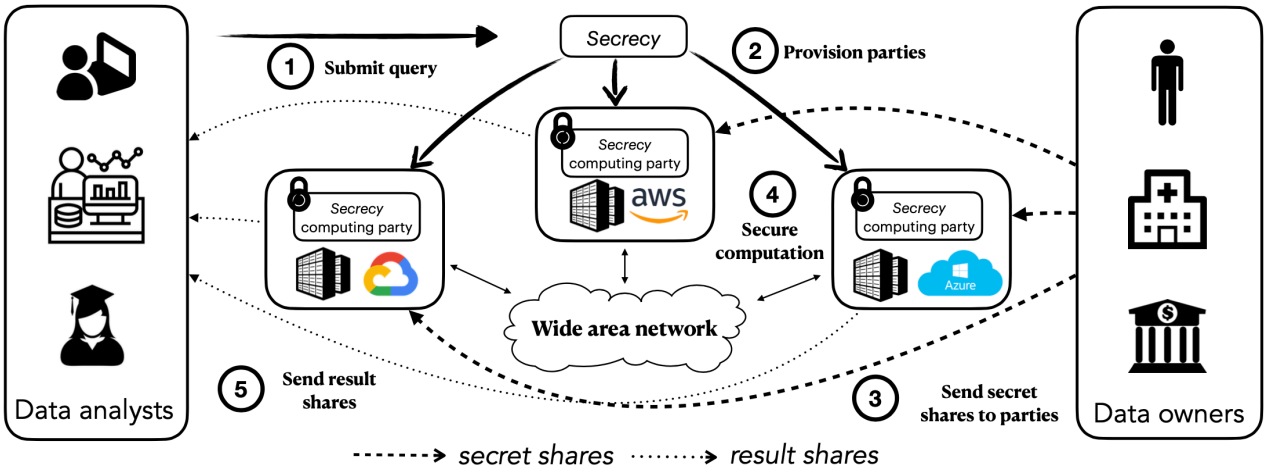


Figure 1: Overview of secure collaborative analytics with SECRECY. Clients (e.g., data analysts) access a catalog with metadata about available private datasets and submit public queries to the SECRECY service. SECRECY performs query planning and provisions computing parties in available non-colluding cloud providers, e.g., GCP, AWS, and Azure. Next, it instructs data owners to distribute *secret shares* (cf. §3) of their data to computing parties. Parties execute the query under MPC and send the results to the analysts. SECRECY considers adversaries who may have complete control over the network. All data remain private as long as an adversary does not compromise the majority of cloud providers.

and automated query planning. However, to achieve practical performance, these works employ optimizations that either leak information to untrusted parties or apply to peer-to-peer settings where data holders also serve as computing parties using trusted resources (we provide more details in §8). Outsourced MPC, on the other hand, removes the computation burden from data holders and has recently gained attention, especially in industry, with systems like Prio [45] Carbyne [11], CrypTen [70] and Cape Privacy's TF Encrypted [1]. Yet, these frameworks focus on certain statistics or ML workloads and do not support general-purpose analytics.

To fill this gap, we present SECRECY, a new relational MPC system for efficient collaborative analytics in the cloud with no information leakage. In SECRECY we take a fundamentally different approach over prior work and we carefully co-design the MPC protocol, query engine, and distributed runtime into a single platform. SECRECY's core novelty is a generic cost-based optimization framework for relational MPC that does not rely on trusted infrastructure. As such, it enables data holders and analysts to use untrusted cloud resources on demand and benefit from the "pay-as-you-go" model while retaining the *full* security guarantees of the cryptographic protocols.

Contributions. We make the following contributions:

- We present a relational MPC system, based on secret sharing, that enables efficient collaborative analytics with no information leakage.
- We design vectorized MPC primitives and relational operators that amortize the network I/O of secret sharing. Contrary to prevailing wisdom, we show that this approach can be competitive with widely adopted MPC techniques for relational analytics in both LAN and WAN environments.
- We define an analytical cost model that formulates MPC query costs in terms of secure computation and communication operations. We use this cost model to develop a novel query optimization framework for relational MPC.
- We implement a Volcano-style query processor that leverages the cost model to automatically apply a rich set of logical, physical, and protocol-aware optimizations which can improve performance by orders of magnitude.
- We evaluate SECRECY's performance and the effectiveness of its optimizations using real and synthetic queries. Our experiments show that SECRECY outperforms state-of-the-art MPC frameworks and scales to much larger datasets.

We believe SECRECY will become a valuable tool to cloud providers, data holders, and analysts by enabling new privacy-preserving applications and marketplaces on existing cloud infrastructure. We will release SECRECY as open-source [9].

2 SECRECY system overview

Figure 1 presents an overview of the SECRECY cloud service. Each party in SECRECY has one or more of the following roles: (i) *data holder or data owner* who provides some input data, (ii) *computing party*, e.g., a cloud provider that provides resources to perform the secure computation, and (iii) *analyst* who issues a query to learn the result. SECRECY supports any number of data owners and uses three computing parties. A "party" is a logical entity and does not necessarily correspond to a single compute node. SECRECY does not make any assumption about the physical deployment: parties can be deployed in private clusters, in a multi-cloud, or across multiple providers in a hybrid or federated cloud.

2.1 Design principles

We have designed SECRECY on the following principles:

1. No information leakage. SECRECY reveals nothing about the input, output, or intermediate data and the execution metadata to untrusted parties, including the cloud providers. It completely hides access patterns, intermediate, and output result sizes. SECRECY does not require data owners to annotate attributes as sensitive or non-sensitive and does not try to sidestep the secure computation. It executes all operations under MPC and protects all attributes to prevent inference attacks that exploit correlations or functional dependencies in the data which may be unknown to data owners.

2. No reliance on trusted execution environments. SECRECY does not rely on any (semi-)trusted party, honest broker or specialized secure hardware. To remove barriers for adoption, we target general-purpose compute and cloud.

3. Decoupling of roles. In SECRECY, a party may have any combination of roles, that is, data owners can (but do not have to) also act as computing parties and/or analysts without affecting the security guarantees. Query optimization in SECRECY does not rely on data ownership and does not require data owners to participate in MPC using trusted resources. Due to decoupling, SECRECY can effectively use a small number of computing parties to support any number of data owners without affecting the scalability of secure computation.

4. High expressivity. SECRECY's protocol does not pose any restriction on the types of queries that can be supported. While there exist many efficient protocols for specific instances of MPC operators, these are often not composable. In SECRECY, we have decided to provide general operator implementations that are independent of the data characteristics and can be composed with each other to create arbitrary query plans.

2.2 Threat model and security guarantees

SECRECY protects data throughout the entire lifecycle and treats the query itself as *public*, i.e., it assumes that data owners and analysts have previously agreed on a relational query to compute and this query is known to the computing parties, as in prior works [22–24, 95, 105]. To evaluate the query, SECRECY servers execute an identical computation and exchange messages with each other according to a protocol. All communication in a SECRECY deployment must be done via authenticated and encrypted channels (e.g., using TLS).

Threat model. SECRECY assumes "honest-but-curious" parties and can withstand adversaries who have two types of capabilities. First, adversaries have complete control over the network and can monitor all network links. Second, adversaries may compromise one computing party and can see all of its internal state (e.g., memory contents, access patterns, and data sent/received) but without altering its execution. That said, most of the techniques we present in this work are also compatible with malicious-secure MPC protocols where parties can deviate from the protocol arbitrarily (cf. §5.4).

Security guarantees. We have purposely designed SECRECY in a modular fashion to ensure it can directly inherit all security guarantees of the underlying MPC protocol. SECRECY relies on the semi-honest 3-party replicated secret sharing protocol by Araki et al. [17, 81]. The protocol provides two types of guarantees: (i) *privacy*, meaning that computing parties do not learn anything about the data, and (ii) *correctness*, meaning that all participants are convinced that the computation output is accurate. As long as the computing parties do not collude, the SECRECY *servers cannot learn anything beyond the size of the input data* (which can also be padded by the data owners). Only the designated analysts learn the result of the query. SECRECY does not use differential privacy to protect the result from possible inference attacks by the analysts but it could be easily augmented to do so (cf. §8). We also stress that formal verification of the SECRECY software is out of scope for this work (but an exciting future direction). For a detailed security analysis please refer to Appendix B.

2.3 Cost-based secure query optimization

Cost-based query optimization on plaintext data relies on selectivity estimations to reduce the size of intermediate results. MPC operators, however, are *oblivious*, i.e., their control flow is independent of the input data and incurs exactly the same accesses for all inputs of the same size. Oblivious operators do not reveal the size of intermediate data to prevent reconstruction attacks based on selectivity statistics [60, 69, 84]. As a consequence, traditional selectivity-based techniques for plaintext queries [41], such as join reordering or filter pushdown, are not effective when optimizing plans under MPC. For instance, given that oblivious selections do not reduce the size of intermediate data, pushing a filter down does not improve the cost of subsequent operators in the plan.

To devise effective optimizations under MPC, we express the plan costs in terms of secure computation and communication operations. In SECRECY, we define three types of costs:

- The **operation cost**, C_o, which is determined by the number of primitive MPC operations per party. Primitive operations can be *local* ($+, \oplus$), which do not require communication, or *remote* ($\times$, $\wedge$), which require some message exchange between parties (cf. §3).
- The **synchronization cost**, C_s, given by the number of communication rounds across parties that are inherent in MPC. Each round corresponds to a barrier, i.e. a synchronization point in the distributed execution, where parties must exchange data in order to proceed.
- The **cost of composition**, C_c, which is also measured in operations and communication rounds required to compose oblivious relational operators under MPC.

SECRECY applies automatic optimizations that aim to mini-

mize at least one of these three costs. We present a comprehensive cost analysis of oblivious operators and their composition in §4. Contrary to plaintext query optimization where estimations are often erroneous [76], in MPC we can use the typical dynamic programming approach from database optimizers [98] to compute *exact* plan costs at compile time, since C_o, C_s, and C_c do not depend on the data distribution.

3 Background on MPC

MPC protocols follow one of two general techniques: obscuring the truth table of each operation using Yao's *garbled circuits* [114], or performing operations over encoded data using *secret sharing* [100]. So far, garbled circuits has been the preferred method to securely compute Boolean circuits in high-latency environments, as they only need a small (constant) number of rounds between computing parties at the cost of incurring a large memory overhead [73]. On the other hand, secret sharing-based approaches require more rounds (that depend on the input size) but have a small memory footprint and consume less overall bandwidth. In this work we employ secret sharing in the honest-majority setting that is reasonable for many real use cases [20,25,26,28,101]. Looking ahead, in §7 we will demonstrate that SECRECY's optimizations make secret sharing competitive in both LAN and WAN settings.

3.1 Replicated secret sharing

SECRECY encodes an ℓ-bit string of sensitive data s by splitting it into three *secret shares* s_1, s_2, and s_3 that individually have the uniform distribution over all possible ℓ-bit strings (for privacy) and collectively suffice to specify s (for correctness). Computing parties are placed on a *logical ring* and each party P_i receives two of the shares s_i and s_{i+1} (i.e., P_1 receives s_1, s_2, P_2 receives s_2, s_3, and P_3 receives s_3, s_1). Hence, any two parties can reconstruct a secret if they collude, but any single party cannot, no matter how powerful it is. SECRECY supports two secret sharing formats (and can also transition from one to the other): *boolean* secret sharing in which $s = s_1 \oplus s_2 \oplus s_3$, where $\oplus$ denotes the bitwise XOR operation, and *additive* or *arithmetic* secret sharing in which $s = s_1 + s_2 + s_3 \bmod 2^\ell$.

3.2 Oblivious primitives

In this section, we provide an overview of the oblivious primitives we use throughout our work. Let s, t be two secrets, and $op(s,t)$ an operation on these secrets. The primitives allow SECRECY servers to start with shares of s and t and jointly compute shares of the result $op(s,t)$ without learning anything about s and t. Each server can use these shares in subsequent operations or send them to the analysts, who can reconstruct the true output of $op(s,t)$. We stress that our oblivious relational operators in §4 are agnostic of the underlying primitives and it would be perfectly possible to implement primitives based on other MPC protocols without affecting the applicability of the optimizations in §5.

Basic operations. When given boolean secret-shared data corresponding to ℓ-bit strings s and t, parties can compute shares of the bitwise XOR $s \oplus t$ locally (i.e., without communication) and shares of the bitwise AND st with synchronization cost equal to $C_s(\texttt{AND}) = 1$ round of communication. The operation cost is the same in both cases, i.e., $C_o(\texttt{XOR}) = C_o(\texttt{AND}) = \ell$ (we consider 1-bit boolean operations to have unit cost). Similarly, given two secrets u and v that have been arithmetically shared, parties can compute shares of the sum $u + v$ locally (similarly to XOR) and shares of the product $u \cdot v$ with 1 communication round (similarly to the bitwise AND operation).

Mixed-mode operations. The above boolean and arithmetic operations are universal and can be used to compute *any* function. Moreover, there exist well-known constructions of several specific operations with fast instantiations based on boolean and/or arithmetic sharing. In SECRECY we implement several such oblivious operations: (in)equality, a compare-and-swap multiplexer, boolean addition with a ripple-carry adder, boolean-to-arithmetic conversion, and more.

For details on the SECRECY primitives, please refer to §A.1.

4 SECRECY operators and cost model

In this section, we first provide an overview of oblivious operators in SECRECY along with their asymptotic costs (§4.1) and the costs of their composition under MPC (§4.2). We then explain how SECRECY computes exact plan costs in §4.3. Although, in practice, the SECRECY planner uses the detailed cost formulas from Appendix A, knowledge of the asymptotic costs is sufficient to follow the optimizations in §5.

4.1 Oblivious relational operators

SECRECY supports a rich class of oblivious relational operators: `SELECT`, `PROJECT`, `(SEMI-)JOIN`, `GROUP-BY`, `DISTINCT`, and `ORDER-BY` with `LIMIT`. It also supports the following aggregations under MPC: `COUNT`, `SUM`, `MIN/MAX`, and global `AVG`.

All operators have the same semantics as their plaintext counterparts but their control flow is data independent; in practice, this means that the SECRECY code does not have any `if` statements that depend (either directly or indirectly) on the input data. At a high level, oblivious selection requires a linear scan over the input relation, join and semi-join operators require a nested-loop over the two inputs, whereas order-by, distinct, and group-by are based on a sorting network.

In all cases, operator predicates can be arbitrary logical expressions with atoms that may also include arithmetic expressions $(+, \times, =, >, <, \neq, \geq, \leq)$ and are evaluated under MPC

Operator	#operations (#messages)	#communication rounds
SELECT	$O(n)$	$O(1)$
JOIN	$O(n \cdot m)$	$O(1)$
SEMI-JOIN	$O(n \cdot m)$	$O(\log m)$
ORDER-BY	$O(n \cdot \log^2 n)$	$O(\log^2 n)$
DISTINCT	$O(n \cdot \log^2 n)$	$O(\log^2 n)$
GROUP-BY	$O(n \cdot \log^2 n)$	$O(\log^2 n)$
MASK	$O(n)$	$O(1)$

Table 1: Summary of operation (C_o) and synchronization costs (C_s) for general oblivious relational operators *w.r.t.* the cardinalities (n, m) of the input relation(s). The asymptotic number of operations equals the asymptotic number of messages per computing party, as each individual operation on secret shares involves a constant number of message exchanges under MPC. Independent messages can be batched in rounds as shown in the rightmost column.

using the oblivious primitives of §3.2. All operators except PROJECT and ORDER-BY append a new attribute to each record of their input relation that stores a (secret-shared) *valid bit*: this bit denotes whether the record belongs to the output of the operator and is computed under MPC.

Table 1 shows the asymptotic operation and synchronization costs per operator with respect to the input size. MASK is a special operator used by SECRECY to hide records (with "garbage" values) upon a condition. The formal operator semantics and their exact costs are given in §A.2.

4.2 Composing oblivious operators

We define the composition of two operators as applying the second operator to the output of the first. One merit of our approach is that all operators of §4.1 reveal nothing about their output or access patterns and can be arbitrarily composed into an *end-to-end oblivious* plan without special treatment.

Let op_1 and op_2 be two SECRECY operators. In general, the composition $op_2(op_1(R))$ has an *extra cost* (additional to the cost of applying the operators op_1 and op_2) as it requires evaluating under MPC a logical expression e_c for each generated tuple. We define the *composition cost* of $op_2(op_1(R))$ as the cost of evaluating e_c on all records generated by op_2. The expression e_c depends on the types of operators. For example, composing two selections, each one appending a valid bit to the input relation, requires ANDing the two bits for each record. Table 2 shows the asymptotic composition costs for different operator pairs. The detailed costs are given in §A.3.

Note that applying the distinct operator to the output of a selection, a group-by or a (semi-)join requires a linear number of rounds. This is a significant increase over the $O(\log^2 n)$ rounds required by distinct when applied to a base relation (cf. Table 1). In §5.2, we propose an optimization that reduces the cost of these compositions to a logarithmic factor.

4.3 Computing optimal plan costs

SECRECY's query planner is based on a typical bottom-up dynamic programming algorithm [98] that computes optimal

Operator pair(s)	#rounds
{SELECT, (SEMI-)JOIN, GROUP-BY, DISTINCT} → DISTINCT	$O(n)$
DISTINCT → {SELECT, (SEMI-)JOIN}	$O(1)$
SELECT ↔ (SEMI-)JOIN	$O(1)$
GROUP-BY → {SELECT, (SEMI-)JOIN}	$O(1)$
{SELECT, (SEMI-)JOIN, DISTINCT, GROUP-BY} → GROUP-BY	$O(\log^2 n)$

Table 2: Summary of composition costs (C_c) in number of rounds for pairs of operators in SECRECY *w.r.t* the number of generated records (n). Arrows denote the order of applying the two operators. Composition incurs a small constant number of boolean operations per record, so its cost in number of operations is $O(n)$ in all cases.

plans based on our analytical cost model and a set of transformation rules that we present in §5. The algorithm identifies all operators in the input query and proceeds in stages: at each stage it creates bigger plans by adding a new operator to sub-plans from the previous stage. Initially, the set of possible sub-plans includes scans of the input relations. When creating a new (sub-)plan, the algorithm checks for all applicable transformation rules and applies them exhaustively to generate equivalent (sub-)plans with lower cost. The cost of a plan is computed as follows. Each time an operator op is added to a sub-plan, SECRECY computes the operation and synchronization costs $C_o(op)$ and $C_s(op)$. If the operator is applied to the output of another operator, SECRECY also computes the composition cost C_c. To do so, it augments the current plan with a special operator $op_{e_c}(op_i(op_j(..)))$ that applies the composition predicate e_c (§4.2). $C_o(op_{e_c})$ and $C_s(op_{e_c})$ amount to the cost of composing the operators op_i and op_j in number of operations and rounds respectively. For a plan with k operators, the total cost is $\sum_{i=1}^{k} \alpha C_o(i) + \beta C_s(i)$, where α, β are parameters of the deployment. The algorithm returns the plan with the minimum cost from the final stage.

5 SECRECY optimizations for relational MPC

Here we present the optimizations we introduce in SECRECY to speed up MPC query execution: (i) logical transformation rules, such as operator reordering and decomposition (§5.1), (ii) physical optimizations, such as operator fusion, vectorized primitives and message batching (§5.2), and (iii) secret-sharing optimizations that further reduce the number of communication rounds for certain operators (§5.3). Table 3 summarizes the notation used in the remainder of the paper.

Target queries. Our work focuses on collaborative analytics under MPC where two or more data owners want to outsource queries on their collective data without compromising privacy. We consider all inputs as sensitive and assume that data owners wish to protect their raw data and avoid revealing attributes of base relations in query results. For example, employing MPC to compute a query that includes patient names along with their diagnoses in the SELECT clause is pointless. Thus, we target queries that return global or per-group aggregates and/or distinct results, as in prior works.

Symbol	Description
ℓ	length of the share representation in bits
R, S	relations with cardinality $\|R\|$ and $\|S\|$
$\sigma_\phi(R)$	selection with predicate ϕ
$R \bowtie_\theta S$	join with predicate θ
$R \ltimes_\theta S$	left semi-join with predicate θ
$\delta_a(R)$	distinct operator on attribute a
$\gamma^g_a(R)$	group-by operator on attribute a with aggregation function g
$s_{\uparrow a}(R)$	sort on attribute a (ascending)

Table 3: Notation used in the paper

5.1 Logical transformation rules

SECRECY uses three types of logical transformations that reorder and decompose operators to reduce the MPC costs:

5.1.1 Blocking operator push-down

Blocking oblivious operators (`GROUP-BY`, `DISTINCT`, `ORDER-BY`) materialize and sort their entire input before producing any output tuple. Contrary to a plaintext optimizer that would most likely place sorting after selective operators, in MPC we have an incentive to push blocking operators down, as close to the input as possible. Since oblivious operators do not reduce the size of intermediate data, sorting the input is clearly the best option. Blocking operator push-down can provide considerable performance improvements in practice, even if the asymptotic costs do not change. As an example, consider the rule that pushes `ORDER-BY` before a selection, i.e., $s_{\uparrow a}(\sigma_\phi(R)) \longrightarrow \sigma_\phi(s_{\uparrow a}(R))$. Although this rule would not generate a more efficient plan in plaintext evaluation, it does so in the MPC setting. This is because the operations required by the oblivious `ORDER-BY` depend on the cardinality and the number of attributes of the input relation. Applying the selection after the order-by reduces the actual (but not the asymptotic) operation cost, as σ_ϕ appends one attribute to R.

Applicability. Rules in this class are valid relational transformations with no special applicability conditions under MPC.

5.1.2 Join push-up

The second class of rules leverage the fact that `JOIN` is the only operator whose output is larger than its input. Based on this, we have an incentive to perform joins as late as possible in the query plan so that we avoid applying other operators to join results, especially those that require materializing the join output. For example, placing a blocking operator after a join requires sorting the cartesian product of the input relations, which increases the operation cost of the blocking operator to $O(n^2 \log^2 n)$ and the synchronization cost by 4×.

Example. Consider the following query:

```
Q1: SELECT DISTINCT R.id
    FROM R, S
    WHERE R.id = S.id
```

and the rule $\delta_{id}(R \bowtie_{id=id} S) \longrightarrow \delta_{id}(R) \bowtie_{id=id} \delta_{id}(S)$. Let R and S have the same cardinality n. A plan that applies `DISTINCT` after the join operator requires $O(n^2 \log^2 n)$ operations. On the other hand, pushing `DISTINCT` before `JOIN` reduces the operation cost to $O(n^2)$ and the composition cost from $O(n^2)$ to $O(1)$ in number of rounds. The asymptotic synchronization cost is the same for both plans, i.e. $O(\log^2 n)$, but the actual number of rounds when `DISTINCT` is pushed before `JOIN` is 4× lower.

Applicability. Rules in this class have the same applicability conditions as similar rules for plaintext queries [42, 113], even though their goal is different. In our setting, the re-orderings do not aim to reduce the size of intermediate data. In fact, a plan that applies `DISTINCT` on a `JOIN` input produces exactly the same amount of intermediate data as a plan where `DISTINCT` is placed after `JOIN`, yet our analysis reveals that the second plan has higher MPC costs.

1 $s_{\uparrow a_\theta \uparrow a_k}(R)$; `//sort input relation R on` a_θ, a_k
2 let $d \leftarrow |R|/2$; `//Distance of tuples to aggregate`
3 **while** $d \geq 1$ **do**
4 **for** *each pair of tuples* (t_i, t_{i+d}), $0 \leq i < |R| - d$, **do**
 `//Are tuples in the same group?`
5 let $b \leftarrow t_i[a_k] \stackrel{?}{=} t_{i+d}[a_k]$;
 `//Are tuples in semi-join output too?`
6 let $b_c \leftarrow b \wedge t_i[a_\theta] \wedge t_{i+d}[a_\theta]$; `//`$b_c$ `is a bit`
 `//Oblivious aggregation via multiplexing`
7 $t_i[a_g] \leftarrow b_c \cdot \left(t_i[a_g] + t_{i+d}[a_g]\right) + (1 - b_c) \cdot t_i[a_g]$;
8 $t_{i+d}[a_v] \leftarrow \neg b_c$; `//`$a_v$ `is the valid bit`
9 mask t_{i+d} when $t_{i+d}[a_v] = 0$;
10 $d = d/2$;
11 mask remaining tuples with $t[a_v] = 0$ and shuffle R;

Algorithm 1: 2nd phase of Join-Aggregation decomposition

5.1.3 Join-Aggregation decomposition

Consider a query plan where a `JOIN` on attribute a_j is followed by a `GROUP-BY` on another attribute $a_k \neq a_j$. In this case, pushing the `GROUP-BY` down does not yield a semantically equivalent plan. Still, we can optimize the plan by decomposing the aggregation in two phases and push the first and most expensive phase of `GROUP-BY` before the `JOIN`.

Let R, S be the join inputs, where R includes the group-by key a_k. The first phase of the decomposition sorts R on a_k and computes a semi-join on a_j that appends two attributes to R: the valid bit a_θ introduced by the semi-join, and a second attribute a_g that stores the result of a partial aggregation[1] (we come back to this later).

In the second phase, we compute the final aggregates per a_k using Algorithm 1, which takes into account the attribute a_θ and updates the partial aggregates a_g in-place using odd-even aggregation. The decomposition essentially replaces the join

[1] In case the aggregation function is `AVG`, we need to keep the value sum (numerator) and count (denominator) as separate secret-shared attributes in R.

with a semi-join and a partial aggregation in order to avoid performing the aggregation on the cartesian product $R \times S$. This way, we significantly reduce the number of operations and communication rounds, but also ensure that the space requirements remain bounded by $|R|$, since the join output is not materialized. Note that this optimization is fundamentally different than performing a partial aggregation in the clear (by the data owners) and then computing the global aggregates under MPC [22, 95]; in our case, all data are secret-shared amongst parties and *both* phases are under MPC.

Example. Consider the following query:

Q2:
```
SELECT R.a_k, COUNT(*)
FROM R, S
WHERE R.id = S.id
GROUP BY R.a_k
```

Let R and S have the same cardinality n. The plan that applies `GROUP-BY` to the join output requires $O(n^2 \log^2 n)$ operations and $O(\log^2 n)$ communication rounds. When decomposing the aggregation $\gamma_{a_k}^{\texttt{COUNT(*)}}$ in two phases, the operation cost is reduced to $O(n^2)$ and the synchronization cost is 4× lower. The space requirements are also reduced from $O(n^2)$ to $O(n)$. In our example, the partial aggregation corresponds to the function $t[a_\theta] = \sum_{\forall t' \in S} \theta(t, t'), t \in R$, where $\theta(t, t') := t[\texttt{id}] \stackrel{?}{=} t'[\texttt{id}]$. Similar partial aggregations can be defined for `SUM`, `MIN/MAX`, and `AVG`.

Decomposition with DISTINCT. A similar idea can also be employed when the join is followed by a `DISTINCT`. The transformation rule in this case is $\delta_{R.a}(R \bowtie_\theta S) \rightarrow \delta'_a(s_{\uparrow R.a_\theta, \uparrow R.a}(R \ltimes_\theta S))$, where a_θ denotes the semi-join bit and $\delta'(\cdot)$ is the final phase of distinct that compares adjacent tuples with $a_\theta = 1$. For example, the plan $\delta_{R.a}(R \bowtie_{b=b} S)$ can be replaced with the equivalent plan $\delta'_{R.a}(s_{\uparrow R.a_\theta, \uparrow R.a}(R \ltimes_{b=b} S))$ to reduce the operation cost from $O(n^2 \log^2 n)$ to $O(n^2)$ and the synchronization cost from $O(n^2)$ to $O(\log^2 n)$.

Applicability. The decomposition technique we described is applicable to any θ-join followed by (i) a `GROUP-BY` with aggregation or (ii) a `DISTINCT` operator, under the condition that the group-by or distinct keys belong to one join input.

5.2 Physical optimizations

We now describe a set of physical optimizations in SECRECY.

5.2.1 Predicate fusion

Fusion is a common optimization in plaintext query planning, e.g., when predicates of multiple filters are merged and executed by a single operator. Fusion has been recently used to speed up secure ML pipelines in Cerebro [117] and is also applicable to oblivious relational operators. In our setting, fusion is achieved by identifying independent operations that can be executed efficiently within the same communication round. For example, if the equality check of an equi-join and a selection are independent of each other, a fused operator requires $\lceil \log \ell \rceil + 1$ rounds instead of $2\lceil \log \ell \rceil + 1$ (cf. §A). Next, we describe a somewhat more interesting case of fusion.

5.2.2 Operator fusion

Recall that applying `DISTINCT` after `SELECT` requires n communication rounds (§4.2). We can avoid this overhead by fusing the two operators in a different way, that is, sorting the input relation on the selection bit first and then on the distinct attribute. Sorting on two (instead of one) attributes adds a small constant factor to each oblivious compare-and-swap operation, hence, the asymptotic complexity of the sorting step remains the same. When distinct is applied to the output of other operators, including selections and (semi-)joins, this physical optimization keeps the number of rounds required for the composition low.

Example. Consider the following query:

Q3:
```
SELECT DISTINCT id
FROM R
WHERE a_k = 'c'
```

Fusing the distinct and selection operators reduces the number of communication rounds from $O(n)$ to $O(\log^2 n)$, as if the distinct operator was applied only to R (without a selection). `DISTINCT` can be fused with a join or a semi-join operator in a similar way. In this case, the distinct operator takes into account the (semi-)join bit.

5.2.3 Vectorization and message batching

In secret sharing protocols, non-local operations require exchanging very small messages. Applying multiple such independent operations in a vectorized fashion and exchanging the respective messages in bulk improves performance tremendously. Consider applying a selection with an equality predicate on a relation with n tuples. Performing oblivious equality on one tuple requires $\lceil \log \ell \rceil$ rounds. Applying the selection tuple-by-tuple and sending messages eagerly (as soon as they are generated) results in $n \cdot \lceil \log \ell \rceil$ rounds. Instead, if we apply independent selections across the entire relation and exchange messages in bulk, we can reduce the total synchronization cost to $\lceil \log \ell \rceil$. We have designed all SECRECY primitives to apply vectorization and message batching by default, otherwise the cost of secret sharing is prohibitive. Costs in Tables 1 & 2 already take message batching into account.

5.3 Secret-sharing optimizations

Here we propose optimizations that take advantage of mixed-mode MPC protocols which permit both arithmetic and boolean computations. While SECRECY uses boolean secret sharing for most operations, computing arithmetic expressions or aggregations like `COUNT` and `SUM` on boolean shares

requires using a ripple-carry adder (RCA), which in turn requires many communication rounds. Performing these operations on additive shares would require no communication, but converting shares from one format to another can be expensive. Below, we describe two optimizations that avoid the RCA in aggregations and predicates with constants.

5.3.1 Dual sharing

The straight-forward approach of switching from boolean to additive shares (and vice versa) based on the type of operation does not pay off; the conversion itself relies on RCA, which has to be applied twice to switch to the other representation and back. The cost-effective way would be to evaluate logical expressions using boolean shares and arithmetic expressions using additive shares. However, this is not always possible because arithmetic and boolean expressions in oblivious queries often need to be composed into the same formula. We mitigate this problem using a dual secret-sharing scheme.

Recall the example query **Q2** from §5.1.3 that applies an aggregation function to the output of a join according to Algorithm 1. The attribute a_θ in Algorithm 1 is a single-bit attribute denoting that the respective tuple is included in the join result. During oblivious evaluation, each party has a boolean share of this bit that is used to compute the arithmetic expression in line **6**. The naïve approach is to evaluate the following equivalent logical expression directly on the boolean shares of b_c, $t_i[a_g]$, and $t_{i+d}[a_g]$:

$$t_i[a_g] \leftarrow b_\ell \wedge \texttt{RCA}\Big(t_i[a_g], t_{i+d}[a_g]\Big) \;\oplus\; \overline{b}_\ell \wedge t_i[a_g]$$

where RCA is the oblivious ripple-carry adder primitive, b_ℓ is a string of ℓ bits (the length of a_g) all of which are set equal to b_c, and $\overline{b}_\ell$ is the binary complement of b_ℓ. Evaluating the above expression requires ℓ communication rounds for RCA plus two more rounds for the logical ANDs ($\wedge$). On the contrary, SECRECY evaluates the equivalent formula in line **6** of Algorithm 1 in four rounds (independent from ℓ) as follows. First, parties use arithmetic shares for the attribute a_g to compute the addition locally. Second, each time they compute the bit b_c in line **5**, they exchange boolean as well as arithmetic shares of its value. To do this efficiently, we rely on the single-bit conversion protocol also used in CrypTen [70], which requires two rounds of communication. Having boolean and arithmetic shares of b_c allows SECRECY to use it in boolean and arithmetic expressions without paying the cost of RCA.

5.3.2 Proactive sharing

The previous optimization relies on b_c being a single bit. In many cases, however, we need to compose boolean and additive shares of arbitrary values. Representative examples are join predicates with arithmetic expressions on boolean shares, e.g. $(R.a - S.a \geq c)$, where a is an attribute and c is a constant. We can speedup the oblivious evaluation of such predicates by proactively asking the data owners to send shares of the expression results. In the previous example, if parties receive boolean shares of $S.a + c$ they can avoid computing the boolean addition with RCA. A similar technique is also applicable for selection predicates with constants. In this case, to compute $a > c$, if parties receive shares of $a - c$ and $c - a$, they can transform the binary equality to a local comparison with zero. Note that proactive sharing is fundamentally different than having data owners perform local filters or pre-aggregations prior to sharing. In the latter case, the computing parties might learn the selectivity of a filter or the number of groups in an aggregation (if results are not padded). In our case, parties simply receive additional shares and will not learn anything about the intermediate query results.

5.4 Generality of optimizations

The logical and physical query optimizations constructed in this work (§5.1-5.2) apply generally to any mixed-mode MPC protocol that supports the primitives we describe in §3.2. This includes protocols that remain secure in the face of a malicious adversary who can deviate from the protocol arbitrarily (e.g., [46, 71, 89]), and (authenticated) garbled circuit protocols [109, 114] combined with conversions to arithmetic secret sharing [50, 89] as needed. SECRECY can also support alternative instantiations of oblivious primitives with different cost profiles, such as constant-round equality and comparisons with higher operation costs [48, 86].

While the secret-sharing optimizations of §5.3 are specific to SECRECY's underlying MPC protocol (§3.1), we expect that similar techniques can be developed also for other protocols. Extending the SECRECY planner to consider the cost profiles of various building blocks is an exciting avenue for future work. We provide a formal discussion of generality in Appendix B.

6 SECRECY implementation

Despite a rich open-source ecosystem of general-purpose MPC frameworks [62], we found that existing tools either lack support for general relational operations (with θ-predicates) or cannot effectively amortize network I/O. For these reasons, we implemented SECRECY in C/C++, entirely from scratch. We designed our secure primitives to operate directly on relations and we also built a library of general oblivious relational operators that can be combined into arbitrary query plans.

System overview. Figure 2 shows the SECRECY architecture and software stack. Data analysts submit queries through a client application that exposes a SQL interface and provides a query planner that performs query rewriting and cost-based optimization. Data owners use the secret-sharing generation module to distribute random shares of their data to the computing parties. Computing parties can be deployed on premises,

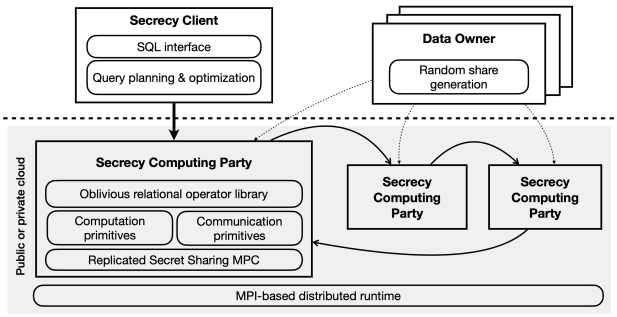


Figure 2: The SECRECY system consists of (i) a client application that can be used by data analysts to submit queries, (ii) a data owner application to generate and distribute secret shares, and (iii) three computing parties that execute queries under MPC.

in a hybrid cloud, or across multiple clouds. To automate cloud deployment we use Ansible [7]. The parties' software stack consists of (i) a custom implementation of the replicated secret sharing protocol, (ii) a library of secure computation and communication primitives, and (iii) a library of oblivious relational operators. The distributed runtime and communication layer are based on MPI [5]. Each party is a separate MPI process that handles both computation and communication.

Operator pipelining. SECRECY relational operators and secure primitives are designed to process table rows in batches. The batch size is configurable and allows SECRECY to compute expensive operators, such as joins, with full control over memory requirements. While batching does not reduce the total number of operations, we leverage it to compute on large inputs in a pipelined fashion, without running out of memory or switching to a disk-based evaluation.

Query planning and execution. Upon startup, the parties establish connections to each other and learn the process IDs of other parties. Next, they receive shares for each input relation from the data owners. Queries are specified either in SQL (and go through query planning) or in a declarative DSL that allows seamless operator composition by abstracting MPC details. For SQL parsing we use the Hyrise parser [12]. SECRECY's planner generates the optimal query plan as explained in §4.3. To evaluate a query, parties execute the same oblivious physical plan on their random shares and return the results to the designated client. We use a 64-bit share representation by default, so $\ell = 64$ (cf. Table 3).

7 Experimental evaluation

Our experimental evaluation is structured into four parts:

Benefits of query optimization. In §7.2, we evaluate the benefits of SECRECY's optimizations on eight real and synthetic queries. We show that SECRECY's cost-based optimizer reduces the runtime of complex queries by up to three orders of magnitude both in a LAN and a multi-cloud setting.

Performance on real and synthetic queries. In §7.3 we evaluate SECRECY's performance as input sizes grow. We use queries that include selections, group-by, distinct, semi-join, and theta-joins with both equality and inequality predicates. Our results demonstrate that SECRECY can scale to millions of input rows and evaluate complex queries in reasonable time with modest use of resources.

Micro-benchmarks. In §7.4, we evaluate individual logical, physical, and secret-sharing optimizations on the three queries from §5.1-5.3. Our results demonstrate that pushing down blocking operators reduces execution time by up to 1000× and enables queries to scale to 100× larger inputs. Further, we show that operator fusion and dual sharing improve execution time by an order of magnitude in the WAN setting.

Comparison with state-of-the-art frameworks. In §7.5, we compare SECRECY with SMCQL [22] and the 2-party semi-honest version of EMP [108]. We choose SMCQL (the ORAM-based version) as the only open-source relational framework with semi-honest security and no leakage. We also choose the EMP library since it is used by all recent systems, namely Shrinkwrap [23], SAQE [24], a new version of SMCQL, and Senate [95]. Although none of these systems is publicly available, they all build their relational MPC engines on top of EMP. We show that SECRECY outperforms them both and can comfortably process much larger datasets.

We provide additional micro-benchmarks and experiments with EMP in Appendix D.

7.1 Evaluation setup

We use three cloud deployments: (i) `AWS-LAN` uses an *EC2 r5.xlarge* instance per party in the `us-east-2` region, (ii) `AWS-WAN` distributes parties across `us-east-2` (Ohio), `us-east-1` (Virginia), and `us-west-1` (California), and (iii) `MULTI-CLOUD` distributes parties across three different cloud providers, namely AWS (Ohio), Google Cloud (South Carolina), and Azure (Virginia). VMs have 32GB of memory and run Ubuntu `20.04`, `C99`, `gcc 5.4.0`, and `MPICH 3.3.2`. Measurements are averaged over at least three runs and plotted in log-scale, unless otherwise specified.

Queries. We use 11 queries for evaluation, including five real-world queries from previous MPC works [22–24, 95, 105]. Three are medical queries [22]: *Comorbidity* returns the ten most common diagnoses of individuals in a cohort, *Recurrent C.Diff.* returns the distinct ids of patients who have been diagnosed with `cdiff` and have two consecutive infections between 15 and 56 days apart, and *Aspirin Count* returns the number of patients who have been diagnosed with heart disease and have been prescribed aspirin after the diagnosis was made. We also use queries from other MPC application areas [95]: *Password Reuse* asks for users with the same password across different websites, while *Credit Score* asks for persons whose credit scores across different agencies have

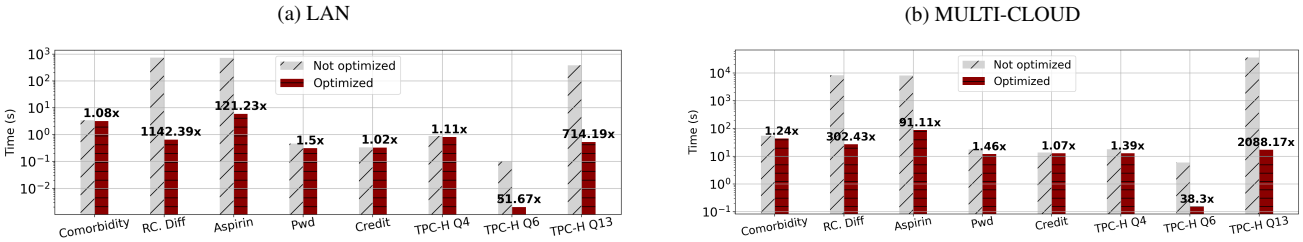


Figure 3: SECRECY end-to-end performance when optimizations are enabled (`Optimized`) and disabled (`Not optimized`) for real and synthetic queries. Logical and physical optimizations result in over 1000× lower execution times, while secret-sharing optimizations improve performance by up to ~ 52×. Not optimized plans still use vectorization and message batching (§5.2.3), otherwise the cost of secret sharing is prohibitive.

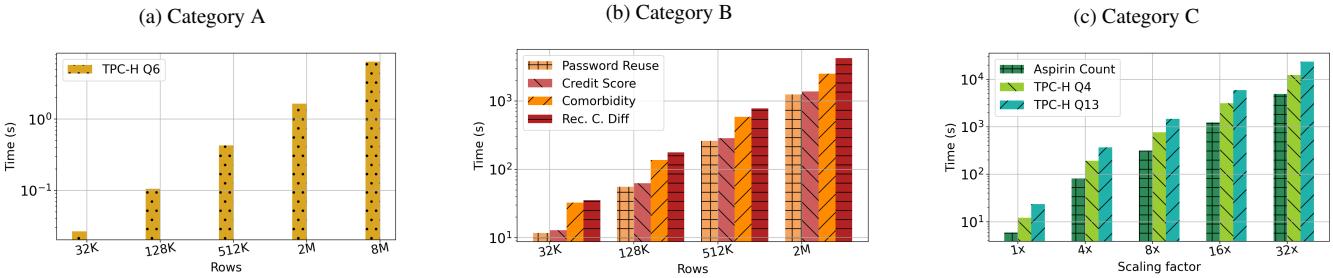


Figure 4: Scaling behavior of optimized real and synthetic queries on SECRECY

significant discrepancies in a particular year. In addition to the real-world queries, we use the TPC-H queries (**Q4**, **Q6**, **Q13**) [103] that have been used in SAQE [24]. Finally, to evaluate the performance gains from each optimization in isolation, we use **Q1**, **Q2**, **Q3** from §5.1-5.3.

Datasets. All experiments use randomly generated tables with 64-bit values. Note that SECRECY's MPC protocol assumes a fixed-size representation of shares that is implementation-specific and could be increased to any 2^k value. We also highlight that using random inputs is no different than using real data, as all operators are oblivious and the data distribution does not affect the amount of computation or communication. Regardless if the input is real or random, parties compute on secret shares, which are by definition random. In all experiments we designate one party as the data owner who distributes shares and learns the results. SECRECY uses exactly three computing parties; therefore, the number of data owners and analysts does not affect query performance, only the cumulative input size does.

7.2 Benefits of query optimization

We compare the performance of 8 queries optimized by SECRECY with that of plans without the optimizations of §5. For a fair comparison, we implement baseline plans using SECRECY's batched operators. Although this favors the baseline, the communication cost of MPC is otherwise prohibitive and queries cannot scale beyond a few hundred input rows. We execute each plan with $1K$ rows per input relation. For Q4 (resp. Q13), we use $1K$ rows for `LINEITEM` (resp. `ORDERS`) and maintain the size ratio with the other input relation as specified in the TPC-H benchmark. For *Comorbidity*, we use a cohort of 256 patients. We run this experiment on `AWS-LAN` and `MULTI-CLOUD` and present the results in Figure 3.

In the LAN setting, SECRECY achieves the highest speedups for *Recurrent C.Diff.*, *Aspirin Count*, and Q13, that is, 1142×, 121×, and 714× lower execution times, respectively. Optimized plans for these queries leverage join push-up (*Aspirin Count*), fusion (*Recurrent C.Diff.*), and join-aggregation decomposition (Q13). The optimized plans for *Comorbidity*, *Password Reuse*, Q4, and Q6 leverage dual and proactive sharing, achieving up to 52× speedup compared to the baseline. Finally, the *Credit Score* query leverages dual sharing which, in this case, provides a modest improvement. SECRECY achieves significant speedups in the wide area, too. The performance improvement is higher for *Comorbidity*, Q4, and Q13 in the multi-cloud setting, as these queries leverage optimizations that primarily reduce the synchronization cost. We evaluate the benefit of individual optimizations in §7.4.

7.3 Performance on real and synthetic queries

We now run the optimized plans with increasing input sizes in `AWS-LAN` and report total execution time. For these experiments, we group queries into three categories of increasing complexity. *Category A* includes queries with selections and global aggregations, *Category B* includes queries with select and group-by or distinct operators, and *Category C* includes queries with select, group-by and (semi-)join operators. Figure 4 presents the results.

Q6 in *Category A* consists of five selections and a global aggregation. It requires minimal communication that is independent of the input relation cardinality. As a result, it scales comfortably to large inputs and takes ~ $6s$ for $8M$ rows.

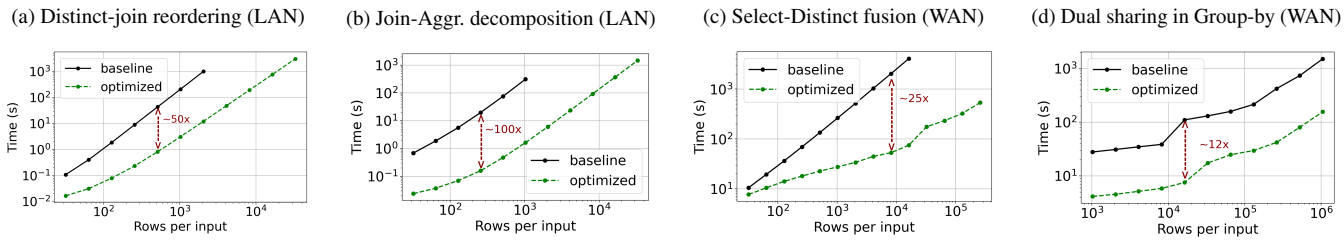


Figure 5: Performance improvement of individual optimizations applied by the SECRECY planner

Queries in *Category B* scale to millions of input rows as well. The cost of these queries is dominated by the oblivious group-by and distinct operators. At $2M$ rows, *Recurrent C.Diff.* completes in $\sim 1.2h$ and *Password Reuse* in $\sim 20min$.

The cost of queries in *Category C* is dominated by joins and semi-joins. The size ratio between the two inputs of each query is different: for Q4 and Q13, we use the ratio specified in the TPC-H benchmark whereas, for *Aspirin Count*, we use inputs of equal size. In Figure 4c, Scaling factor 1× corresponds to $1K$ rows for the small input. As we increase the input sizes, we always keep their ratio fixed. At scaling factor 32×, the most expensive query is Q13, which is optimized with join-aggregation decomposition and takes $\sim 6.5h$ on $295K$ rows. At the same scaling factor, Q4 completes in $\sim 3.4h$ on $164K$ rows, and *Aspirin Count* in $\sim 1.3h$.

While MPC protocols remain expensive for real-time queries, our results demonstrate that offline collaborative analytics on millions of records entirely under MPC are viable.

7.4 Micro-benchmarks

We now use the queries of §5 (**Q1**, **Q2**, **Q3**) to evaluate the impact of SECRECY's optimizations in isolation. We run each query with and without the particular optimization and measure total execution time. Distinct-join reordering and join-aggregation decomposition primarily reduce the operation cost and we evaluate them in `AWS-LAN`. Fusion and dual sharing reduce the synchronization cost and we evaluate them in `AWS-WAN`. Figure 5 shows the results.

Distinct-Join reordering. The optimized plan of Q1 pushes the `JOIN` after `DISTINCT` and, thus, only sorts a relation of n rows instead of n^2. Figure 5a shows that the optimized plan is up to 50× faster than the baseline, which runs out of memory for even modest input sizes.

Join-Aggregation decomposition. The baseline plan of Q2 materializes the result of the join and then applies the grouping and aggregation. Instead, the optimized plan decomposes the aggregation in two phases (cf. §5.1.3). As shown in Figure 5b, this optimization provides 100× lower execution time than that of the baseline plan. Further, the baseline plan runs out of memory for inputs larger than $1K$ rows.

Operator fusion. The baseline plan of Q3 applies the oblivious selection before `DISTINCT`, while the optimized plan fuses the two operators and performs the `DISTINCT` computation in bulk (cf. §5.2.2). Figure 5c shows that this optimization provides more than 25× speedup for large inputs and allows the query to scale to much larger inputs.

	Comorbidity	Recurrent C. Diff.	Aspirin Count
SMCQL	91s	358s	365s
SECRECY	0.083s	0.092s	0.171s

Table 4: SMCQL and SECRECY execution times in LAN for the three medical queries from [22] on 25 tuples per input relation.

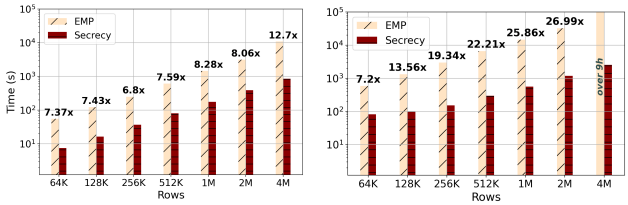


Figure 6: Performance comparison of the oblivious sort operator on EMP and SECRECY in LAN (left) and WAN (right).

Dual sharing. We also evaluate SECRECY's ability to switch between arithmetic and boolean sharing to reduce communication costs for certain operations. For this experiment, we compare the run-time of the optimized `GROUP-BY-COUNT` operator (cf. §5.3) to that of a baseline operator that uses boolean sharing only and, hence, relies on the ripple-carry adder to compute the `COUNT`. Figure 5d plots the results. The baseline operator is 10× slower than the optimized one, as it requires 64 additional rounds of communication per input row.

7.5 Comparison with other MPC frameworks

Existing 3-party frameworks [62] are either proprietary, e.g. [27], or they only support specific operators, such as unique-key joins [82, 99], that cannot be used for any of the queries we consider. We stress that EMP and SMCQL use 2-party garbled circuit protocols that are not directly comparable with SECRECY's. The purpose of these experiments is to showcase the end-to-end performance of the available solutions for relational MPC and not to compare the underlying protocols.

Comparison with SMCQL. In the first set of experiments, we aim to reproduce the results presented in SMCQL [22, Fig. 7] on our experimental setup. We run the three medical queries on SMCQL and SECRECY, using a sample of 25

rows per data owner (50 in total), and present the results in Table 4. We use the plans and default configuration of protected and public attributes, as in the SMCQL project repository. SECRECY is over 1000× faster than SMCQL in all queries.

Comparison with EMP. EMP is a general-purpose MPC framework and does not provide relational operators or query planning. Nevertheless, we include a comparison with EMP because it is the MPC library underpinning several non-open-source relational frameworks (e.g., Shrinkwrap, SAQE, and Senate). For these experiments, we use the oblivious sort operation from the EMP repository [108] that has the same asymptotic complexity with the respective SECRECY sort. Figure 6 shows the results in `AWS-LAN` and `AWS-WAN` for input sizes ranging from $64K$ to $4M$ rows. The performance gap between SECRECY and EMP is significant. SECRECY is up to 12.7× faster in LAN ($\sim 3h$ vs $14min$ for $4M$ input rows). In the WAN setting, SECRECY sorts $4M$ rows in $42min$, while EMP could not complete the computation within $9h$.

8 Related Work

Relational MPC systems. We distinguish two lines of work in this space that are often combined. The first line targets peer-to-peer deployments and reduces multi-party computation by pushing parts of the query to data owners (for plaintext evaluation) or executing the protocol within subsets of the computing parties [13, 22, 44, 95, 105]. The second line includes systems that trade off MPC performance with controlled information leakage [23, 24, 64, 105, 111]. We summarize each system's preconditions and guarantees in Table 5 and provide more details in Appendix C. Function secret sharing in Splinter [106] allows for private queries on public data, which is the opposite to our goal.

Our approach has several advantages over, and is also complementary with, many of the prior techniques. SECRECY's optimizations are agnostic to data ownership and retain the full security guarantees of MPC, merely optimizing its execution. More importantly, this work provides a strong foundation for a unified query optimization framework that can accommodate multi-cloud, peer-to-peer, and hybrid deployments. Prior techniques can be ported into SECRECY by plugging in appropriate cost functions and query transformation rules. For example, pushing parts of the query to data owners, as in Conclave [105], can be done via transformation rules that introduce plaintext operators with certain placement constraints.

Enclave-based approaches. In this line of work, parties process the plaintext data within a physically protected environment. Enclave-based approaches aim to minimize RAM requirements, pad intermediate results, and hide access patterns in untrusted storage. The works by Agrawal et al. [14] and Arasu et al. [19] focus on database queries in this setting. More recent systems such as ObliDB [52], Opaque [116], StealthDB [104], and OCQ [49] rely on Intel's SGX.

Enclave-based systems typically achieve better performance than MPC systems but require different trust assumptions and are susceptible to attacks [33, 35, 36, 59, 74, 75, 107, 112]. Some of these threats can be ameliorated using oblivious operators within the enclave. Our logical optimizations from §5.1 could also be applied in this setting to reduce the number of operations and memory requirements.

System optimizations for MPC. Improving the performance of secure computation via system optimizations is an active research topic. MAGE [73] proposed an interesting technique to reduce the inherent memory overhead of homomorphic encryption and garbled circuits (cf. §3). As SECRECY relies on secret sharing, its memory footprint is small. Instead, secret sharing incurs a higher communication cost, which we amortize using vectorization and message batching (§5.2.3). MPC performance can be further improved by offloading secure primitives to hardware accelerators [54, 55, 79, 102]. Most works in this space focus on ML workloads but similar techniques could also be applied to relational operators.

MPC operators, algorithms, and cost models. Various related works focus on standalone oblivious relational operators, e.g. building group-by from oblivious sort [66], equi-joins [15, 72, 82, 93], or common aggregations [45, 51]. SECRECY is driven by real-world applications that typically require oblivious evaluation of queries with multiple operators. Motivated by similar needs, Wang et al. [110] presented a secure version of the Yannakakis' algorithm, while Ion et al. [65] and Buddhavarapu et al. [34] studied unique-key joins followed by simple aggregations. These works do not provide general cost-based MPC query optimization and they operate in the peer-to-peer setting, where data owners participate in the protocol execution using trusted resources. Recently, CostCO [53] did some nice work on modeling the cost of general MPC programs. Our cost model focuses on relational operators and is tightly integrated with the query planner.

Encrypted DBs. Existing practical solutions in secure database outsourcing [56] operate in a client-server setting and reveal or "leak" information to the database server. Systems based on property-based encryption like CryptDB [96] offer full SQL support and legacy compliance, but each query reveals information that can be used in reconstruction attacks [37, 58, 60, 61, 69, 78, 80, 84]. Systems based on structural encryption [38, 67, 88, 94, 115] provide semantic security that does not eliminate access pattern leaks. SDB [64, 111] uses secret sharing but leaks information to the server whereas Cipherbase [18] relies on a trusted machine. These systems support only one data owner and it would require public-key encryption to evaluate queries that span multiple datasets [31].

Differential Privacy (DP). Systems like DJoin [83], DStress [87], and others [29, 43, 63] use DP to ensure that the out-

Framework	MPC Protocol	Information Leakage	Trusted Party	Query Execution	Main Optimization Objective	Optimization Conditions
Conclave [105]	Secret Sharing / Garbled Circuits	Controlled (Hybrid operators)	Yes	Hybrid	Minimize the use of secure computation	1. Data owners serve as computing parties 2. Data owners provide privacy annotations 3. There exists an additional trusted party
SMCQL [22]	Garbled Circuits / ORAM	No	No[1]	Hybrid	Minimize the use of secure computation	1. Data owners serve as computing parties 2. Data owners provide privacy annotations 3. There exists an honest broker
Shrinkwrap [23]	Garbled Circuits / ORAM	Controlled (Diff. Privacy)	No	Hybrid	Calibrate padding of intermediate results	1. Data owners serve as computing parties 2. Data owners provide privacy annotations and intermediate result sensitivities
SAQE [24]	Garbled Circuits	Controlled (Diff. Privacy)	No	Hybrid	Choose sampling rate for approximate answers	1. Data owners serve as computing parties 2. Data owners provide privacy annotations and differential privacy budgets
Senate [95] [2]	Garbled Circuits	No	No	Hybrid	Reduce joint computation to subsets of parties	1. Data owners serve as computing parties 2. Input or intermediate relations are owned by subsets of the computing parties
SDB [64, 111] [3]	Secret Sharing	Yes (operator dependent)	No	Hybrid	Reduce data encryption and decryption costs	1. Data owner serves as computing party 2. Data owner provides privacy annotations
SECRECY	Rep. Secret Sharing	No	No	End-to-end under MPC	Reduce MPC costs (§ 2.3 and § 4-5)	None

[1] *SMCQL relies on an honest broker that may see protected data in the clear during query evaluation* [22, § 5.1].
[2] *Senate provides security against malicious parties whereas all other systems adopt a semi-honest model.*
[3] *SDB adopts a typical DBaaS model with one data owner and does not support collaborative analytics.*

Table 5: Summary of MPC-based systems for relational analytics. Hybrid execution splits the query plan into a plaintext part (executed by the data owners) and an oblivious part (executed under MPC) and requires data owners to participate in the computation using trusted resources. The rest of the optimizations supported by each system are applicable under one or more of the listed conditions in the rightmost column.

put of a query reveals little about any one input record. This property is independent of (yet symbiotic with) MPC's guarantee that the act of computing the query reveals no more than what may be inferred from its output. The SECRECY primitives from §3.2 can express arbitrary computations and could also be used to add DP noise under MPC. We leave this as future work.

9 Conclusions

This work presents SECRECY, a new system for efficient secure analytics in the cloud with no information leakage. SECRECY can enable new data markets and socially-beneficial data analyses while protecting private data. Our results show that logical optimizations coupled with careful system design can make MPC practical for complex analytics on millions of data records. In the future, we plan to extend SECRECY with multi-objective query optimization that considers cloud fees, data-parallelism via oblivious hashing (e.g. [91, 92]), and support for malicious-secure MPC (e.g., [16, 71]).

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and James Mickens for shepherding the paper. We also thank Jonathan Appavoo, Orran Krieger, Ran Canetti, Leo Reyzin, Azer Bestavros, Kinan Dak Albab, and Ben Getchell for their comments on an early version of this work, Eric Chen for automating the SECRECY deployment, and the MOC Alliance for providing access to their cloud. This work has been partially supported by a Red Hat Collaboratory grant (No. 2022-01-RH02) and a Hariri Institute Focused Research Project Award. M. Varia's work is supported by the DARPA SIEVE program under Agreement No. HR00112020021 and the National Science Foundation under Grants No. 1414119, 1718135, 1801564, and 1931714.

References

[1] Cape Privacy. https://capeprivacy.com. [Online; accessed April 2022].

[2] General Data Protection Regulation (GDPR). https://gdpr.eu/tag/gdpr/. [Online; accessed April 2022].

[3] Health Insurance Portability and Accountability Act (HIPAA). https://www.cdc.gov/phlp/publications/topic/hipaa.html. [Online; accessed April 2022].

[4] HElib. https://github.com/homenc/HElib. [Online; accessed April 2022].

[5] Message Passing Interface (MPI). https://www.mcs.anl.gov/research/projects/mpi/standard.html. [Online; accessed April 2022].

[6] PALISADE. https://gitlab.com/palisade/palisade-release. [Online; accessed April 2022].

[7] Red Hat Ansible Automation Platform. https://www.redhat.com/en/technologies/management/ansible. [Online; accessed April 2022].

[8] SEAL. https://github.com/Microsoft/SEAL. [Online; accessed April 2022].

[9] Secrecy Github Repository. https://github.com/CASP-Systems-BU/Secrecy. [Online; accessed April 2022].

[10] SoK: General-Purpose Compilers for Secure Multi-party Computation. https://github.com/MPC-SoK/frameworks/blob/master/emp/sh_test/test/xtabs.cpp. [Online; accessed April 2022].

[11] The Carbyne Stack: Cloud Native Secure Multiparty Computation. https://carbynestack.io. [Online; accessed April 2022].

[12] The Hyrise Project: C++ SQL Parser. https://github.com/hyrise/sql-parser. [Online; accessed April 2022].

[13] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. In *The Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, 2005.

[14] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. Sovereign joins. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 26, USA, 2006. IEEE Computer Society.

[15] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 86–97, New York, NY, USA, 2003. Association for Computing Machinery.

[16] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority mpc for malicious adversaries – breaking the 1 billion-gate per second barrier. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, pages 843–862, May 2017.

[17] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 805–817, October 2016.

[18] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*, January 2013.

[19] Arvind Arasu and Raghav Kaushik. Oblivious query processing. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 26–37. OpenProceedings.org, 2014.

[20] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.

[21] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex J. Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. RAMPARTS: A programmer-friendly system for building homomorphic encryption applications. In *WAHC@CCS*, pages 57–68. ACM, 2019.

[22] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. SMCQL: secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017.

[23] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: efficient SQL query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3):307–320, 2018.

[24] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment*, 13(12):2691–2705, 2020.

[25] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography*, volume 8975 of *Lecture Notes in Computer Science*, pages 227–234. Springer, 2015.

[26] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2016(3):117–135, 2016.

[27] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[28] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[29] Jonas Böhler and Florian Kerschbaum. Secure sublinear time differentially private median computation. In *NDSS*. The Internet Society, 2020.

[30] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM Conference on Computer and Communications Security*, pages 1175–1191. ACM.

[31] Christoph Bösch, Pieter H. Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys*, 47(2):18:1–18:51, 2014.

[32] Boston Women's Workforce Council (BWWC). Gender/racial pay gap in boston by the numbers. https://thebwwc.org, 2021.

[33] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[34] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Report 2020/599, 2020. https://eprint.iacr.org/2020/599.

[35] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 991–1008. USENIX Association, 2018.

[36] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[37] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 668–679, New York, NY, USA, 2015. Association for Computing Machinery.

[38] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*. The Internet Society, 2014.

[39] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: high throughput 3pc over rings with application to secure prediction. In *CCSW@CCS*, pages 81–92. ACM, 2019.

[40] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS*. The Internet Society, 2020.

[41] Surajit Chaudhuri. An overview of query optimization in relational systems. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press, 1998.

[42] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 354–366. Morgan Kaufmann, 1994.

[43] Albert Cheu, Adam D. Smith, Jonathan R. Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *EUROCRYPT (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 375–403. Springer, 2019.

[44] Sherman S. M. Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*. The Internet Society, 2009.

[45] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 259–282, Boston, Massachusetts, USA, 2017. USENIX Association.

[46] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, pages 2183–2200. USENIX Association, 2021.

[47] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *Financial Cryptography*, volume 9603 of *Lecture Notes in Computer Science*, pages 169–187. Springer, 2016.

[48] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[49] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *EuroSys*, pages 39:1–39:17. ACM, 2020.

[50] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[51] F. Emekci, D. Agrawal, A. E. Abbadi, and A. Gulbeden. Privacy preserving query processing using third parties. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 27–27, 2006.

[52] Saba Eskandarian and Matei Zaharia. Oblidb: oblivious query processing for secure databases. *Proceedings of the VLDB Endowment*, 13(2):169–183, 2019.

[53] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. CostCO: An automatic cost modeling framework for secure multi-party computation. In *IEEE EuroS&P 22*, 2022.

[54] Xin Fang, Stratis Ioannidis, and Miriam Leeser. SIFO: secure computational infrastructure using FPGA overlays. *Int. J. Reconfigurable Comput.*, 2019:1439763:1–1439763:18, 2019.

[55] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the GPU. In *SCN*, volume 8642 of *Lecture Notes in Computer Science*, pages 358–379. Springer, 2014.

[56] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 172–191. IEEE Computer Society, 2017.

[57] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[58] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In Pierangela Samarati, Mohammad S. Obaidat, and Enrique Cabello, editors, *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017*, pages 200–211. SciTePress, 2017.

[59] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331. ACM, 2018.

[61] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1353–1364, New York, NY, USA, 2016. Association for Computing Machinery.

[62] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. In *IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE, 2019.

[63] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *ACM Conference on Computer and Communications Security*, pages 1389–1406. ACM, 2017.

[64] Zhian He, Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, Siu Ming Yiu, and Eric Lo. Sdb: A secure query processing system with data interoperability. *Proceedings of the VLDB Endowment*, 8(12):1876–1879, 2015.

[65] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. *IACR Cryptology ePrint Archive*, 2019:723, 2019.

[66] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. *IACR Cryptol. ePrint Arch.*, 2011:122, 2011.

[67] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 149–180, Cham, 2018. Springer International Publishing.

[68] Randy Howard Katz and Gaetano Borriello. *Contemporary logic design (2. ed.)*. Pearson Education, 2005.

[69] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1329–1340, New York, NY, USA, 2016. Association for Computing Machinery.

[70] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[71] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. In *30th USENIX Security Symposium*. USENIX Association, 2021.

[72] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proc. VLDB Endow.*, 13(11):2132–2145, 2020.

[73] Sam Kumar, David E. Culler, and Raluca Ada Popa. MAGE: Nearly zero-cost virtual memory for secure computation. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 367–385. USENIX Association, July 2021.

[74] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 487–504. USENIX Association, 2020.

[75] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, August 2017. USENIX Association.

[76] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[77] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, December 2020.

[78] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, may 2014.

[79] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522, 2020.

[80] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 279–296. IEEE Computer Society, 2018.

[81] Payman Mohassel and Peter Rindal. Aby^3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

[82] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and PSI for secret shared data. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1271–1287. ACM, 2020.

[83] Arjun Narayan and Andreas Haeberlen. Djoin: Differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 149–162, October 2012.

[84] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[85] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 334–348. IEEE Computer Society, 2013.

[86] Takashi Nishide and Kazuo Ohta. Constant-round multiparty computation for interval test, equality test, and comparison. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 90-A(5):960–968, 2007.

[87] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. Dstress: Efficient differentially private computations on distributed data. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 560–574, 2017.

[88] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374, 2014.

[89] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. pages 2165–2182, 2021.

[90] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *NDSS*. The Internet Society, 2020.

[91] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium*, pages 515–530. USENIX Association, 2015.

[92] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153. Springer, 2019.

[93] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 125–157. Springer, 2018.

[94] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endow.*, 12(11):1664–1678, July 2019.

[95] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. Senate: A maliciously-secure MPC platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association.

[96] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2011.

[97] Anjana Rajan, Lucy Qin, David W. Archer, Dan Boneh, Tancrède Lepoint, and Mayank Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *COMPASS*, pages 49:1–49:4. ACM, 2018.

[98] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 2nd edition, 2000.

[99] Peter Rindal. https://github.com/ladnir/aby3, The ABY3 Framework for Machine Learning and Database Operations. [Online; accessed April 2022].

[100] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[101] Riivo Talviste. *Practical applications of secure multiparty computation*, chapter 12, pages 246–251. IOS Press, 2015.

[102] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2021.

[103] Transaction Processing Performance Council. TPC Benchmark H. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf. [Online; accessed April 2022].

[104] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *Proc. Priv. Enhancing Technol.*, 2019(3):370–388, 2019.

[105] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 3:1–3:18. ACM, 2019.

[106] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 299–313, USA, 2017. USENIX Association.

[107] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.

[108] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[109] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, pages 21–37. ACM, 2017.

[110] Yilei Wang and Ke Yi. *Secure Yannakakis: Join-Aggregate Queries over Private Data*, pages 1969–1981. Association for Computing Machinery, New York, NY, USA, 2021.

[111] Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, and Siu Ming Yiu. Secure query processing with data interoperability in a cloud database environment. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1395–1406, 2014.

[112] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, USA, 2015. IEEE Computer Society.

[113] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 89–100. IEEE Computer Society, 1994.

[114] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, USA, 1986. IEEE Computer Society.

[115] Zheguang Zhao, Seny Kamara, Tarik Moataz, and Zdonik Stan. Encrypted databases: From theory to systems. In *Proceedings of the 11th Annual Conference on Innovative Data Systems Research*, 2021.

[116] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, 2017.

[117] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. Cerebro: A platform for multi-party cryptographic collaborative learning. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2723–2740. USENIX Association, 2021.

A ANALYTICAL COST MODEL

A.1 Oblivious SECRECY primitives

Boolean operations. As explained in §3.2, we consider that each single-bit operation on shares has unit cost, so the operation cost of both XOR and AND operations is $C_o(\texttt{XOR}) = C_o(\texttt{AND}) = \ell$, where ℓ is the length of the share representation in bits. Recall that the synchronization cost of XOR is $C_s(\texttt{XOR}) = 0$ whereas the synchronization cost of AND is $C_s(\texttt{AND}) = 1$ round of communication. We further explain these costs below.

In SECRECY, each party starts with two shares of the input secrets s, t and ends up with two shares of the output. Initially, P_1 has s_1, s_2, t_1, t_2 whereas P_2 has s_2, s_3, t_2, t_3, and P_3 has s_3, s_1, t_3, t_1. Observe that $s \oplus t = (s_1 \oplus s_2 \oplus s_3) \oplus (t_1 \oplus t_2 \oplus t_3) = (s_1 \oplus t_1) \oplus (s_2 \oplus t_2) \oplus (s_3 \oplus t_3)$. Each parenthesis corresponds to a share of $s \oplus t$ and each party can compute two out of the three shares by simply XORing its input shares.

Logical AND is a bit more complex. Observe that $st = (s_1 \oplus s_2 \oplus s_3) \wedge (t_1 \oplus t_2 \oplus t_3)$. After distributing the AND over the XOR and doing some rearrangement we have $st = (s_1t_1 \oplus s_1t_2 \oplus s_2t_1) \oplus (s_2t_2 \oplus s_2t_3 \oplus s_3t_2) \oplus (s_3t_3 \oplus s_3t_1 \oplus s_1t_3)$. Again, each parenthesis corresponds to a share of st. Using its input shares, each party can locally compute one of these shares. The parties then XOR this share with a fresh sharing of the number 0 (which is created locally) so that the final share is uniformly distributed [17]. In the end, each party sends the computed share to its successor on the ring (clockwise) so that all parties end up with two shares of st. Logical OR and NOT are based on the XOR and AND primitives.

Equality/Inequality. Using these boolean operations, parties can jointly compute $s \stackrel{?}{=} t$ (resp. $s \stackrel{?}{<} t$) by computing a sharing of $s \oplus t$ and then taking the oblivious boolean-AND of each of the bits of this string (resp., taking the value of s_i at the first bit i in which the two strings differ). As a result, taking the equality of ℓ-bit strings requires $C_o(\text{eq}) = 2\ell - 1$ operations (namely, ℓ XORs plus $\ell - 1$ ANDs) and $C_s(\text{eq}) = \lceil \log \ell \rceil$ rounds. Similarly, inequality comparison has $C_o(\text{ineq}) = 4\ell - 3$ and $C_s(\text{ineq}) = \lceil \log(\ell + 1) \rceil$. As special cases, $s < 0$ requires no communication, and equality with a public constant $s{=}c$ can also be done locally provided that the data owners have secret-shared the results of $s - c$ and $c - s$ [70].

Compare-and-swap. The parties can calculate the min and max of two strings. Setting $b = (s{<}t)$, we can use a multiplexer to compute $s' = \min\{s, t\} = bs \oplus (1 \oplus b)t$ and $t' = \max\{s, t\} = (1 \oplus b)s \oplus bt$. Evaluating these formulas requires 6 more operations and 1 more synchronization round beyond the cost of the oblivious inequality.

Sort and shuffle. Given an array of n secret-shared strings, each of length ℓ, oblivious sort in SECRECY is based on a bitonic sorter that comprises $\log n \cdot (\log n + 1)/2$ stages and performs $n/2$ independent compare-and-swap operators in each stage. Hence, sorting has operational cost $C_o(\text{sort}_n) = \frac{1}{4} n \log n \cdot (\log n + 1) \cdot (C_o(\text{ineq}) + 6)$ and synchronization cost $C_s(\text{sort}_n) = \frac{1}{2} \log n \cdot (\log n + 1) \cdot (C_s(\text{ineq}) + 1)$. We can obliviously shuffle values in a similar fashion: each party appends an attribute that is populated with locally generated random values, sorts the values on this attribute, and then discards it.

Boolean addition. Given boolean-shared integers s and t, computing the boolean share of $s + t$ using a ripple-carry adder [68] can be done with $C_o(\texttt{RCA}) = 5\ell - 3$ operations in $C_s(\texttt{RCA}) = \ell$ rounds.

Arithmetic operations. Arithmetic addition and multiplication work similarly to XOR and AND respectively (see above). Scalar multiplication $c \cdot u$, where c is a public constant, does not require communication.

Conversion. We can convert between additive and boolean sharings [50, 81, 89] by securely computing all of the XOR and AND gates in a ripple-carry adder. Single-bit conversion can be done in two rounds with the simple protocol that is also used in CrypTen [70].

A.2 Oblivious SECRECY operators

Let R, S, and T be relations with cardinalities $|R|$, $|S|$, and $|T|$ respectively. Let also $t[a_i]$ be the value of attribute a_i in tuple t. To simplify the presentation, we describe each operator based on the logical (i.e., secret) relations and not the random shares distributed across parties. That is, when we say that "*an operator is applied to a relation R and defines another relation T*", in practice this means that each party begins with shares of R, performs some MPC operations on the shares, and ends up with shares of T.

PROJECT. Oblivious projection has the same semantics as its plaintext counterpart. The operation and synchronization costs of oblivious `PROJECT` are both zero since each party can locally disregard the shares corresponding to the filtered attributes.

SELECT. An oblivious selection with predicate φ on a relation R defines a new relation:

$$T = \{t \cup \{\varphi(t)\} \mid t \in R\}$$

with the same cardinality as R, i.e. $|T| = |R|$, and one more single-bit attribute for each tuple $t \in R$ that contains ϕ's result when applied to t. This bit denotes whether t is included in the output relation T and is securely computed under MPC so that its true value remains hidden (i.e., secret-shared) from the computing parties. Note that, in contrast to a typical selection in the clear, oblivious selection defines a relation with the same cardinality as the input, i.e., it does not remove tuples from R so that the true size of T is kept secret.

Costs: The operation cost of `SELECT` is $C_o(\sigma_\phi(R)) = C_o(\phi(t)) \cdot |R|, \quad t \in R$, where $C_o(\phi(t))$ is the operation cost

of evaluating ϕ on a single tuple $t \in R$. Since predicate evaluation can be performed independently for each tuple in R, the total number of rounds to perform the `SELECT` equals the number of rounds to evaluate the selection predicate on a single tuple, i.e., $C_s(\sigma_\phi(R)) = C_s(\phi(t)),\ t \in R$.

Both $C_o(\phi(t))$ and $C_s(\phi(t))$ are independent of the actual t contents: they only depend on ϕ's syntax and the lengths of the attributes used in ϕ. In SECRECY, a predicate ϕ can be an arbitrary logical expression with atoms that may also include arithmetic expressions $(+, \times, =, >, <, \neq, \geq, \leq)$ and is constructed using the primitives of §A.1. Consider the example predicate $\phi :=$ `age>30 AND age<40` that requires `AND`ing the results of two oblivious inequalities under MPC. Based on the costs of primitive operations, we have: $C_o(\phi(t)) = 2C_o(\text{ineq}) + C_o(\texttt{AND})$ and $C_s(\phi(t)) = C_s(\text{ineq}) + 1$. In §5.3, we described a technique we use in SECRECY that can reduce selections to local operations (with $C_s = 0$).

JOIN. An oblivious θ-join between two relations R and S, denoted with $R \bowtie_\theta S$, defines a new relation:

$$T = \{(t \cup t' \cup \{\theta(t,t')\}) \mid t \in R \wedge t' \in S\}$$

where $t \cup t'$ is a new tuple that contains all attributes of $t \in R$ along with all attributes of $t' \in S$, and $\theta(t,t')$ is θ's result when applied to the pair of tuples (t,t'). T is the cartesian product of the input relations $(R \times S)$, where each tuple is augmented with a (secret-shared) bit denoting whether the tuple t "matches" with tuple t' according to θ. We emphasize that our focus in this work is on general-purpose oblivious joins that can support arbitrary predicates; there also exist special cases of oblivious join algorithms, e.g., primary- and foreign-key equi-joins with lower asymptotic complexity [15, 72, 82, 93] or compositions of equi-joins with specific operators [34] that could be added to SECRECY if desired.

Costs: The general oblivious `JOIN` requires a nested-loop over the input relations to check all possible pairs, so its operation cost is $C_o(R \bowtie_\theta S) = C_o(\theta(t,t')) \cdot |R| \cdot |S|,\ t \in R, t' \in S$. However, the total number of communication rounds to evaluate the `JOIN` is independent of the input cardinality; it only depends on the join predicate θ, i.e., $C_s(R \bowtie_\theta S) = C_s(\theta(t,t')), t \in R, t' \in S$. For example, a range join $R \bowtie_{a<b} S$ has $C_o(R \bowtie_{a<b} S) = 2|R| \cdot |S| \cdot C_o(\text{ineq})$ and $C_s(R \bowtie_{a<b} S) = C_s(\text{ineq})$. The constant asymptotic complexity in number of rounds with respect to the input cardinality holds for *any* θ-join. Join predicates in SECRECY can be arbitrary expressions whose cost is computed as explained above for selection predicates.

SEMI-JOIN. An oblivious (left) semi-join between two relations R and S on a predicate θ, denoted with $R \ltimes_\theta S$, defines a new relation:

$$T = \{(t \cup \{\bigvee_{\forall t' \in S} \theta(t,t')\}) \mid t \in R\}$$

with the same cardinality as R, i.e. $|T| = |R|$, and one more attribute that stores the result of the formula $f(\theta,t,S) = \bigvee_{\forall t' \in S} \theta(t,t'),\ t \in R$ indicating whether the tuple in R "matches" any tuple in S.

Costs: The operation cost of the general oblivious `SEMI-JOIN` is $C_o(R \ltimes_\theta S) = C_o(f(\theta,t,S)) \cdot |R| = C_o(\theta(t,t')) \cdot |R| \cdot |S| + |R| \cdot (|S| - 1),\ t \in R, t' \in S$. The formula $f(\theta,t,S)$ can be evaluated independently for each tuple $t \in R$ using a binary tree of `OR` operations, therefore, the synchronization cost of the semi-join is $C_s(R \ltimes_\theta S) = C_s(\theta(t,t')) + \lceil \log |S| \rceil, t \in R, t' \in S$ (i.e., independent of $|R|$).

ORDER-BY. Oblivious order-by on attribute a_k has the same semantics as the non-oblivious operator. Hereafter, sorting a relation R with m attributes on ascending (resp. descending) order of an attribute $a_k, 1 \leq k \leq m$, is denoted as $s_{\uparrow a_k}(R) = T$ (resp. $s_{\downarrow a_k}(R) = T$). We define order-by on multiple attributes using the standard semantics. For example, sorting a relation R first on attribute a_k (ascending) and then on a_n (descending) is denoted as $s_{\uparrow a_k \downarrow a_n}(R)$. An order-by operator is often followed by a `LIMIT` that defines the number of tuples the operator must output.

Costs: Oblivious `ORDER-BY` in SECRECY relies on a bitonic sorter of §A.1 that internally uses an oblivious multiplexer. Hence, the operation and synchronization costs are $C_o(s_{\uparrow a}(R)) = C_o(\text{sort}_{|R|})$ and $C_s(s_{\uparrow a}(R)) = C_s(\text{sort}_{|R|})$, as given in §A.1. In this case, the number of operations required by each oblivious multiplexing is linear to the number of attributes in the input relation, however, the total number of rounds depends only on the cardinality of the input. The analysis assumes one sorting attribute; adding more sorting attributes increases the number of operations and communication rounds in each comparison by a small constant factor.

GROUP-BY with aggregation. An oblivious group-by aggregation on a relation R with m attributes defines a new relation $T = \{f(t') \mid t' = t \cup \{a_g, a_v\},\ t \in R\}$ with the same cardinality as R, i.e. $|T| = |R|$, and two more attributes: a_g that stores the result of the aggregation, and a_v that denotes whether the tuple t is 'valid', i.e., included in the output. Let a_k be the group-by key and a_w the attribute whose values are aggregated. Let also $S = \left[t_1[a_w], t_2[a_w], ..., t_u[a_w]\right]$ be the list of values for attribute a_w for all tuples $t_1, t_2, ..., t_u \in R$ that belong to the same group, i.e., $t_1[a_k] = t_2[a_k] = ... = t_u[a_k]$, $1 \leq u \leq |R|$. The function f in T's definition above is defined as follows:

$$f(t_i) = \begin{cases} t_i[a_g] = agg(S), t_i[a_v] = 1, \ i = u', \ 1 \leq u' \leq u \\ t_{inv}, \ i \neq u', \ 1 \leq i \leq u \end{cases}$$

where t_{inv} is a tuple with $t_{inv}[a_v] = 0$ and the rest of the attributes set to a special reserved value, while $agg(S)$ is the aggregation function, e.g. `MIN`, `MAX`, `COUNT`, `SUM`, `AVG`, and is implemented using the primitives of §A.1. Put simply, oblivious aggregation sets the value of a_g for one tuple per group equal to the result of the aggregation for that group and updates (in-place) all other tuples with "garbage." This operation

is followed by an oblivious shuffling to hide the group boundaries when opening the relations to the learner (and only if there is no subsequent shuffling in the query plan). Groups can be defined on multiple attributes using the standard semantics.
Costs: The GROUP-BY operator $\gamma_{a_k}^{agg}(R)$ breaks into two phases: an oblivious sort on the group-by key(s) and an odd-even aggregation [66] applied to the sorted input. The odd-even aggregation performs $(|R|(\log|R|-1)+1)\cdot C_o(agg(t,t'))$ operations in $\log|R|\cdot C_s(agg(t,t'))$ rounds, where $C_o(agg(t,t'))$ and $C_s(agg(t,t'))$ are the operation and synchronization costs, respectively, of applying the aggregation function to a single pair of tuples $t,t' \in R$ (independent of $|R|$). Accounting for the initial sorting on the group-by keys, the total operation cost of the oblivious group-by is $C_o(\gamma_{a_k}^{agg}(R)) = C_o(s_{\uparrow a_k}(R)) + (|R|(\log|R|-1)+1)\cdot C_o(agg(t,t'))$. The total synchronization cost is $C_s(\gamma_{a_k}^{agg}(R)) = C_s(s_{\uparrow a_k}(R)) + \log|R|\cdot C_s(agg(t,t'))$. The analysis can be easily extended to multiple group-by keys.

DISTINCT. The oblivious distinct operator is a special case of group-by with aggregation, assuming that a_k is not the group-by key as before but the attribute where distinct is applied. For distinct, there is no a_g attribute and the function f is defined as follows:

$$f(t_i) = \begin{cases} t_i[a_v] = 1, \;\; i = u', \;\; 1 \le u' \le u \\ \\ t_i[a_v] = 0, \;\; i \neq u', \;\; 1 \le i \le u \end{cases}$$

Distinct marks one tuple per group as 'valid' and the rest as 'invalid'.

Costs: The DISTINCT operator includes an oblivious sort on the distinct attribute(s) followed by a second phase where the operator compares adjacent tuples in the sorted input to set the distinct bit a_v. Setting the distinct bit for each tuple is independent from the rest of the tuples, so all distinct bit operations can be performed in bulk. The total operation cost $C_o(\delta a_k(R)) = C_o(s_{\uparrow a_k}(R)) + (|R|-1)\cdot C_o(\text{eq})$ and synchronization cost $C_s(\delta a_k(R)) = C_s(s_{\uparrow a_k}(R)) + C_o(\text{eq})$ of oblivious distinct are dominated by the oblivious sort.

MASK. Let t_{inv} be a tuple with all attributes set to a special reserved value. A mask operator with predicate p on a relation R defines a new relation $T = \{f(t) \mid t \in R\}$, where:

$$f(t) = \begin{cases} t, \;\; p(t) = 0 \\ \\ t_{inv}, \;\; p(t) = 1 \end{cases}$$

Mask is used at the end of the query, just before opening the result to the learner, and only if there is no previous masking. The cost analysis of MASK is similar to that of SELECT.

Global aggregations. SECRECY also supports global aggregations without a group-by clause. The total operation cost of a global aggregation is $C_o(agg(R)) = C_o(agg(t,t'))\cdot(|R|-1)$, where $C_o(agg(t,t'))$ is the operation cost of applying the aggregation function to a single pair of tuples $t,t' \in R$. The total synchronization cost is $C_s(agg(R)) = C_s(agg(t,t'))\cdot\lceil\log|R|\rceil$, since the aggregation can be applied using a binary tree of function evaluations.

A.3 Composition of oblivious operators

Composing selections and joins. Recall that selections, joins, and semi-joins append a single-bit attribute to their input relation that indicates whether the tuple is included in the output. To compose a pair of such operators, we compute both single-bit attributes and take their conjunction under MPC. For example, for two selection operators σ_1 and σ_2 with predicates φ_1, φ_2, the composition $\sigma_2(\sigma_1(R))$ defines a new relation $T = \{t \cup \{e_c = \varphi_1(t) \wedge \varphi_2(t)\} \mid t \in R\}$. The cost of composition in this case is the cost of evaluating the expression $\varphi_1(t) \wedge \varphi_2(t)$ for each tuple in T. This includes $|T|$ independent boolean ANDs which can be evaluated in one round.

Composing distinct with other operators. Applying a selection or a (semi-)join to the result of DISTINCT requires one communication round to compute the conjunction of the selection or (semi-) join bit with the bit a_v generated by distinct. However, applying DISTINCT to the output of a selection, a (semi-)join or a group-by operator, requires some care. Consider the case where DISTINCT is applied to the output of a selection. Let a_ϕ be the attribute added by the selection and a_k be the distinct attribute. To set the distinct bit a_v at each tuple, we must make sure there are no other tuples with the same attribute a_k, with $a_\phi = 1$, and whose distinct bit a_v is already set. To do so, the distinct operator must process tuples *sequentially* and the composition itself requires n rounds, where n is the cardinality of the input. This results in a significant increase over the $O(\log^2 n)$ rounds required by distinct when applied to a base relation. Applying distinct to the output of a group-by or (semi-)join incurs a linear number of rounds for the same reason. In §5.2, we proposed an optimization that reduces the cost of these compositions to a logarithmic factor.

Composing group-by with other operators. To perform a group-by on the result of a selection or (semi-)join, the group-by operator must apply the aggregation function to all tuples in the same group that are also included in the output of the previous operator. Consider the case of applying group-by to a selection result. To identify the aforementioned tuples, we need to evaluate the formula:

$$b \leftarrow b \wedge t_i[a_\phi] \wedge t_j[a_\phi]$$

at each step of the group-by operator, where b is the bit that denotes whether the tuples t_i and t_j belong to the same group and a_ϕ is the selection bit. This formula includes two boolean ANDs that require two communication rounds. Applying

group-by to the output of a (semi-)join has the same composition cost; in this case, we replace a_ϕ in the above formula with the (semi-)join attribute a_θ.

To apply a selection to the result of GROUP-BY, we must compute a boolean AND between the selection bit a_ϕ and the 'valid' bit a_v of each tuple generated by the group-by. The cost of composition in number of rounds is independent of the group-by result cardinality, as all boolean ANDs can be applied in bulk. The same holds when applying a (semi-)join to the output of group-by. Finally, composing two group-by operators has the same cost with applying GROUP-BY to the result of selection, as described above.

Composing order-by with other operators. Composing ORDER- BY with other operators is straight-forward. Applying an operator to the output of order-by has zero composition cost. The converse operation, applying ORDER-BY to the output of an operator, requires a few more boolean operations per oblivious compare-and-swap (due to the attribute/s appended by the previous operator), but does not incur additional communication rounds.

B SECURITY ANALYSIS

We have purposely designed SECRECY in a modular *black-box* fashion, with a hierarchy of *MPC protocol functionalities* → *oblivious primitives* → *relational operators* → *optimizations*. This design choice provides two benefits: (i) immediate *inheritance* of all security guarantees provided by the underlying MPC protocol, and (ii) *flexibility* to support different protocols in the future that might have a different number of parties, threshold, and threat model.

Inheritance of security guarantees. SECRECY relies on a set of functionalities that must be provided by the MPC protocol. These functionalities enable parties to receive secret-shared inputs and return secret-shared outputs: (i) $\mathcal{F}_{\text{add}}$ and $\mathcal{F}_{\text{mult}}$ that add and multiply their inputs, (ii) $\mathcal{F}_{\text{xor}}$ and $\mathcal{F}_{\text{and}}$ that take boolean operations of their inputs, (iii) $\mathcal{F}_{\text{a2b}}$ and $\mathcal{F}_{\text{b2a}}$ that perform conversions between arithmetic and boolean representations, (iv) $\mathcal{F}_{\text{eq}}$ and $\mathcal{F}_{\text{cmp}}$ to compute the equality and comparison predicates (where the hardest step of the latter usually involves extracting the most significant bit of an arithmetic-shared value), and (v) $\mathcal{F}_{\text{sh}}$ and $\mathcal{F}_{\text{rec}}$ that allow external participants to secret-share data to and reconstruct data from the computing parties.

In this section, we argue that SECRECY retains the security guarantees provided by the underlying MPC protocol, or equivalently that it retains the security guarantees of these ideal functionalities. Our reasoning shows that SECRECY compiles each query into a sequence of calls to these functionalities that is *oblivious*, meaning that its control flow is independent of its input and all data remains hidden:

1. SECRECY calls the functionalities of the MPC protocol in a black-box manner. As a result, computing parties always operate on secret-shared data; only $\mathcal{F}_{\text{rec}}$ provides any data in the clear (namely to the learner), and SECRECY only calls this functionality once at the end of the query execution.

2. The control flow of each relational operator (§A.2) is oblivious, i.e., data-independent. Concretely, SELECT and PROJECT always require a single pass over the input, (semi-)JOINs require a nested for-loop over the two inputs, ORDER-BY is based on an oblivious sorting network, and GROUP-BY and DISTINCT consist of an ORDER-BY followed by an additional oblivious step (to apply the aggregation and identify the unique records, respectively).

3. SECRECY composes relational operators (§A.3) using the protocol functionalities (e.g., taking ANDs under MPC) within an oblivious linear scan over the output of the composition.

4. The logical and physical transformations of §5 rewrite the oblivious sequence of calls to the protocol functionalities into a new semantically equivalent sequence of calls that is also oblivious and has lower execution cost.

As a result, semi-honest security of the full SECRECY protocol follows by inspection of the ideal functionalities. Privacy is satisfied against all parties because none of the functionalities ever provides a (non-secret-shared) output to the data owners or computing parties, and only the final $\mathcal{F}_{\text{rec}}$ provides an output to the analyst as desired. Correctness of the full protocol follows immediately from correctness of each individual functionality.

Generality of optimizations. The logical and physical query optimizations constructed in this work (§5.1-5.2) apply generally to any mixed-mode MPC protocol that supports the set of functionalities we describe above. This level of abstraction is commonly used by modern mixed-mode MPC protocols (e.g., [17, 39, 40, 46, 50, 71, 81, 89, 90]).

If providing malicious security, we require these functionalities to validate the shares of their inputs and outputs (e.g., using an information-theoretic MAC or replicated sharing), either immediately or with delayed validation before invoking $\mathcal{F}_{\text{rec}}$. As a consequence, SECRECY satisfies correctness against the computing parties because input validation binds them to provide the output of the prior step as the input shares into the next functionality. Additionally, correctness against the data owners and analyst follow from the fact that, aside from the data owners' initial sharing through $\mathcal{F}_{\text{sh}}$, none of the functionalities allow them to provide an input so they cannot influence the protocol execution.

As a result, the techniques from SECRECY can be applied to any N-party MPC protocol that provides semi-honest or malicious security against T adversarial parties. In particular, SECRECY can be instantiated with 2, 3, and 4-party secret

sharing-based protocols that remain secure in the face of a malicious adversary who can deviate from the protocol arbitrarily (e.g., [46, 71, 89]), or with (authenticated) garbled circuit protocols [109, 114] combined with occasional conversions to arithmetic secret sharing [50, 89] as needed. Protocols that provide the stronger cryptographic guarantee of robustness often do so by running several MPC executions both before and after evicting the malicious party, and by the same logic as above SECRECY even maintains the robust security of these protocols.

C SUMMARY OF MPC-BASED SYSTEMS FOR RELATIONAL ANALYTICS

Here we provide more details on existing systems for relational MPC summarized in Table 5. Hybrid execution splits the query plan into parts that can be evaluated in plaintext by a single party (the appropriate data owner) versus parts that inherently require multiple parties' data (executed under MPC). Therefore, hybrid execution is only feasible when data owners can compute part of the query on premise. SMCQL, SDB, and Conclave can further sidestep MPC when some attributes have been annotated as non-sensitive. Shrinkwrap and SAQE build on SMCQL to calibrate leakage based on user-provided privacy budgets, and Senate reduces joint computation when some relations are owned by subsets of the computing parties. This is common in peer-to-peer MPC but does not occur in a typical outsourced setting like the one of Figure 1, where all computing parties receive shares of the input data.

As shown in Table 5, Senate is the only relational system with support for malicious security. Due to its hybrid execution model, Senate requires additional steps to verify the integrity of local computations by the data owners (not to be confused with formal software verification). While SECRECY currently focuses on semi-honest security, the Araki et al. protocol [17] and subsequent mixed-mode ABY^3 protocols [81] can be extended to provide malicious security with low computational cost [16, 71]. By optimizing MPC rather than sidestepping it, a malicious-secure version of SECRECY would not impose any new restriction on the supported set of queries or the mixed-mode MPC protocol utilized.

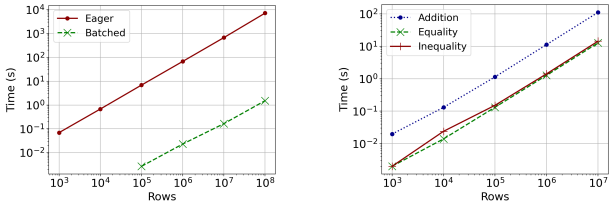
D ADDITIONAL EXPERIMENTS

D.1 Performance of SECRECY primitives

Here we present a set of micro-benchmarks that evaluate the performance of SECRECY's MPC primitives in `AWS-LAN`.

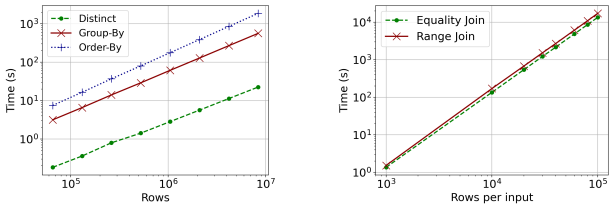
Effect of message batching on communication latency. In the first experiment, we measure the latency of inter-party communication using two messaging strategies. Recall that, during a message exchange, each party sends one message to its successor and receives one message from its predecessor on the logical 'ring'. *Eager* exchanges data among parties as soon as they are generated, thus,. producing a large number of small messages. The *Batched* strategy, on the other hand, collects data into batches and exchanges them only when computation cannot otherwise make progress, thus, producing as few as possible, albeit large messages.

We run this experiment with increasing data sizes and measure the total time from initiating the exchange until all parties complete the exchange. Figure 7a shows the results. We see that batching provides two to four orders of magnitude lower latency than eager messaging. Using batching in our experimental setup, parties can exchange $100M$ 64-bit data shares in $2s$. These results reflect the network performance in our cloud testbed. We expect better performance in dedicated clusters with high-speed networks and higher latencies if the computing parties communicate over the internet.



(a) Effect of message batching (b) Comparison and addition

Figure 7: Performance of oblivious SECRECY primitives



(a) Unary operators (b) Join operators

Figure 8: Performance of oblivious SECRECY operators

Performance of secure computation primitives. We now evaluate the performance of oblivious primitives that require communication among parties. These include equality, inequality, and addition with the ripple-carry adder. In Figure 7b we show the execution time of oblivious primitives as we increase the input size from $1K$ rows to $10M$ rows. All primitives scale well with the input size as they all depend on a constant number of communication rounds. Equality requires six rounds. Inequality requires seven rounds and more memory than equality. Boolean addition is not as computation-intensive as inequality, but requires a higher number of rounds (64).

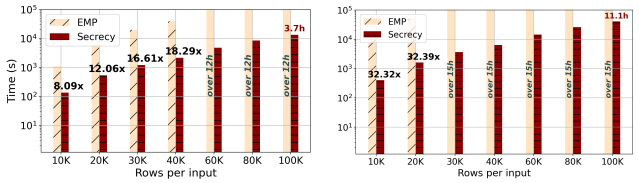


Figure 9: Performance comparison of an oblivious join operator on EMP and SECRECY in LAN (left) and WAN (right).

D.2 Performance of SECRECY operators

The next set of experiments evaluates the performance of SECRECY's relational operators. We apply `DISTINCT`, `GROUP-BY`, `ORDER-BY`, and `JOIN` (equality and range) to relations of increasing size and measure the total execution time per operator in `AWS-LAN`. We empirically verify the cost analysis of §A and show that our vectorized implementations are efficient and scale to millions of input rows with a single CPU thread. Figure 8 shows the results.

Unary operators. In Figure 8a, we plot the execution time of unary operators vs the input size. Recall from §A.2 that `DISTINCT` and `GROUP-BY` are both based on sorting and, thus, their cost includes the cost of `ORDER-BY` for unsorted inputs of the same cardinality. To shed more light on the performance of `DISTINCT` and `GROUP-BY`, Figure 8a only shows the execution time of their second phase, that is, after the input is sorted and, for `GROUP-BY`, before the final shuffling (which has identical performance to sorting).

For an input relation with n rows, `DISTINCT` performs $n-1$ equality comparisons, one for each pair of adjacent rows. Since all these comparisons are independent, our vectorized implementation uses batching, thus, applying `DISTINCT` to the entire input in six rounds of communication (the number of rounds required for oblivious equality on pairs of 64-bit shares). As a result, `DISTINCT` scales well with the input size and can process $10M$ rows in $20s$. `GROUP BY` is slower than `DISTINCT`, as it requires significantly more rounds of communication, logarithmic to the input size. Finally, `ORDER BY` relies on our implementation of bitonic sort, where all $\frac{n}{2}$ comparisons at each level are batched within the same round.

Joins. The oblivious join operators in SECRECY hide the size of their output, thus, they compute the cartesian product between the two input relations and produce a bit share for all pairs of records, resulting in an output with $n \cdot m$ entries. We run both operators with $n = m$, for increasing input sizes, and plot the results in Figure 8b. The figure includes equi-join and range-join results for up to $100K$ rows per input, as we capped the duration of this experiment to $5h$. SECRECY executes joins in batches without materializing their entire output at once. As a result, it can perform $10B$ equality and inequality comparisons under MPC within the experiment duration limit.

D.3 EMP vs SECRECY on oblivious join

Here we compare EMP with SECRECY using an oblivious join operator that is based on the sample program from the SoK project [10]. For these experiments, we use inputs of the same cardinality and increase the size from $10K$ to $100K$ rows per input. We cap the time of these experiments to $12h$. Fig. 9 plots the results in `AWS-LAN` and `AWS-WAN`. Within the experiment duration, EMP can evaluate joins on up to $40K$ rows per input (in $11h$). SECRECY is 18× faster for the same input size and can process up to $100K$ rows per input in less than $4h$.

E QUERIES USED IN THE PAPER

Here we list the queries used in §7 (in SQL syntax):

Comorbidity:
```
SELECT diag, COUNT(*) cnt
FROM diagnosis
WHERE pid IN cdiff_cohort
GROUP BY diag
ORDER BY cnt DESC
LIMIT 10
```

Recurrent C. Diff.:
```
WITH rcd AS (
   SELECT pid, time, row_no
   FROM diagnosis
   WHERE diag=cdiff)
 SELECT DISTINCT pid
 FROM rcd r1 JOIN rcd r2 ON r1.pid = r2.pid
 WHERE r2.time - r1.time >= 15 DAYS
 AND r2.time - r1.time <= 56 DAYS
 AND r2.row_no = r1.row_no + 1
```

Aspririn Count:
```
SELECT count(DISTINCT pid)
FROM diagnosis as d, medication as m on
d.pid = m.pid
WHERE d.diag = hd AND m.med = aspirin
AND d.time <= m.time
```

Password Reuse:
```
SELECT ID
FROM R
GROUP BY ID, PWD
HAVING COUNT(*)>1
```

Credit Score:
```
SELECT S.ID
FROM (
 SELECT ID, MIN(CS) as cs1, MAX(CS) as cs2
 FROM R
 WHERE R.year=2019
 GROUP-BY ID ) as S
WHERE S.cs2 - S.cs1 > c
```

TPC-H Q4:

```
SELECT o_orderpriority, count(*) as order_count
FROM orders
WHERE o_orderdate >= date '[DATE]' AND
 o_orderdate < date '[DATE]' + interval '3' month
 AND EXISTS (
      SELECT *
      FROM lineitem
      WHERE l_orderkey = o_orderkey
      AND l_commitdate < l_receiptdate
     )
GROUP BY o_orderpriority
ORDER BY o_orderpriority
```

TPC-H Q6:

```
SELECT sum(l_extendedprice*l_discount) as revenue
FROM lineitem
WHERE l_shipdate >= date '[DATE]' AND
 l_shipdate < date '[DATE]' + interval '1' year
 AND l_discount between [DISCOUNT] - 0.01
 AND [DISCOUNT] + 0.01 and l_quantity < [QUANTITY]
```

TPC-H Q13:

```
SELECT c_count, count(*) as custdist
FROM (
      SELECT c_custkey, count(o_orderkey)
      FROM customer left outer join orders ON
        c_custkey = o_custkey
        AND o_comment = `[WORD]'
      GROUP BY c_custkey
      ) as c_orders (c_custkey, c_count)
GROUP BY c_count
ORDER BY custdist desc, c_count desc
```

FLASH: Towards a High-performance Hardware Acceleration Architecture for Cross-silo Federated Learning

Junxue Zhang[1,2], Xiaodian Cheng[1,2], Wei Wang[2], Liu Yang[1,2], Jinbin Hu[1], Kai Chen[1]
[1]*iSINGLab @ Hong Kong University of Science and Technology* [2]*Clustar*

Abstract

Cross-silo federated learning (FL) adopts various cryptographic operations to preserve data privacy, which introduces significant performance overhead. In this paper, we identify nine widely-used cryptographic operations and design an efficient hardware architecture to accelerate them. However, directly offloading them on hardware statically leads to (1) inadequate hardware acceleration due to the limited resources allocated to each operation; (2) insufficient resource utilization, since different operations are used at different times. To address these challenges, we propose FLASH, a high-performance hardware acceleration architecture for cross-silo FL systems. At its heart, FLASH extracts two basic operators—modular exponentiation and multiplication—behind the nine cryptographic operations and implements them as highly-performant engines to achieve adequate acceleration. Furthermore, it leverages a dataflow scheduling scheme to dynamically compose different cryptographic operations based on these basic engines to obtain sufficient resource utilization. We have implemented a fully-functional FLASH prototype with Xilinx VU13P FPGA and integrated it with FATE, the most widely-adopted cross-silo FL framework. Experimental results show that, for the nine cryptographic operations, FLASH achieves up to 14.0× and 3.4× acceleration over CPU and GPU, translating to up to 6.8× and 2.0× speedup for realistic FL applications, respectively. We finally evaluate the FLASH design as an ASIC, and it achieves 23.6× performance improvement upon the FPGA prototype.

1 Introduction

Training a high-quality machine learning model requires massive data, which is likely to be distributed across different institutions or companies in the real world. However, the increasing concern about data privacy and emerging regulations and lawsuits restrict these data from being collected together in one place for centralized training. To solve this problem, federated learning (FL) has been proposed to enable distributed learning among these *data silos* by performing local computation within a data silo and securely aggregating the intermediate results (*e.g.*, gradients/parameters) to generate a global model without revealing any original data to the outside world [44, 48, 89].

To ensure the security of cross-silo FL, various cryptographic techniques have been used. For example, partially homomorphic encryptions (PHE), *e.g.*, Paillier, have been used to enable parameter computation/aggregation directly on ciphertexts [73]. RSA is used to build the blind signature-based Private Set Intersections (PSI) for sample alignment [45]. In this paper, we perform a comprehensive analysis of existing cross-silo FL applications and identify nine widely-used cryptographic operations, such as encryption/decryption, computation over ciphertexts, *etc*. (more details in §3.1). While preserving privacy, these cryptographic operations significantly degrade the performance (§3.2). For example, our experiments show that these operations cause up to 60.74× performance degradation. The reasons are two-fold: (1) they are of high computational complexity, *e.g.*, Paillier encryption has a $O(2^N)$[1] time complexity; (2) they introduce large number calculations, *e.g.*, additively HE and RSA encryption generate 2048-bit ciphertexts which need to be broken down to multiple 64-bit numbers and executed with limited parallelism on current CPU architecture.

In this paper, we ask: *can we offload these cryptographic operations to dedicated hardware to accelerate cross-silo FL?* Towards answering this question, our first attempt went with GPU. However, as we will reveal in §3.3, the cryptographic operations used in cross-silo FL involve complicated computation stages and dramatically inflate the data, making them inappropriate for GPUs. While GPU is ideal for performing data parallelism over tensors with short numbers, *e.g.*, single-precision floats, it fails to provide efficient pipeline execution for cryptographic operations with large numbers, *e.g.*, 2048-bit integers. The reason is that the limited size of the shared memory within one Streaming Multiprocessor (SM) causes frequent data exchange between external and on-chip shared memory during the pipeline execution, significantly compromising the performance. While it might be feasible to work around the limitation of GPUs, it requires complex mechanisms such as a complex memory orchestration system. Our paper does not take this direction (§6).

To further accelerate cross-silo FL, we seek a more efficient hardware acceleration architecture beyond the existing GPU architecture. To this end, we choose to use FPGA as a prototype and further explore an Application-specific Integrated Circuit (ASIC). We believe such customized hardware architecture will exhibit several desired properties for our purpose. First, it is possible to tailor a hardware architecture for efficient cross-silo FL by customizing the hardware circuits from scratch, so that we can design an optimized fine-grained pipelining with flexible bit-width support for accelerating cryptographic operations. Second, the customized hardware architecture allows us to provide sufficient on-chip memory for storing large numbers used in the processing pipeline for superior performance. However, while promising, we identify

[1] N is the bit-width of the exponent n, and n is the public key in Paillier encryption.

that directly offloading the nine cryptographic operations to the hardware statically will pose two key challenges (§3.3):

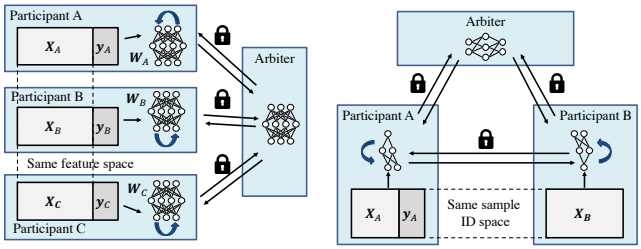
- **Inadequate hardware acceleration due to limited resources.** To achieve high performance, one operation may need multiple hardware instances for high parallelism. However, as the hardware resource of a chip is limited, directly offloading all these nice operations to the hardware causes inadequate resources to speed up each operation, leading to suboptimal performance. Our implementation with this approach on a Xilinx VU13P FPGA [23] chip shows that each operation only achieves $\sim$ 50% acceleration on average.
- **Insufficient resource utilization due to static offloading.** Different FL applications use different cryptographic operations, and within each application, different operations are used at different times. Consequently, statically offloading *all* the operations as a whole results in resource under-utilization because not all the operations are used at all times simultaneously.

To address the challenges, we take a closer look at these nine cryptographic operations and observe that almost all of them build upon two basic operators: *modular exponentiation and modular multiplication.* Based on this observation, we propose FLASH, a high-performance hardware acceleration architecture for cross-silo federated learning (§4). At its core, FLASH uses the majority of hardware resources to implement the two basic operators as high-performance engines to achieve adequate hardware acceleration. We also design fine-grained pipelines with sufficient on-chip memory to improve both the intra- and inter-engine execution efficiency for superior performance. Furthermore, based on these basic engines, FLASH adopts a dataflow scheduling module to dynamically compose these engines into different cryptographic operations on-demand to achieve high resource utilization.

We have provided a down-scale but full functional implementation of FLASH with Xilinx VU13P FPGA [23][2] for prototyping purpose, integrated it with FATE [2]—the most widely-adopted cross-silo FL framework—and evaluated it extensively with real-world FL applications. We compare the performance of FLASH with (1) the vanilla FATE, which uses GMP [19] to implement these cryptographic operations with CPU. GMP provides a highly-optimized implementation for modular multiplication and exponentiation operations, which uses many optimization algorithms, including but not limited to the two mentioned in our paper. We choose Intel Xeon Silver 4114 CPU similar to prior works [76]; (2) the FATE where the cryptographic operations are accelerated by NVIDIA P4[3]

[2]We use FPGA for prototyping purposes, so we do not consider the price advantages/disadvantages of the VU13P FPGA chip.

[3]We note that latest NVIDIA A100 [9]/H100 [10] may have a better performance than P4/VU13P, however, they are much more expensive while still sharing the architectural deficiency of GPU in general as discussed in §3.3. The focus of FLASH is to pursue a more efficient hardware acceleration architecture for cross-silo FL.



(a) Horizontal FL (b) Vertical FL

Figure 1: Two paradigms of cross-silo FL.

GPU [11, 43]. Here we use P4 GPU because it has the closest INT8 TOPS (although $\sim 2\times$ better) as FLASH (we typically use INT8 TOPS to denote the general computation power of a chip). P4 has $\sim$ 20 INT8 TOPS [11]. VU13P has $\sim$ 38.3 INT8 DSP TOPS while reaching peak 891MHz operation frequency [23, 24]. As FLASH uses 300MHz, it achieves $\sim$ 12.9 DSP INT8 TOPS. Moreover, both of them are built with 16nm technology. Finally, with the standard Synopsys software tools (*e.g.*, Design Compiler [14], VCS [16] and Prime Time [15]), we further evaluate the performance of FLASH if implemented as an ASIC.

Overall, some of our key results are as follows:

- Across the nine concrete cryptographic operations (§6.2), FLASH (with FPGA implementation) outperforms CPU and GPU by $10.4\times$ – $14.0\times$ and $1.4\times$ – $3.4\times$, respectively.
- Over the nine realistic FL applications (§6.3), FLASH (with FPGA implementation) can consistently outperform CPU and GPU by up to $6.8\times$ and $2.0\times$, respectively.
- Our software evaluation of FLASH as an ASIC with 12nm and 28nm fabrication techniques (§6.5) shows that it can achieve $23.6\times$ and $7.1\times$ *additional* performance improvement upon the FPGA implementation, respectively.

As a final note, we are fully aware that there exist various other privacy-preserving techniques [27, 71, 81, 85]. However, in current industry-level deployments, Paillier and RSA schemes built with the nine cryptographic operations we investigated in this work are, to date, the most widely adopted approach in cross-silo FL systems [2, 41, 49], primarily due to the reason that they can achieve relatively better performance and are easier to use compared to other privacy-preserving schemes. Our goal is to provide plug-and-play acceleration capability for these industry-level cross-silo FL systems.

2 Cross-silo Federated Learning

FL was first proposed by Google to train a language model for keyboard input prediction from massive Android devices without leaking privacy-sensitive data [36, 90]. Recently, FL has evolved from the above cross-device scenarios to collaboratively train machine learning models across different data silos, *i.e.*, cross-silo FL [44, 48, 89]. A data silo is a repository or collection of data under the control of a single entity (*e.g.*, institution, company, *etc.*), and is isolated from other entities due to the ever-improving management regulations

or laws [50]. Cross-silo FL enables machine learning among these data silos and supports both vertical and horizontal FL.

Horizontal FL: As shown in Figure 1a, participants in Horizontal FL have different sample spaces but the same feature space, and each participant owns the labels of its samples. In most cases, there is an arbiter for parameter aggregation (the arbiter is a third-party server to assist the FL computations). To train a model, each participant trains local model $\boldsymbol{w}$ with its own samples and encrypts its model weights via PHE (*e.g.*, Paillier [74]). Then all participants send their encrypted weights to the arbiter, and the arbiter directly performs an aggregation over the received ciphertexts to obtain the global model. Eventually, the arbiter sends the aggregated global model to all participants for next-round computation.

Vertical FL: As shown in Figure 1b, participants in Vertical FL have the same sample space but different feature spaces. Only one participant holds the label of the FL task. Before training a model, all participants have to align the samples among different data silos based on the common IDs (similar to joining two tables in a database based on the common IDs). One of the most commonly used algorithms is RSA blind signature-based PSI (RSA-PSI) [45]. After sample alignment, all participants can follow a pre-defined protocol for model training, such as Vertical Linear Regression (Please see Table 1 in [89]) and SecureBoost [42]. During the process, participants use PHE to encrypt their intermediate results and exchange them with other participants or the arbiter.

For interested readers, we have provided a detailed explanation of how cross-silo FL works and its security analysis in Appendix A.

Summary: Various cryptographic technics are used in cross-silo FL. These cryptographic technics are composed of various operations, *e.g.*, data encryption/decryption via additively HE, computation over ciphertext, *etc*., and we call these operations as *cryptographic operations* in this paper.

3 Analysis of Cryptographic Operations

3.1 Cryptographic Operations

In this section, we present nine cryptographic operations that are widely used in cross-silo FL. Our study is based on the implementation of FATE [2], the most widely adopted open-source framework for cross-silo FL. However, our analysis can also be applied to other cross-silo FL frameworks, *e.g.*, FedLearner [4], TF Encrypted [18], *etc*. Specifically, these nine cryptographic operations are as follows:

O1. Paillier Encryption. This operation uses Paillier [28,73], an additively homomorphic cryptographic algorithm, to encrypt cleartexts into ciphertexts. The operation is mainly used for protecting the intermediate data during model training.

O2. Paillier Decryption. This operation decrypts Paillier ciphertexts into cleartexts. It is used when participants need to decrypt the intermediate results for local model updates in the training phase.

O3. Ciphertext Matrix Addition. This operation is used to add two matrices (or vectors/values) of ciphertexts. As Paillier is used, ciphertexts can be summed up.

O4. Ciphertext & Cleartext Matrix Element-wise Multiplication. This operation performs Hadamard product [58] between ciphertext matrix and cleartext matrix.

O5. Ciphertext & Cleartext Matrix Multiplication[4]. This operation performs matrix multiplication between two matrices of ciphertexts and cleartexts, respectively.

O6. Ciphertext Histogram Building. This operation performs addition operations over encrypted gradient statistics to build decision trees [42].

O7. RSA Encryption/Decryption. This operation conducts encryption or decryption with the public or private key of the RSA algorithm correspondingly. This operation is used when multiple participants try to perform PSI for sample alignment [45].

O8. RSA Blind. This operation blinds the cleartexts with encrypted random numbers.

O9. RSA Unblind. This operation unblinds RSA ciphertexts to remove the random numbers from the ciphertexts.

As shown subsequently (§3.2), these cryptographic operations have a large impact on the performance of cross-silo FL applications due to the following two reasons:

- **High time complexity:** These operations are of high computation complexity, *e.g.*, Paillier encryption has a time complexity of $O(2^N)$. Thus these algorithms are expensive to compute.
- **Large number computation:** Cryptographic operations significantly inflate data, yielding large numbers, *e.g.*, 2048-bit integer. The large number will need to be divided into multiple small numbers and executed on the current CPU architecture with limited parallelism.

3.2 Quantifying the Performance Impact

We now quantify the performance impact of these cryptographic operations with realistic cross-silo FL applications through testbed experiments.

Our testbed is equipped with an Intel Xeon Silver 4114 CPU [5] and 192GB memory. We choose three most widely-adopted vertical FL applications and one horizontal FL application for evaluation: RSA blind signature-based PSI (RSA-PSI), Vertical Logistic Regression (VLR) [53], SecureBoost Decision Tree (SBT) [42] and Horizontal Logistic Regression (HLR). The dataset we use is a commercial dataset from a bank with $\sim 100,000$ samples and 80 features. For vertical FL applications, the dataset is vertically partitioned into two parts: one part contains 80 features while the other contains one feature. We first perform RSA-PSI to obtain the

[4] To efficiently process a large matrix, we will use optimization algorithms such as blocking the matrix and performing multiplications of the blocked matrices. Thus this operation is not a simple combination of matrix element-wise multiplication (**O4**) and addition (**O3**).

Applications & Their Sub-tasks		Involved Operations	w/o CO (s)	w CO (s)	Degradation
RSA-PSI	Computing intersection	**O7, O8, O9**	7.91	20.62	2.60× ↓
VLR (One Epoch) Total: 12.05× ↓	Encrypting logits	**O1**	0	32.05	-
	Aggregating logits	**O3**	0.63	2.04	3.23× ↓
	Computing fore gradients[a]	**O3, O4**	0.74	2.92	3.93× ↓
	Computing gradients	**O3, O4, O5**	3.77	135.96	36.08× ↓
	Decrypting gradients	**O2**	0	0.04	-
	Computing loss	**O1, O3, O4**	4.08	8.22	2.02× ↓
SBT (One Epoch) Total: 3.49× ↓[b]	Encrypting gradients	**O1**	0	54.71	-
	Aggregating gradients	**O3, O6**	12.02	223.91	18.62× ↓
	Split information synchronization	**O2**	3.62	13.03	3.60× ↓
HLR (One Epoch) Total: 60.74× ↓[b]	Computing gradients	**O3, O4, O5**	1.73	177.94	102.80× ↓
	Model update	**O3, O4**	0.0002	0.10	526.10× ↓
	Model re-encryption	**O1, O2**	0	0.69	-

[a] According to Federated Logistic Regression [53], the gradient computation takes two steps: fore gradients computation and gradients computation.
[b] The overall performance degradation of SBT/HLR is smaller than the sum of those sub-tasks because we do not include pure cleartext computation or networking communication sub-tasks in the table.

Table 1: Performance penalty caused by cryptographic operations (CO) with different cross-silo FL applications.

data intersection. Then, we run VLR and SBT over the data intersection, respectively. For horizontal FL, the dataset is horizontally partitioned into two parts, each with $\sim 50,000$ samples. The four applications are executed both with cryptographic operations implemented using GMP (w/ CO) and without cryptographic operations (w/o CO). To implement model training w/o CO, we modify the code of FATE to skip these cryptographic operations. To perform a fine-grained analysis, we also break down these four applications into sub-tasks, and for each sub-task, we show the adopted cryptographic operations. All the applications are executed with ten CPU cores in parallel. Table 1 shows the results, and we make the following observations:

- **Cryptographic operations considerably degrade the performance.** In general, cryptographic operations significantly degrade the performance of cross-silo FL applications. In our experiment, the cryptographic operations cause RSA-PSI, VLR, SBT and HLR to suffer 2.60×, 12.05×, 3.49× and 60.74× performance degradation, respectively. Moreover, the combinations of these cryptographic operations can degrade the performance from $\sim 2.02\times$ to $\sim 526.10\times$.
- **Not all the cryptographic operations are used at all times simultaneously.** Different FL applications use different cryptographic operations, and even within a single application, different sub-tasks use different operations.

3.3 Challenges of Offloading Cryptographic Operations

To accelerate these cryptographic operations, we chose GPU as our first attempt, as it has been widely adopted in various offloading scenarios. However, as our exploration proceeds, we find that the cryptographic operations in cross-silo FL require complicated pipeline computation and significantly inflate the data, posing the following challenges for GPU.

The hardware architecture of GPU is tailored for performing data parallelism over tensors, which are mostly short numbers. However, as we will show in §4.2.1, to efficiently execute cryptographic operations, we have to use several steps to optimize the computation, *e.g.*, Montgomery Modular Multiplication [65], in which pipeline parallelism is needed. Furthermore, massive large numbers should be stored in the shared memory during the pipeline execution. However, these large numbers, *e.g.*, 2048-bit integer numbers, can easily overflow the on-chip memory of the GPU. For example, the amount of shared memory per SM is 96 KB for NVIDIA V100 [13]. No more than 384 2048-bit integer numbers can be stored inside one SM. Therefore, after processing a small amount of data, the GPU has to swap data between the shared memory and external memory, interrupting the pipeline execution. While it is possible to solve the aforementioned limitations, it requires complicated memory orchestration, such as a suitable double-buffering [33], which may pose further challenges.

To this end, our paper does not take this direction but moves one step further beyond the existing GPU architecture, by seeking a more efficient, customized hardware acceleration architecture for cross-silo FL. We envision that with a customized hardware architecture, we can implement fine-grained pipelining for those cryptographic operations with large numbers. The hardware can also support variable bit-widths to match the cross-silo FL scenarios where different public key sizes are used (which yield large numbers with different bit-widths). Furthermore, with a customized design, we are able to invest sufficient on-chip memory for caching large numbers used in the pipeline execution. The data swapping between on-chip and external memory can be part of the pipeline to avoid the performance penalty mentioned above.

In this paper, we follow the rule-of-thumb approach to use FPGA as a prototype and evaluate the potential of ASIC via software tools [30, 51, 87]. However, we confront the following two challenges in our design:

1. Inadequate hardware acceleration due to limited resources. As identified in §3.2, all the cryptographic operations cause a performance penalty, so we should offload all of them to hardware. Furthermore, to realize sufficient accelera-

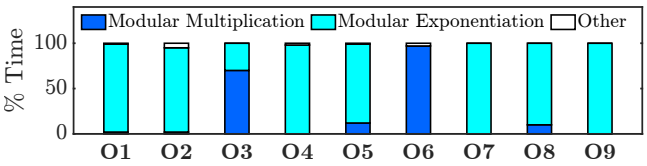


Figure 2: Cryptographic operation computation time analysis.

tion, each operation requires multiple hardware instances of accelerating modules/circuits for high parallelism. However, in practice, the hardware chip has limited resources, and if we naïvely offload all cryptographic operations to the chip, each operation has inadequate resources to be fully accelerated. Taking the DSP resources as an example, our preliminary implementation on VU13P FPGA [23] chip shows that to accelerate Paillier encryption (**O1**) by 2×, we need to use 2630 DSPs. Yet, a high-end FPGA chip, such as VU13P [23], only has 12288 DSPs, leaving $<$ 1365 DSPs for one operation on average[5] (some DSPs are reserved for PCIe, memory controller, *etc.*). Thus, directly offloading *all* operations on VU13P FPGA chip leads to only $\sim$ 50% acceleration on average. A similar problem also applies to the ASIC design.

2. Insufficient resource utilization due to static offloading. Different from software, hardware function is static after being configured/programmed/taped out, thus it cannot change its function dynamically. Nevertheless, as shown in §3.2, not all the cryptographic operations are used at all times simultaneously. Consequently, if we statically offload all cryptographic operations on the hardware chip, only part of these cryptographic operations is used at a time. Therefore, such static offloading causes low resource utilization and further leads to suboptimal performance.

3.4 Opportunities with Accelerating Basic Operators

To overcome the above challenges, we further take a look at the internal of these nine cryptographic operations. We discover that *all* these operations are composed of two basic operators: modular multiplication and exponentiation. Then we further find that the performance of these operations is mainly decided by the two basic operators.

Paillier Encryption: Given the public key (n,g) and data m $(0 \le m < n)$, the Paillier encryption algorithm takes two steps: (1) selecting a random number r where $0 < r < n$ and $r \in \mathbb{Z}_n^*$; (2) computing ciphertext $c = g^m \cdot r^n \mod n^2$. The formula can also be simplified to $c = (1+mn) \cdot r^n \mod n^2$ by setting $g = (1+n)$. We use $[[\cdot]]$ to denote the Paillier encryption, *e.g.*, $c = [[m]]$.

Homomorphic Addition: Given two plaintext a and b, homomorphic addition guarantees that $[[a]] \bigoplus [[b]] = [[a+b]]$. In Paillier cryptosystem, $[[a]] \bigoplus [[b]]$ is defined as $[[a]] * [[b]] \mod n^2$. The homomorphic addition is used by operations **O3**, **O5** and **O6**.

[5] We will show later that all these nine operations share similar building blocks, thus they require similar resources to implement.

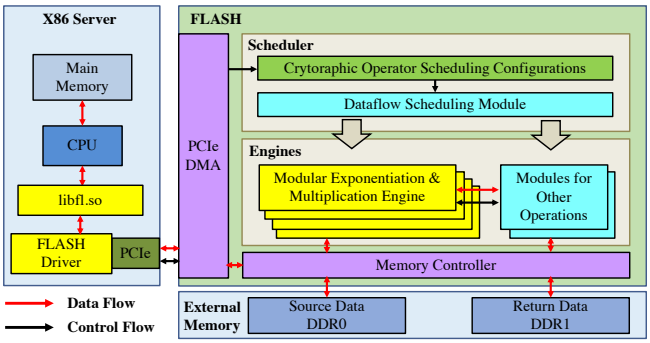


Figure 3: FLASH architecture.

Homomorphic Multiplication: Given a plaintext a and k, the homomorphic multiplication is denoted by $k \cdot [[a]]$. It can be actually regarded as a homomorphic addition: $\Sigma^k [[a]]$. Thus, $k \cdot [[a]] = [[a]]^k \mod n^2$. The homomorphic multiplication is used by operations **O4** and **O5**.

Paillier Decryption: Given the public key (n,g), private key (p,q) and ciphertext c, the Paillier Decryption algorithm can be optimized via Chinese Remainder Theorem (CRT) to reduce the original workload to only about one-quarter of the original decryption algorithm. To use CRT, we define L_p and L_q to be $L_p(x) = \frac{x-1}{p}$ and $L_q(x) = \frac{x-1}{q}$. The decryption algorithm takes the following three steps: (1) computing $m_p = L_p(c^{p-1} \mod p^2) L_p(g^{p-1} \mod p^2)^{-1} \mod p$; (2) computing $m_q = L_q(c^{q-1} \mod q^2) L_q(g^{q-1} \mod q^2)^{-1} \mod q$; and (3) computing plaintext $m = \mathrm{CRT}(m_p, m_q) \mod n$.

RSA-related Operations: The RSA-related operations are used in RSA blind signature-based PSI [45]. It is commonly known that the core of these RSA-related algorithms is either modular multiplication or modular exponentiation.

Through the above mathematical analysis, we find that *all* cryptographic operations used in cross-silo FL are built upon the two basic operators: modular multiplication and exponentiation[6] . Then, we further perform testbed experiments to investigate how these two basic operators impact the performance of the nine original cryptographic operations. The results are shown in Figure 2. Clearly, we find that across *all* nine original operations, the two basic operators occupy $>$ 95% of the total execution time.

Observation: We can compose all the nine cryptographic operations with these two basic operators, and by accelerating these two basic operators, all the nine original operations used in cross-silo FL applications can be effectively accelerated.

4 The FLASH Design

Inspired by the above observation, we present FLASH, a high-performance hardware acceleration architecture for cross-silo FL. This section describes how we design FLASH in detail. Please note that our design has been fully implemented in our FLASH prototype with FPGAs as well as rigorously evaluated

[6] Appendix B and C provide more details of these operations.

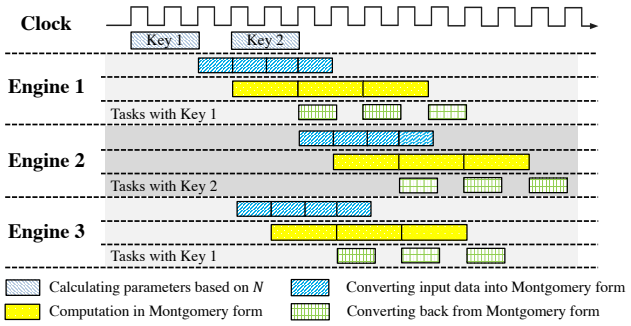


Figure 4: Pipeline executions of both inter- and intra-engines.

with the Synopsys tools for the ASIC.

4.1 Architecture Overview

Figure 3 shows the overall architecture of FLASH. It contains a hardware acceleration card and an integrated software package. The accelerator card can be plugged into a server via PCIe Gen3×16 interface. The server is installed with cross-silo FL software, *e.g.*, FATE. The software can invoke FLASH's software package to *offload* the cryptographic operations on the card for efficient acceleration.

The idea of our FLASH design is that it does not directly offload all cryptographic operations on the hardware, but leverages the insights of our observation to (1) utilize the limited programmable resource for most performance-critical basic operators: modular exponentiation and multiplication to achieve *adequate acceleration* (§4.2), and (2) design an on-chip dataflow scheduling module to dynamically compose different cryptographic operations on-demand based on these engines, achieving *high resource utilization* (§4.3). In addition, to make FLASH a general solution to support a wide range of cross-silo FL frameworks, our software package provides standard APIs. In this way, different cross-silo FL software can utilize FLASH by harnessing its APIs (§4.4).

4.2 Modular Exponentiation and Multiplication Engines

To implement modular exponentiation and multiplication operators as high-performance engines on hardware, FLASH makes the following design decisions. First, instead of directly offloading the modular exponentiation and multiplication operators, we optimize the algorithms of the two operators to make them suitable for the hardware implementation (§4.2.1). Second, based on the optimized algorithms, FLASH further leverages pipelining technologies to efficiently execute them with high parallelism (§4.2.2). Third, we provide sufficient on-chip memory for pipeline execution and make the data transfer as part of the pipeline to efficiently exchange data between off-chip memory and engines (§4.2.3).

4.2.1 Algorithm Optimization

The mathematical formulas of the two basic operators: modular exponentiation (Equation 1) and multiplication (Equation 2) are as follows:

Algorithm 1 Montgomery Modular Multiplication

▷ Given 3 input numbers X, Y and N, the Montgomery Modular Multiplication outputs $Z = X \cdot Y \cdot R^{-1} \mod N$, where R is a power of 2 and $\lfloor \log_2 R \rfloor = \lfloor \log_2 N \rfloor$.

Input: $X = (X_{d-1}, ..., X_0), Y = (Y_{d-1}, ..., Y_0), N = (N_{d-1}, ..., N_0), N'$, where
$N' = (-N)^{-1} \mod r$, ▷ N' is pre-computed in S1
$r = 2^w, d = \lfloor \log_r N \rfloor + 1$, ▷ r and d is used to split data
$\gcd(N, r) = 1$, with $N \times N' \equiv -1 \mod r$

Output: $Z = \text{ModMult}(X, Y, N) = X \times Y \times R^{-1} \mod N$

$Z = (Z_{d-1}, ..., Z_0) = 0$ ▷ Initialization
for all $i = 0, 1, ..., d-1$ **do** ▷ Loop on Y
  $\alpha = [X_0 \times Y_i]_{\text{low}}$
  $\beta = \alpha + Z_0$
  $q = [\beta \times N']_{\text{low}}$
  $\delta_1 = \alpha + [q \times N_0]_{\text{low}}$
  $\delta_2 = \delta_1 + Z_0$
  $Z_0 = [\delta_2]_{\text{low}}$
  $C = [\delta_2]_{\text{high}}$
  for all $j = 1, 2, ..., d-1$ **do** ▷ Loop on X
    $\delta_0 = [X_{j-1} \times Y_i]_{\text{high}} + [q \times N_{j-1}]_{\text{high}}$
    $\delta_1 = \delta_0 + [X_j \times Y_i]_{\text{low}} + [q \times N_j]_{\text{low}}$
    $\delta_2 = \delta_1 + Z_j + C$
    $Z_{j-1} = [\delta_2]_{\text{low}}$
    $C = [\delta_2]_{\text{high}}$ ▷ Carry higher bits
  end for
  $Z_{d-1} = C$
end for
if $Z \geq N$ **then**
  $Z = Z - N$
end if
return Z

$$P = m^e \mod n \qquad m, e, n \in \mathbb{Z}^+ \tag{1}$$

$$P = ab \mod n \qquad a, b, n \in \mathbb{Z}^+ \tag{2}$$

When used in cryptographic operations, all the numbers a, b, m, n are large numbers, leading to high computation complexity. Therefore, before designing FLASH's engines, we first apply some commonly-used optimization strategies in the cryptographic research community to optimize the two basic operators, including Binary Exponentiation [52] and Montgomery Modular Multiplication [65], *etc*. The advantages of using these optimization strategies are: (1) lowering the number of multiplications used in modular exponentiation from $O(2^N)$ to $O(N)$ (N is the bit-width of e), and (2) replacing the modulo operation with the hardware-friendly bit-shifting operation. Appendix D provides details of how they work.

After applying these optimization methods, FLASH's modular exponentiation and multiplication operators require the following four stages to complete the computation:

S1. Preparing common data needed in Montgomery space based on the input data n. Since, in both Paillier and RSA cryptosystems, n is decided by the public key, we can reuse these prepared data for all computations with the same public key. This is common in cross-silo FL applications as they use one key for all cryptographic operations within one application.

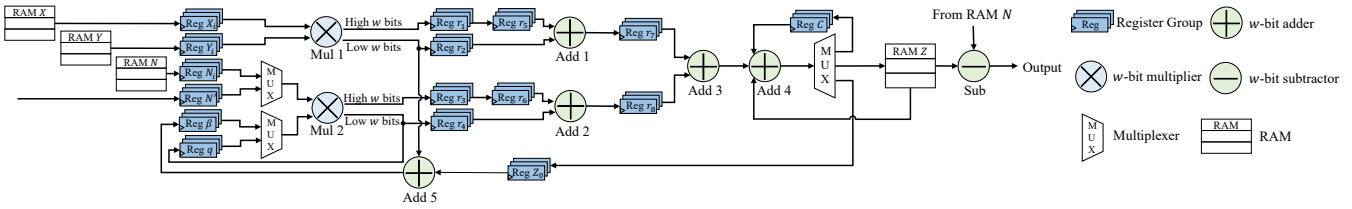


Figure 5: The Montgomery Modular Multiplication engine circuit design. Our circuit uses two multipliers and four on-chip RAMs for efficient pipelining. For more details of how this circuit works, please refer to Appendix E.

S2. Performing input data pre-precessing and converting them into Montgomery space.

S3. Performing computation in Montgomery form. Major operations in this stage include large-number multiplication, addition, and bit-shifting. No modulo operation is needed.

S4. Converting all output data of the operators out of Montgomery space.

4.2.2 Pipelining

We next introduce how FLASH efficiently performs the above four computation stages via inter- and intra-engine pipelining.

Inter-engine pipelining: To enable inter-engine pipelining, FLASH employs an engine pipeline stage manager to control the execution strategies for different stages. Figure 4 gives an overview of how these stages are pipelined. First, FLASH reserves S1 as a standalone stage, which can be executed in advance once it obtains the public key. Second, for all computation tasks with the same public key (Engine 1 and 3 in Figure 4), they can be executed in parallel once their data preparation is completed (S1). The start time of these engines may have a small gap of several clock cycles because FLASH adopts a round-robin strategy to dispatch stage executions. Third, for tasks with different keys (Engine 2 in Figure 4), they can be executed independently.

Intra-engine pipelining: FLASH further performs intra-engine pipelining within the most computation-intensive stage S3 to accelerate the stage's internal execution. The key design goals are: (1) FLASH should support variable bit-widths thus the application can choose the key length based on their security requirements; (2) the hardware resources should be fully utilized to achieve superb performance.

To achieve the first goal, FLASH builds an efficient pipeline that processes data based on radix-2^w arithmetic [60] (we use $w = 64$ in FLASH's implementation). Given any input data, we split it into d w-bit integers. For example, $d = 16$ when bit-width of X is 1024, and $d = 32$ when bit-width of X is 2048. Theoretically, the pipeline can be adapted for input data with any bit-width as long as the bit-width is or can be extended by complementary zeros to the integer multiple of d. After data splitting, the complete algorithm of S3 (Montgomery Modular Multiplication) is shown in Algorithm 1. Note, compared to the original Montgomery Modular Multiplication (shown in Algorithm D.2 in Appendix D.2), we make the following optimizations to make the algorithm more hardware-friendly: (1) the computation of S (*i.e.*, line 6 in Algorithm D.2) is separated into computations of lower w bits and higher w bits so that the bit-width required in operations (*e.g.*, addition) is halved; (2) the first iteration of the inner loop where $j = 0$ is unrolled to remove the conditionals in the original algorithm (*i.e.*, line 7 to 9 in Algorithm D.2) and keep the consistency of computation logic.

In Algorithm 1, the most computation-intensive operations are the multiplications of $X_j \times Y_i$ and $q \times N_j$ respectively (both operations require d^2 w-bit multiplications). Moreover, the data required in these two multiplications are totally independent. Therefore, we use two multipliers (one 64-bit multiplier consists of 32 DSP48E2 slices [22] on our FPGA prototype), Mul 1 and Mul 2, for these two multiplication operations and reuse them to execute the rest of the multiplication operations as well (operations assigned to Mul 1 and Mul 2 are marked with red and green respectively in Algorithm 1). Since the multiplier can continuously process data, to fully utilize the multiplier, we have the following design decisions. First, we design a circuit to fully pipeline the inner loop (line 10 to 16 of Algorithm 1). The circuit is shown in Figure 5 and due to the limited space, we defer a detailed description of how it works in Appendix E. Second, when the multiplications of i-th iteration finish and some other operations are still under execution, *e.g.*, addition operations in the right part of the Figure 5, FLASH allows direct starting the multiplications in $i+1$-th iteration to minimize the delay between different iterations. We also use Figure 6 to visualize how the operations in S3 are efficiently pipelined.

4.2.3 On-chip & Off-chip Memory

To provide sufficient on-chip memory for efficient pipeline execution, FLASH allocates four memory units for each modular multiplication and exponentiation engine (shown in Figure 5). For our FPGA prototype implementation, we use 4× 36Kbit BRAM units. While the on-chip memory is mainly used for pipeline execution, FLASH further exploits external memory (shown in Figure 3) for input, output and intermediate data storage. To achieve high performance, data exchange between on-chip and off-chip memory is part of the pipeline itself, *i.e.*, when the data at the on-chip memory is consumed, FLASH simultaneously fetches new data from the off-chip memory, so that the data fetching time can overlap with the computation time. As the data fetch time is typically shorter than the computation time, it effectively hides the off-chip

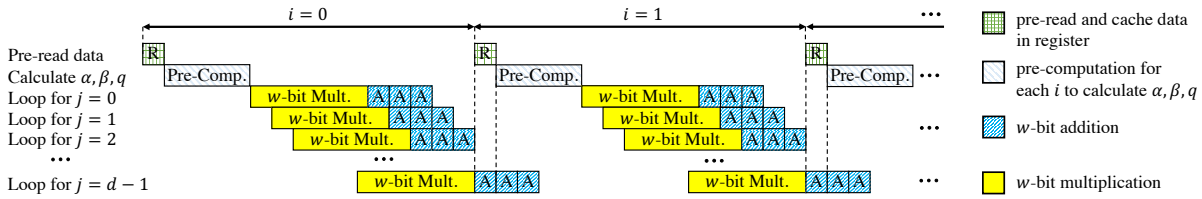


Figure 6: Efficient pipelining of Montgomery Modular Multiplication.

memory access latency, leading to perfect pipelining.

Moreover, we also design a hierarchical data distribution mechanism to solve the design difficulties encountered when manipulating large external memory. Specifically, instead of directly using long paths to send data from the memory controller to all engines, FLASH distributes data at multiple layers. Such a hierarchical mechanism leads to two advantages: (1) low design difficulties: since a small fan-out with short logical paths is required in each layer, the placement of the circuits is much easier; (2) improved performance: because the time constraints of the short path are much easier to meet, such method can allow a high operating frequency. Due to the limited space, we defer a detailed introduction in Appendix F.

4.3 Dataflow Scheduling

We now introduce how FLASH composes various cryptographic operations over basic engines through dataflow scheduling. First, we show how our engines can work at different modes (§4.3.1). Then, we present how different cryptographic operations are constructed by combining particular engines (§4.3.2).

4.3.1 Dynamic Engine Switching

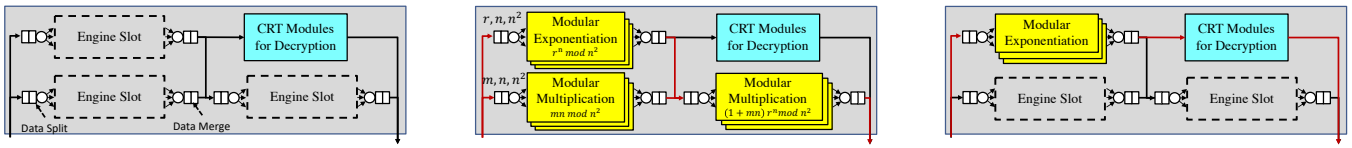
To build various cryptographic operations over basic engines, FLASH needs to enable dynamic engine switching between modular exponentiation and multiplication. Mathematically, modular exponentiation can be realized by performing modular multiplication multiple times. Thus, FLASH leverages a hardware control module to achieve it without reconfiguring hardware (it is almost impossible to reconfigure the ASIC). Specifically, to accelerate modular exponentiation, FLASH constructs a dataflow loop over the multiplication engine multiple times. In contrast, when the engine needs to execute modular multiplication, FLASH directs the dataflow through the modular multiplication engine once. While the design works well for most cryptographic operations that use either modular exponentiation or multiplication, it cannot directly support operations that simultaneously require both modular exponentiation and multiplication, *e.g.*, Paillier encryption (**O1**), matrix multiplication (**O5**), in which FLASH has to decide the ratio of engines in different modes.

How to decide the ratio? We use domain knowledge in cross-silo FL applications to decide the ratio. Taking matrix multiplication operation (**O5**) as an example, it first performs ciphertexts and cleartexts multiplication (requires modular exponentiation) and then ciphertexts addition (requires modular multiplication). Considering the modular exponentiation, the exponent e is a cleartext, which has a common bit-width of 64. As mentioned in §4.2.1, since we use Binary Exponentiation to optimize the modular exponentiation, the number of modular multiplication required may vary from 64 to 127 depending on the specific value of cleartext. On average, 96 modular multiplications are required. Thus, the throughput of the modular exponentiation should be $\sim 1/96$ of modular multiplication. Based on this, we can adjust the ratio of the engines working in different modes to make the throughput of both modular exponentiation and modular multiplication balanced. In this way, the hardware resources can be efficiently utilized and no engines will sit idle.

4.3.2 Building Cryptographic Operations

As shown in Figure 7a, the core idea of dataflow scheduling is to use an on-chip controller to dynamically determine: (1) which data paths (they are logical paths that do not reflect the physical wiring) should be active, and (2) what to put in the engine slots, based on which operation is offloaded. Each engine slot contains one data splitting module and one data merging module to distribute data to different engines and aggregate results from these engines, respectively. These data splitting and merging modules have physical wires to all engines, and by configuring which wires are active, we can logically assign engines to these engine slots. We also design a hierarchical data distribution mechanism, as mentioned in §4.2.3, for better performance.

Specifically, we can construct a Paillier encryption operator by following the dataflow scheduling strategy shown in Figure 7b. As mentioned in §3.4, the Paillier encryption follows equation: $c = (1+mn) \cdot r^n \mod n^2$. So we can distribute the data m, n, n^2 to modular multiplication engines (these engines are denoted E_1) to calculate $r_1 = mn \mod n^2$ and distribute the data r, n, n^2 to modular exponentiation engines (these engines are denoted E_2) to calculate $r_2 = r^n \mod n^2$. Then the results can be further sent to modular multiplication engine (these engines are denoted E_3) to calculate $(1+r_1) \times r_2 \mod n^2$. Please note that the $1+r_1$ is completed in the input data pre-precessing stage (S2 in §4.2.1) with a lightweight dedicated hardware module. The ratios of E_1, E_2 and E_3 are determined through the strategies discussed above, thus we can assign particular numbers of engines to these engine slots. Similarly, Figure 7c shows the dataflow used in Paillier decryption. In this case, FLASH uses other modules besides modular exponentiation and multiplication engines to real-



(a) All available dataflow scheduling paths. (b) Dataflow for encryption. (c) Dataflow for decryption.

Figure 7: Dataflow scheduling. Black arrow indicates all available paths for dataflow scheduling while red arrow indicates the active paths for a particular cryptographic operation. Each engine slot can have multiple engines.

ize the decryption operation. In particular, we use CRT to optimize the decryption algorithm as discussed in §3.4, thus FLASH implements several CRT modules to accelerate this operation. We put all engines, working in modular exponentiation mode, in the top left corner engine slot to achieve high resource utilization.

4.4 Software Integration

While our current implementation integrates FLASH with FATE, the design of FLASH is generic and works with other cross-silo FL systems/frameworks. They can harness FLASH by using the standard APIs.

For easy integration, FLASH provides Python NumPy-similar APIs as shown in Listing 1. The Python APIs are also wrappers of the C/C++ library: `libfl.so`. Besides providing standard APIs, the `libfl.so` library manipulates the status of all installed FLASH accelerators, such as temperature, workload, *etc*.

By using these APIs, users can easily create encrypted scalar, vector, or matrix via Paillier or RSA encryption method. Users can further perform homomorphic addition and multiplication operations over these data. To reduce the data exchange between the FLASH accelerator and the host, we put the computation results on the off-chip memory unless the `get` API is used. As shown in §4.2.3, the data exchange between on-chip and off-chip memory is efficiently pipelined, leading to better end-to-end performance. Moreover, since `libfl.so` works in a stateless way, it can be easily scaled out to support different tasks from various FL applications.

Listing 1: FLASH's NumPy-like APIs

```
import flash_np as np
# Generating two Paillier-encrypted arrays accelerated by FLASH
x1 = np.array([1, 2, 3], encryption="paillier")
x2 = np.array([4, 5, 6], encryption="paillier")

x3 = x1 + x2 # Homomorphic addition
x4 = np.array([1, 2, 3], encryption=None)
x5 = x4 * x1 # Ciphertext & cleartext multiplication

x3.decrypt() # Decrypting the ciphertext
x5.decrypt()

x3.get() # Transferring the data from accelerator to host
x5.get()
```

Multi-accelerator Support: The server-side software also enables multi-accelerator support. If there are multiple FLASH accelerators on the server, when applications invoke the APIs, `libfl.so` will break the task into multiple sub-tasks and dispatch them to multiple accelerators. The dispatching strategy is the least workload first and can be configured to use different strategies, such as round-robin.

5 Implementation

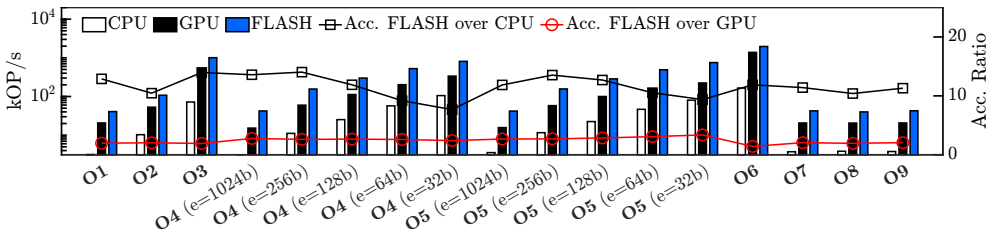
Prototype Implementation with FPGA: We implement FLASH with FPGA using $\sim 30,000$ lines of Verilog [84] code. We use Xilinx Virtex UltraScale+ VU13P chip [23] in our implementation. FLASH implements 300 modular exponentiation and multiplication engines with the chip. As the VU13P chip consists of four dies, we need to distribute components on different dies in a balanced way to achieve high resource utilization. In our implementation, we first place large modules such as PCIe and DDR controllers on separate dies with the consideration that they should be close to the location of their corresponding I/O pins. Then, with the settle-down of large modules, we place different numbers of engines on different dies to make the resource utilization of each die approximately the same to avoid the possibility of local congestion. As a final note, the operation frequency of our FPGA implementation is 300MHz while we achieved $\sim 88\%$ DSP resource utilization, which, to the best of our knowledge, is relatively high in FPGA's industry [92].

Server-side Software Stack Implementation: Our implementation of FLASH's server-side software contains $\sim 10,000$ lines of C/C++ and Python code. This includes modifications of FATE to harness FLASH's acceleration capacity. We mainly modify the `federatedml` module [3] in FATE by replacing normal collection operations with FLASH's NumPy-like APIs. We further use Xilinx DMA (XDMA) IP Reference driver [21] for high-performance direct memory access through the PCIe interface.

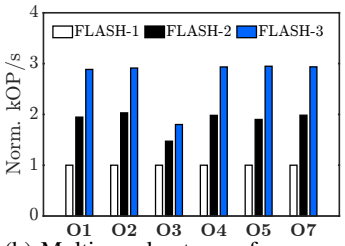
Evaluating FLASH as ASIC: We leverage multiple standard software to assess the FLASH design as an ASIC. Specifically, we first use Synopsys Design Compiler [14] to convert FLASH's design logics into physical implementations, *i.e.*, netlist, over both 12nm and 28nm technology libraries. Then, we use Synopsys VCS [16] to verify that the generated netlist functions correctly and use Synopsys Prime Time [15] for static timing analysis to validate that the netlist satisfies all timing constraints. More evaluation results of the ASIC performance will be discussed in §6.5.

6 Evaluation

In this section, we first present our evaluation methodology (§6.1). Then we show that for the nine cryptographic



(a) Cryptographic operation performance of all compared schemes. The left Y-axis is associated with the bar chart and is in the log scale. The right Y-axis is associated with the line chart.



(b) Multi-accelerator performance with selected operations.

Figure 8: Performance of cryptographic operations.

	FPGA	Logic Cells	DSP	Public Key $N = 1024$bit		Public Key $N = 2048$bit	
				Encryption (kOP/s)	**Decryption** (kOP/s)	**Encryption** (kOP/s)	**Decryption** (kOP/s)
FLASH	VU13P	3,780,000	12,288	40.706	107.707	6.033	19.373
PCP [80]	7VX330T [25]	326,400	1,120	1.40625	1.15625	-	-
HLS [91]	VU9P [23]	2,586,000	6,840	5.238	5.238	-	-
SoC [31]	ZU9EG [26]	600,000	2,520	-	-	0.561	0.563

Table 2: Resource consumption & performance comparison among FLASH and other Paillier accelerators.

	Models	Datasets
Vertical FL	RSA-PSI [45] VLR [53] SBT [42]	CreditCard [1]
Horizontal FL	HLR MLP LSTM [57] DenseNet169 [59] ResNet50 [55] VGG16 [82]	CreditCard [1] FMNIST [86] Shakespeare [17] Cifar-10 [66]

Table 3: Models & datasets used in evaluation of FLASH.

operations, FLASH achieves up to 14.0× and 3.4× acceleration over CPU and GPU (§6.2), translating up to 6.8× and 2.0× speedup for realistic FL applications (§6.3), respectively. Finally, we evaluate the performance of FLASH as an ASIC (§6.5).

6.1 Methodology

Environment Setup: We use two X86 servers in our setup. Each server is equipped with a Mellanox CX-4 NIC [6] and connected to a Mellanox SN2100 [7] switch via 40Gbps DAC cables. To reflect realistic networking situations in real-world cross-silo FL deployments, we use `netem` [8] to limit the networking bandwidth to be 50Mbps [7]. As to other hardware configurations, each server is equipped with one Intel Xeon Silver 4114 CPU [5], 192GB memory, and one FLASH acceleration card (In the multi-accelerator experiment, each server will be installed with multiple acceleration cards). We deploy FATE v1.5 as the cross-silo FL framework.

Schemes Compared: We mainly compare the performance achieved by FLASH with that achieved by: (1) Original FATE that uses a highly-optimized GMP library to execute cryptographic operations with CPU (denoted as CPU in the following charts). We choose Intel Xeon Silver 4114 CPU similar to prior works [76]. All CPU experiments are executed with all ten physical cores in parallel. (2) GPU-based accelerator (denoted as GPU). We extend the GPU implementation of HAFLO [43] which only implements logistic regression. Note that only the cryptographic operations are accelerated by GPU in our experiments. We use NVIDIA P4 GPU because it has the same technology of 16nm and achieves the closest INT8 TOPS as FLASH (although $\sim 2\times$ better. P4 reaches ~ 20 INT8 TOPS while FLASH achieves ~ 12.9 INT8 TOPS).

Performance Metrics: We use *the number of operations performed per second (OP/s)* as the metric when evaluating the performance of cryptographic operations, and *acceleration ratio over CPU/GPU* as the metric when evaluating FL applications.

6.2 Cryptographic Operations

To demonstrate that FLASH can efficiently accelerate the nine cryptographic operations, we compare the performance achieved by CPU, GPU, and FLASH, respectively. For operations **O4** and **O5**, we also evaluate different exponent bitwidths (32bit – 1024bit). The experiment results are shown in Figure 8a. In general, FLASH can consistently outperform CPU and GPU for all cryptographic operations. Specifically, FLASH outperforms CPU by 7.7× – 14.0× and GPU by 1.4× – 3.4×, showing that FLASH's hardware architecture fits the computational requirements of these cryptographic operations. Furthermore, we observe that when handling a larger exponent, FLASH tends to achieve a better acceleration ratio. For example, FLASH achieves 13.6× acceleration than CPU when evaluating O4 with $e = 1024$bit, but drops to 7.7× with $e = 32$bit. The results show that when the computation is more intensive, *i.e.*, with a large exponent, FLASH can achieve even better performance.

Multi-accelerator Support: We inspect how FLASH performs when we use multiple FLASH acceleration cards to speed up cryptographic operations. We evaluate one, two and three accelerators, denoted as FLASH-1, FLASH-2 and FLASH-3 respectively. For space limitation, we only pick some operations for demonstration: **O1**, **O2**, **O3**, **O4** with

[7] More details on how network bandwidth affects FLASH are shown in §6.4

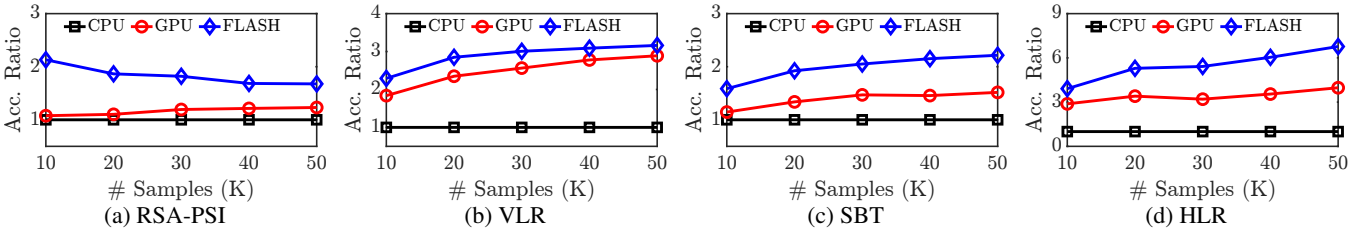


Figure 9: Performance of RSA-PSI, VLR, SBT, and HLR with changing data volumes.

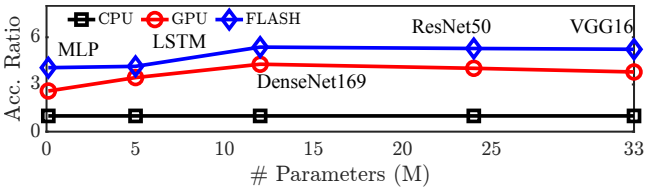


Figure 10: Performance of five deep learning applications.

$e = 1024$bit, **O5** with $e = 1024$bit, and **O7**. The results are shown in Figure 8b. We observe that for most cryptographic operations, *e.g.*, **O1**, **O2**, **O4**, **O5** and **O7**, the overall performance of FLASH is almost linear to the number of accelerators: FLASH-2 achieves 1.90× – 1.98× while FLASH-3 achieves 2.89× – 2.95× speedup for these operations. However, for **O3**, FLASH-2 and FLASH-3 only achieve 1.47× and 1.80× acceleration, respectively. The reason is as follows: the computation workload of **O3** is relatively low, thus the control overhead, *e.g.*, multi-accelerator synchronization, takes a considerable portion, leading to non-linear speedup. However, in the real-world use case, we envision that FLASH with multiple accelerators would still be an efficient solution to accelerate large-scale cross-silo FL applications.

Comparison with Other Paillier Accelerators: To give readers a better understanding of how efficient the FLASH's hardware design is, we further compare FLASH with some state-of-the-art hardware-based solutions, *e.g.*, Paillier Cryptoprocessor (PCP) [80], HLS [91], and SoC [31] based solutions. Moreover, due to the limited hardware resources, some of these works only implement a subset of cryptographic operations supported by FLASH. The comparison results are shown in Table 2. PCP and HLS report their data with public key $N = 1024$bit, while SoC uses $N = 2048$bit, thus we report the performance of FLASH with both $N = 1024$ and 2048bit. The results show that, compared to PCP, HLS and SoC, FLASH consumes 10.97×, 1.80×, 4.88× DSP resources, but delivers 28.95×, 7.77×, and 10.75× encryption acceleration and 93.15×, 20.56×, and 34.38× decryption acceleration, respectively. The results demonstrate that by using inter- & intra-engine pipelining and dataflow scheduling, FLASH can (1) deliver much better performance if utilizing comparable resources, and (2) support more complete functions.

6.3 Cross-silo FL Applications

We then present how FLASH can accelerate real-world cross-silo FL applications, including both vertical and horizontal. The models and datasets used are shown in Table 3. For vertical FL, before performing the model training algorithms, we first run a commonly used sample alignment algorithm: RSA blind signature-based PSI (RSA-PSI). Then, we perform Vertical Logistic Regression (VLR) [53] and Secure Boosting Tree (SBT) [42] algorithms over the data intersection (generated from PSI), respectively. For horizontal FL, we mainly evaluate Horizontal Logistic Regression (HLR) and five deep learning applications with different parameters. Each application runs a fixed number of epochs.

RSA-PSI, VLR, SBT, and HLR: The performance of RSA-PSI, VLR, SBT, and HLR is related to the data volumes. Thus we evaluate FLASH with different data volumes. The results are shown in Figure 9. In general, FLASH consistently outperforms CPU and GPU by achieving 1.6× – 6.8× and 1.1× – 2.0× acceleration ratio respectively. The results have demonstrated that by designing a tailored hardware acceleration architecture for cross-silo FL, we can effectively speed up FL applications and outperform the existing CPU/GPU architectures. Furthermore, we also notice that for RSA-PSI and VLR, GPU tends to reach a similar acceleration ratio as FLASH while processing more data. The reason is that for RSA-PSI and VLR, the cleartext computation, which is purely executed on the CPU, takes a significant portion of the total computation time. For example, in VLR, when handling 50K data samples in one epoch, after sufficient acceleration, the ciphertext computation takes $< 10\%$ of the total computation time. Therefore, the performance is mainly decided by the time of cleartext computation when the cryptographic operations are sufficiently accelerated, which leads to the results that FLASH and GPU achieve similar acceleration ratios over CPU. In contrast, for HLR and SBT, FLASH can achieve a higher acceleration ratio than GPU because the cryptographic operations of these two applications consume a significant portion of the total computation time.

Deep Learning Applications: We have further evaluated five deep learning models of different numbers of parameters with horizontal FL. The results are shown in Figure 10. We find that FLASH can outperform CPU and GPU by achieving 4.1× – 5.4× and 1.2× – 1.6× acceleration ratio respectively due to a similar reason discussed above. Furthermore, we note that for models with more parameters, *e.g.*, DenseNet169, ResNet50, VGG16, FLASH can achieve a higher speedup than models with fewer parameters, *e.g.*, MLP, LSTM. This experiment implies that for more computation-intensive tasks, FLASH can deliver more notable results.

	28nm Technology Library (Actual Op. Frequency: 800MHz)			**12nm Technology Library** (Actual Op. Frequency: 1120MHz)		
	Area/Unit (mm^2)	**# Unit**	**Total Area (mm^2)**	**Area/Unit (mm^2)**	**# Unit**	**Total Area (mm^2)**
PCIe Gen3×16	8.46	1	8.460 (6.56%)	5.25	1	5.250 (4.04%)
DDR4 Controller	7.25	2	14.500 (11.24%)	4.43	2	8.860 (6.81%)
Engine Logic	0.093	800	74.480 (57.72%)	0.046	1900	87.499 (67.26%)
Engine Memory	0.033	800	26.200 (20.30%)	0.014	1900	25.927 (19.93%)
Dataflow Scheduling & Others	5.399	1	5.399 (4.18%)	2.561	1	2.561 (1.97%)
Total	-	-	129.04 (99.26%)	-	-	130.10 (100.08%)

Table 4: ASIC resource evaluation for both 28nm and 12nm technology libraries.

Correctness: In addition to evaluating the performance of the above nine cross-silo FL applications, we also validate the final results of all compared schemes (we avoid the randomness by setting an identical random seed). Results have shown that all schemes yield identical results, showing that FLASH does not affect the correctness of model training.

Summary: Implemented as an FPGA prototype, FLASH has already largely outperformed CPU and achieved moderately better performance than GPU with comparable price. We also understand that high-end GPUs, *e.g.*, A100 [9], H100 [10], may outperform FLASH's FPGA prototype due to more advanced foundry technology, which are also of much higher price. However, they still share the drawbacks as mentioned in §3.3. The goal of our paper is to design a more efficient hardware acceleration architecture for cross-silo FL beyond existing CPU/GPU architectures. As we will demonstrate in §6.5, if implemented as an ASIC, the performance of FLASH can be significantly improved, which should boost the acceleration ratio for these applications to a much higher level.

6.4 FLASH Deep-dive

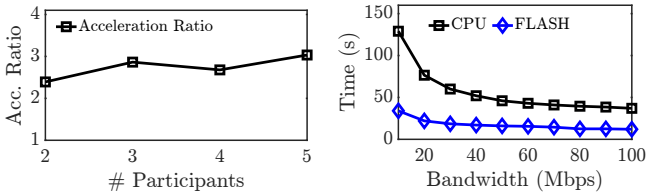
In this part, we mainly investigate (1) how the number of participants and (2) how the varying network bandwidth affects the performance of FLASH respectively.

Number of Participants: We evaluate VLR with two to five participants and measure the acceleration ratio of FLASH over CPU. The experiment result is shown in Figure 11a and we observe that in general, the number of participants does not largely impact the acceleration of FLASH.

Varying Bandwidth Setting: In this part, we use `netem` [8] to limit the available bandwidth between the two participants from 10Mbps to 100Mbps. We run VLR and measure the execution time of one iteration with both CPU and FLASH. Figure 11b shows the results and we can observe that when the bandwidth is over 50Mbps, the running times of both CPU and FLASH are stable, where FLASH outperforms CPU by $\sim 3\times$. The results show that the varying network bandwidth does not have a noticeable impact on FLASH.

6.5 ASIC Performance Assessment

Given that our FPGA-based prototype implementation of FLASH has performance limitations due to the intrinsic drawback of FPGA (*e.g.*, low operation frequency), in this section we intend to demonstrate some preliminary results of how FLASH performs as an ASIC. As introduced in §5, we use



(a) Performance of VLR with varying participants.

(b) Performance of VLR with varying networking bandwidth.

Figure 11: FLASH deep-dive.

standard software tools to assess the performance of FLASH if implemented as an ASIC. We evaluate FLASH's ASIC implementation with two technology libraries: 28nm and 12nm. Based on the industry experience, we set the operating frequency to be 1000MHz and 1400MHz, respectively, for these two technology libraries. Furthermore, we set the die area to be ~ 130mm^2. We believe this setting could balance the performance and power consumption for FLASH.

The detailed evaluation includes the following steps: First, we perform logic synthesis using Synopsys Design Compiler [14] to convert FLASH's design into netlist under the frequency and die area constraints. Table 4 illustrates the results. With 28nm technology library, we can allocate 800 modular multiplication and exponentiation engines successfully, while with 12nm technology library, we can allocate 1900 such engines. Second, we use Synopsys VCS [16] and Synopsys Prime Time [15] to confirm that both netlists are valid and function correctly. The third step is to estimate the performance gain of FLASH as an ASIC. Since the actual operating frequency after physical design should be lower than logic synthesis, we reduce the actual operation frequency by multiplying 80% by the design target for a conservative purpose.

Then, our final performance estimation is as follows. With 28nm technology library, we can allocate 2.67× engines compared to our FPGA implementation (800 v.s. 300), and the operation frequency of these engines is 2.67× that of the FPGA implementation (800MHz v.s. 300MHz), leading to an overall 7.11× performance gain on modular exponentiation operator (we use modular exponentiation operator as the metrics since it can fulfill the computation capacity of an engine). With 12nm technology library, we can allocate 6.33× engines (1900 v.s. 300) with 3.73× operation frequency (1120MHz v.s. 300MHz), and achieve 23.64× overall performance improvement. To give our readers a better understanding of FLASH's performance as an ASIC, we also

evaluate the modular exponentiation operator with a state-of-the-art GPU – NVIDIA A100 [9], our results show that A100 can only achieve 5.78× performance gain than our downscale FPGA prototype. Finally, we estimate the power consumption for a single engine, which is ∼ 16.6mWatt with 28nm technology library. Thus, the total power consumption for all engines is ∼ 13.28Watt. Although we do not have the power consumption data of other parts, *e.g.*, PCIe controller, we believe the total power consumption of FLASH as an ASIC should be significantly lower than the 120Watt of our FPGA implementation.

7 Discussion

Benefit to Future GPU/TPU Design: Nowadays, GPU [9–13] and TPU [63] have been widely adopted to accelerate deep learning applications. These accelerators mainly target accelerating convolution operations with tensors, where most numbers are short floats. In contrast, FLASH targets accelerating the identified nine cryptographic operations that are widely adopted in cross-silo FL. Moreover, most numbers used in FLASH are large numbers with a bit-width of 2048 bit or even longer. However, in some cross-silo applications, *e.g.*, horizontal deep learning in §6.3, both convolution and cryptographic operations exist. Therefore, we can foresee a co-design of GPU/TPU with FLASH in the future. For example, GPU/TPU can efficiently accelerate the local model training while FLASH is used to accelerate the model encryption and aggregation. We will make FLASH as an IP core in the future, and thus GPU/TPU vendors can use FLASH in their design to accomplish the aforementioned co-design.

FLASH v.s. Other GPU/FPGA Implementations: Some existing works also target accelerating modular exponentiation operations with GPU [43, 54, 61] or FPGA [29, 31, 34, 35, 80, 83, 91], which leverage similar algorithm optimization methods, *e.g.*, Binary Exponentiation [52] and Montgomery Modular Multiplication [65]. Yet, none of them performs a thorough analysis towards *all* cryptographic operations used in cross-silo FL and offloads them efficiently on the hardware-based accelerator as FLASH. Moreover, our idea of composing various cryptographic operations based on the two basic operators via dataflow scheduling is designed for the cross-silo FL scenarios, making FLASH a unique solution compared to prior FPGA-based implementations. As a final note, our design of FLASH is not limited to FPGA but is also applied to ASICs.

Extending to Other Application Domains: While FLASH is introduced for accelerating cross-silo FL, it can speed up applications in other domains as well. First, the Paillier and RSA cryptosystems used in cross-silo FL are also widely adopted in other domains. Thus FLASH can accelerate applications built on them, *e.g.*, electronic voting [47], electronic cash [38], and threshold cryptosystem [46]. Second, since FLASH's core idea is to accelerate modular multiplication and exponentiation operators, cryptographic systems/operations built on them, such as Diffie-Hellman key exchange [56], can also benefit from FLASH.

8 Related Works

Besides the related works discussed in §7, we further cover the following two related directions in this section.

Accelerating FL: Recently, due to the increasing deployment of FL, various research works have emerged to accelerate FL. MAGE proposes to optimize the secure computation from a memory perspective [67]. BatchCrypt tries to optimize the Paillier encryption by encoding a batch of quantized gradients into a long integer and encrypting it in one batch [95]. VF^{2}Boost proposes a novel training protocol to reduce the idle time of each participant [49]. Relative to them, we design FLASH from a different angle: accelerating the cryptographic operations used in FL, and our FLASH could be easily combined with these prior works.

Domain Specific Accelerator (DSA): DSA has recently been an emerging research topic that adopts hardware, *e.g.*, FPGA, ASIC, *etc.*, to accelerate particular applications [37, 62, 64, 75, 75, 76, 79, 93, 94]. For example, Tiara [94] uses FPGA and a programmable switch to accelerate layer-4 load balancing. FlowBlaze [75] offloads complex networking functions to a NetFPGA SmartNIC. hXDP [37] proposes to use FPGA to accelerate eBPF programs for fast XDP execution. MicroRec [62] offloads neural networks to FPGA to implement efficient recommendation systems. Various DSAs have been proposed to accelerate fully homomorphic encryption (FHE) [96], such as HEAX [76], F1 [78], BTS [64] and CraterLake [79]. Similar to them, FLASH follows the principle of DSA to design a hardware-based solution to efficiently accelerate cross-silo FL.

9 Conclusion

This paper presented FLASH, a hardware acceleration architecture for cross-silo FL. We have provided a fully functional FPGA prototype and evaluated our design as an ASIC. Extensive experiments with realistic applications and cryptographic operations have shown that FLASH is a viable solution.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Prof. Andreas Haeberlen for their constructive feedback and suggestions. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20-R, GRF-16215119, GRF-16213621, the NSFC Grant (62062005, 62102046), the Natural Science Foundation of Hunan Province (2022JJ30618), and the Scientific Research Fund of Hunan Provincial Education Department (22B0300). Kai Chen is the corresponding author.

References

[1] Credit Card Cheating Detection. https://www.kaggle.com/arslanali4343/credit-card-cheating-detection-cccd.

[2] Federated AI Technology Enabler. https://fate.fedai.org.

[3] Federated Machine Learning. https://github.com/FederatedAI/FATE/tree/master/python/federatedml.

[4] FedLearner. https://github.com/bytedance/fedlearner.

[5] Intel Xeon Silver 4114 Processor. https://www.intel.com/content/www/us/en/products/sku/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz/specifications.html.

[6] Mellanox ConnectX-4 EN Adapter Card Single/Dual-Port 100 Gigabit Ethernet Adapter. https://www.mellanox.com/products/ethernet-adapters/connectx-4-en.

[7] Mellanox SN2100 Open Ethernet Switch. https://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2100.pdf.

[8] netem. https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[9] NVIDIA A100. https://www.nvidia.com/en-us/data-center/a100/.

[10] NVIDIA H100. https://www.nvidia.com/en-us/data-center/h100/.

[11] NVIDIA P4. https://images.nvidia.com/content/pdf/tesla/184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf.

[12] NVIDIA P40 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/documents/nvidia-p40-datasheet.pdf.

[13] NVIDIA V100 Datasheet. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.

[14] Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[15] Synopsys Prime Time. https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html.

[16] Synopsys VCS. https://www.synopsys.com/verification/simulation/vcs.html.

[17] Text generation with an RNN. https://www.tensorflow.org/text/tutorials/text_generation.

[18] TF Encrypted. https://github.com/tf-encrypted/tf-encrypted.

[19] The GNU Multiple Precision Arithmetic Library. https://gmplib.org.

[20] Timing Closure User Guide. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx14_7/ug612.pdf.

[21] Xilinx DMA. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[22] Xilinx UltraScale Architecture DSP Slice User Guide. https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.

[23] Xilinx UltraScale+ FPGA Production Table and Production Selection Guide. https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[24] Xilinx Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics. https://docs.xilinx.com/v/u/en-US/ds923-virtex-ultrascale-plus.

[25] Xilinx Virtex7 FPGAs. https://www.xilinx.com/support/documentation/selection-guides/virtex7-product-table.pdf.

[26] Xilinx Zynq UltraScale+ MPSoCs. https://docs.xilinx.com/v/u/en-US/zynq-ultrascale-plus-product-selection-guide.

[27] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 308–318. ACM, 2016.

[28] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4):79:1–79:35, 2018.

[29] Timo Alho, Panu Hämäläinen, Marko Hännikäinen, and Timo D. Hämäläinen. Compact modular exponentiation accelerator for modern FPGA devices. *Comput. Electr. Eng.*, 33(5-6):383–391, 2007.

[30] Arash AziziMazreah and Lizhong Chen. Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pages 94–105. IEEE, 2019.

[31] Milad Bahadori and Kimmo Järvinen. A programmable soc-based accelerator for privacy-enhancing technologies and functional encryption. *IEEE Trans. Very Large Scale Integr. Syst.*, 28(10):2182–2195, 2020.

[32] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[33] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. Shredder: Gpu-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 14. USENIX Association, 2012.

[34] Thomas Blum. Montgomery modular exponentiation on reconfigurable hardware. In *14th IEEE Symposium on Computer Arithmetic (Arith-14 '99), 14-16 April 1999, Adelaide, Australia*, pages 70–77. IEEE Computer Society, 1999.

[35] Thomas Blum and Christof Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE Trans. Computers*, 50(7):759–764, 2001.

[36] Kallista A. Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019.

[37] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA nics. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 973–990. USENIX Association, 2020.

[38] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 101–115. IEEE Computer Society, 2007.

[39] Di Chai, Leye Wang, Kai Chen, and Qiang Yang. Secure federated matrix factorization. *IEEE Intell. Syst.*, 36(5):11–20, 2021.

[40] Di Chai, Leye Wang, Junxue Zhang, Liu Yang, Shuowei Cai, Kai Chen, and Qiang Yang. Practical lossless federated singular vector decomposition over billion-scale data. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, pages 46–55. ACM, 2022.

[41] Chaochao Chen, Jun Zhou, Li Wang, Xibin Wu, Wenjing Fang, Jin Tan, Lei Wang, Alex X. Liu, Hao Wang, and Cheng Hong. When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 2652–2662. ACM, 2021.

[42] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Dimitrios Papadopoulos, and Qiang Yang. SecureBoost: A lossless federated learning framework. *IEEE Intell. Syst.*, 36(6):87–98, 2021.

[43] Xiaodian Cheng, Wanhang Lu, Xinyang Huang, Shuihai Hu, and Kai Chen. HAFLO: gpu-based acceleration for federated logistic regression. *CoRR*, abs/2107.13797, 2021.

[44] Yong Cheng, Yang Liu, Tianjian Chen, and Qiang Yang. Federated learning for privacy-preserving AI. *Commun. ACM*, 63(12):33–36, 2020.

[45] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2010.

[46] Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*, volume 2727 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2003.

[47] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *Int. J. Inf. Sec.*, 9(6):371–385, 2010.

[48] Olga Fink, Torbjørn H. Netland, and Stefan Feuerriegel. Artificial intelligence across company borders. *Commun. ACM*, 65(1):34–36, 2022.

[49] Fangcheng Fu, Yingxia Shao, Lele Yu, Jiawei Jiang, Huanran Xue, Yangyu Tao, and Bin Cui. VF^{2}Boost: Very fast vertical federated gradient boosting for cross-enterprise learning. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 563–576. ACM, 2021.

[50] Michelle Goddard. The eu general data protection regulation (GDPR): European regulation that has a global impact. *International Journal of Market Research*, 59(6):703–705, 2017.

[51] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. HARE: hardware accelerator for regular expressions. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 44:1–44:12. IEEE Computer Society, 2016.

[52] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.

[53] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *CoRR*, abs/1711.10677, 2017.

[54] Owen Harrison and John Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 350–367. Springer, 2009.

[55] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[56] Martin E Hellman, Bailey W Diffie, and Ralph C Merkle. Cryptographic apparatus and method, April 29 1980. US Patent 4,200,770.

[57] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[58] Roger A Horn. The hadamard product. In *Proc. Symp. Appl. Math*, volume 40, pages 87–169, 1990.

[59] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017.

[60] M.K. Ibrahim. Radix-2^n multiplier structures: a structured design methodology. *IEE Proceedings E (Computers and Digital Techniques)*, 140(4):185–190, 1993.

[61] Keon Jang, Sangjin Han, Seungyeop Han, Sue B. Moon, and KyoungSoo Park. Sslshader: Cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[62] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B. Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. MicroRec: Efficient recommendation inference by hardware and data structure solutions. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[63] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12. ACM, 2017.

[64] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 711–725. ACM, 2022.

[65] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[66] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[67] Sam Kumar, David E. Culler, and Raluca Ada Popa. MAGE: nearly zero-cost virtual memory for secure computation. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 367–385. USENIX Association, 2021.

[68] Pankaj Kumbhare and Vamsi Krishna. Designing high-performance video systems in 7 series FPGAs with the AXI interconnect. *Xilinx, Inc., San Jose, CA, USA, Appl. Note*, 7:1–24, 2012.

[69] Gang Liang and Sudarshan S. Chawathe. Privacy-preserving inter-database operations. In *Intelligence and Security Informatics, Second Symposium on Intelligence and Security Informatics, ISI 2004, Tucson, AZ, USA, June 10-11, 2004, Proceedings*, volume 3073 of *Lecture Notes in Computer Science*, pages 66–82. Springer, 2004.

[70] Yang Liu, Yan Kang, Chaoping Xing, Tianjian Chen, and Qiang Yang. A secure federated transfer learning framework. *IEEE Intell. Syst.*, 35(4):70–82, 2020.

[71] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[72] M Rai Nigam and S Bande. AXI interconnect between four master and four slave interfaces. *Int. J. Eng. Res. Gen. Sci*, 2(4):2091–2730, 2014.

[73] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[74] Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Trans. Inf. Forensics Secur.*, 13(5):1333–1345, 2018.

[75] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani Brunella, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, and Felipe Huici. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 531–548. USENIX Association, 2019.

[76] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1295–1309. ACM, 2020.

[77] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[78] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*, pages 238–252. ACM, 2021.

[79] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 173–187. ACM, 2022.

[80] Ismail San, Nuray At, Ibrahim Yakut, and Huseyin Polat. Efficient paillier cryptoprocessor for privacy-preserving data mining. *Secur. Commun. Networks*, 9(11):1535–1546, 2016.

[81] Sinem Sav, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. POSEIDON: privacy-preserving federated neural network learning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[83] Daisuke Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2007.

[84] Donald E. Thomas and Philip Moorby. *The Verilog hardware description language (2. ed.)*. Kluwer, 1995.

[85] Han Tian, Chaoliang Zeng, Zhenghang Ren, Di Chai, Junxue Zhang, Kai Chen, and Qiang Yang. Sphinx: Enabling privacy-preserving online learning over the cloud. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2487–2501. IEEE, 2022.

[86] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[87] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind. AQUOMAN: an analytic-query offloading machine. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 386–399. IEEE, 2020.

[88] Liu Yang, Ben Tan, Vincent W. Zheng, Kai Chen, and Qiang Yang. Federated recommendation systems. In *Federated Learning - Privacy and Incentive*, volume 12500 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2020.

[89] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.*, 10(2):12:1–12:19, 2019.

[90] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving google keyboard query suggestions. *CoRR*, abs/1812.02903, 2018.

[91] Zhaoxiong Yang, Shuihai Hu, and Kai Chen. Fpga-based hardware accelerator of homomorphic encryption for efficient federated learning. *CoRR*, abs/2007.10560, 2020.

[92] Tian Ye, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. Performance modeling and FPGA acceleration of homomorphic encrypted convolution. In *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*, pages 115–121. IEEE, 2021.

[93] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: an fpga-accelerated embedding-based retrieval system. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 841–856. USENIX Association, 2022.

[94] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 1345–1358. USENIX Association, 2022.

[95] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 493–506. USENIX Association, 2020.

[96] Junxue Zhang, Xiaodian Cheng, Liu Yang, Jinbin Hu, Ximeng Liu, and Kai Chen. Sok: Fully homomorphic encryption accelerators. *CoRR*, abs/2212.01713, 2022.

Appendix

A Cross-silo Federated Learning

Cross-silo federated learning (FL) denotes the scenario where companies or institutions collaboratively train machine learning models without data privacy leakage [39,40,44,48,88,89]. Compared with cross-device FL, where participants are mobile devices, cross-silo FL focuses more on data security and incentive mechanism. From the data partition angle, Yang *et al.* proposed to categorize FL into horizontal FL, vertical FL, and federated transfer learning [70]. Federated transfer learning has rarely been applied in the industry and remains in the research stage. In most cases, cross-device FL contains only horizontal FL, while cross-silo FL usually contains both horizontal and vertical FL. Because of the different data partition situations, horizontal and vertical FL are different in model construction, protocol design as well as the utilized cryptographic systems.

A.1 Cross-silo Horizontal FL

Participants in horizontal FL have different sample ID spaces but the same feature space, as shown in Figure 1a. Each participant owns the labels of their samples. Therefore, horizontal FL enlarges the number of training samples to train a model with better generalization ability. In most cases, there is a third-party central server for parameter aggregation.

The tth iteration of the training process among three participants is shown as follows:

1. All participants negotiate about keys for encryption.
2. Each participant i trains local model $\boldsymbol{w}_i^t$ with its own samples and encrypts its model weights to $[[\boldsymbol{w}_i^t]]$ with either additively homomorphic encryption (*e.g.*, Paillier [74]).
3. Each participant i sends the encrypted weights to the central server.
4. The server receives the encrypted local model weights from all participants and aggregates them to global model weights $\frac{\sum_i [[\boldsymbol{w}_i^t]]}{n}$, where n stands for the number of participants. Because the weights are encrypted via additively homomorphic encryption, we can directly perform an aggregation over the ciphertext.
5. The central server sends the aggregated global weights to all participants.
6. Each participant receives the global weights and decrypts them locally. Then the participant can update its local model $\boldsymbol{w}_i^{t+1}$ with the decrypted global model.

After the federated training process, each participant obtains the same well-trained model. Thus, each participant can perform model inference locally.

A.2 Cross-silo Vertical FL

Participants in vertical FL scenarios have the same sample ID space but different feature spaces, as shown in Figure 1b. Under normal circumstances, only one participant holds the label of the FL task, which is called the active party. The other participants without labeling information are called passive parties. Compared with horizontal FL, vertical FL could enrich the feature information of samples. Unlike horizontal FL, the training process of vertical FL is conducted after the entity alignment stage, which aligns common samples while protecting privacy. Besides, the training schema of vertical FL is also different from horizontal FL. More specifically, each participant only owns part of the model parameters corresponding to the local feature dimensions. Hence, vertical FL cannot simply conduct the secure aggregation as horizontal FL does. In addition, various machine learning algorithms do not have a unified design of the vertical FL implementation.

Taking the federated linear regression [89] between two participants as an example, we illustrate the training process as follows:

1. Participants and the third-party central server negotiate about keys for encryption.
2. Passive party B computes local point estimate $u_{B,j}^t$ and partial loss $L_{B,j}^t$ for the jth aligned sample, then encrypts them to $[[u_{B,j}^t]]$ and $[[L_{B,j}^t]]$ with Paillier [74]. Active party A calculates local point estimate $u_{A,j}^t$.
3. Passive party B sends the encrypted numbers to the active party A.
4. Active party A receives the encrypted numbers and computes the total loss $[[L_j^t]]$ and the intermediate results $[[d_j^t]]$ used to calculate gradients.
5. Active party A sends $[[L_j^t]]$ to server and $[[d_j^t]]$ to passive party B.
6. Party B and party A separately compute encrypted gradients $[[\frac{\partial L_j^t}{\partial \boldsymbol{w}_B^t}]]$ and $[[\frac{\partial L_j^t}{\partial \boldsymbol{w}_A^t}]]$ and add random masks.
7. Both parties send the encrypted and masked gradients to the central server.
8. The third-party central server decrypts the received ciphertext to get the plain-text masked gradients and sends them back.
9. Party B and party A respectively remove the random masks from gradients and update the local partial model.

All participants of vertical FL should be involved in the inference stage since each of them only owns part of the whole model.

A.3 Security Analysis of Cross-silo FL

The adopted FL algorithms in this paper are proved secure under the semi-honest assumption [42, 89]. The semi-honest assumption means that each party does not violate the federated protocols and only tries to infer the sensitive data of other parties from the received messages. For the horizontal FL models, the transmitted model updates are protected by additively HE for aggregation. Therefore, nothing can be learned by the arbiter. Moreover, each party obtains the aggregated model updates and can only calculate the average model updates of the other parties. Hence, given more than two parties, the model updates computed over the local data of one party cannot be leaked to the other parties [89]. For the vertical federated linear models, the transmitted intermediate results are protected by random masks and HE, which reveals no information. Furthermore, from the obtained model updates, one party cannot infer the sensitive data of other parties without prior knowledge of their data structures [89]. For the vertical SecureBoost model, the active party with labels could learn some information agreed in advance, such as the instance spaces and the responsible parties of splits. However, under the protection of HE, the original data records of one party cannot be revealed to other parties, either [42].

B Paillier Cryptosystem

Paillier Cryptosystem [73] is a widely-used additively (*i.e.*, partially) homomorphic encryption scheme. Paillier cryp-

tosystem supports two kinds of operations, including the addition of two values of ciphertext and multiplication between ciphertext and cleartext. We will introduce Paillier key generation, encryption and decryption respectively in the following section.

Key Generation: Key generation of Paillier Cryptosystem follows the following steps.

1. Choose two random prime numbers p and q which satisfy that $\gcd(pq,(p-1)(q-1))=1$, where gcd stands for the greatest common divisor.
2. Compute $n=p\cdot q$.
3. Compute $\lambda=\mathrm{lcm}(p-1,q-1)$, where lcm means the least common multiple.
4. Randomly select an integer g which satisfies that $\gcd(\mathrm{L}_n(g^\lambda \mod n^2),n)=1$. Function $\mathrm{L}_n(x)$ is defined as $\mathrm{L}_n(x)=(x-1)/n$.
5. Compute $\mu=[\mathrm{L}_n(g^\lambda \mod n^2)]^{-1} \mod n$.

After the above computation, we will obtain the public key: (n,g) and private key: (λ,μ) respectively.

Encryption: The encryption algorithm of Paillier is straightforward and follows the following equation.

$$c=g^m\cdot r^n \mod n^2. \tag{B.1}$$

Optimization of Encryption: The encryption can be accelerated by assigning public key g as $n+1$. Therefore, the encryption algorithm is simplified as follows.

$$\begin{aligned} c &= g^m\cdot r^n \mod n^2 \\ &= (n+1)^m\cdot r^n \mod n^2 \\ &= [(\sum_{i=0}^{m}\binom{m}{i}\cdot n^i)\cdot r^n] \mod n^2 \\ &= [(1+m\cdot n)\cdot r^n] \mod n^2 \end{aligned} \tag{B.2}$$

One modular exponentiation operation is saved by this optimization. FLASH uses the optimized encryption for better performance.

Decryption: Paillier ciphertext c is decrypted to plaintext m with both public key (n,g) and private key (λ,μ):

$$m=\mathrm{L}_n(c^\lambda \mod n^2)\cdot\mu \mod n \tag{B.3}$$

Optimization of Decryption: The workload of the decryption algorithm of Paillier can be reduced with the Chinese Remainder Theorem (CRT). In this scheme, prime numbers p and q generated with the key pair are considered as the private key. The process of optimized decryption is shown below:

1. Compute $h_p=\mathrm{L}_p(g^{p-1} \mod p^2)^{-1} \mod p$ and $h_q=\mathrm{L}_q(g^{q-1} \mod q^2)^{-1} \mod q$.
2. Compute $m_p=\mathrm{L}_p(c^{p-1} \mod p^2)\cdot h_p \mod p$ and $m_q=\mathrm{L}_q(c^{q-1} \mod q^2)\cdot h_q \mod q$. Function $\mathrm{L}_p(x)$ and $\mathrm{L}_q(x)$ are defined by $\mathrm{L}_p(x)=(x-1)/p$ and $\mathrm{L}_q(x)=(x-1)/q$. It can be proved that $m_p=m \mod p$ and $m_q=m \mod q$, where m is the plaintext corresponding to ciphertext c.
3. Apply CRT to recombine the modular residues. $m=\mathrm{CRT}(m_p,m_q) \mod pq$.

With the optimization above, the workload can be reduced to only about one-quarter of the original decryption algorithm, leading to better performance. FLASH also uses optimized decryption in its implementation.

C RSA Intersection

RSA (Rivest–Shamir–Adleman) is an asymmetric public-private key method used to securely transfer data [77]. The whole RSA algorithm mainly contains three operations: key generation, encryption, and decryption.

Key Generation: The generation process is shown below:

1. Randomly choose two distinct prime numbers p and q.
2. Compute $n=p\cdot q$.
3. Compute $\lambda(n)=\mathrm{lcm}(p-1,q-1)$.
4. Randomly choose a number e such that $1<e<\lambda(n)$ and $\gcd(e,\lambda(n))=1$.
5. Compute d by solving $d\cdot e=1 \mod \lambda(n)$.

Generally speaking, (n,e) is regarded as a public key, while d is regarded as a private key.

Encryption: Using public key (n,e), plain-text message m is encrypted to cipher-text message c:

$$c=m^e \mod n. \tag{C.4}$$

Decryption: Using private key d, cipher-text message c is decrypted to plain-text message m:

$$m=c^d \mod n. \tag{C.5}$$

RSA-based PSI: The RSA-based private set intersection can protect the privacy of sample ID out of the intersection set with the mechanism of blind RSA signature [45, 69]. We take the two-party setting as an example. Party A contains three user IDs, *i.e.*, $\mathcal{X}_A=\{x_1,x_2,x_3\}$, while party B contains four user IDs, *i.e.*, $\mathcal{X}_B=\{x_1,x_2,x_4,x_5\}$. They want to find their common users via RSA-based intersection:

1. Party A generates RSA keys n,e,d and sends public key (n,e) to party B.
2. Party B blinds and encrypts its user IDs $\mathcal{X}_B$ to $\mathcal{Y}_B=\{H(x)\cdot r^e \mod n \mid x\in\mathcal{X}_B\}$, where r is a unique random number for each x, and sends $\mathcal{Y}_B$ to party A.
3. Party A signs the received $\mathcal{Y}_B$, obtains $\mathcal{Z}_B=\{y^d \mod n = r\cdot H(x)^d \mod n \mid y\in\mathcal{Y}_B\}$ and sends $\mathcal{Z}_B$ to party B.
4. Party A also signs its own user IDs, gets $\mathcal{D}_A=\{H(H(x))^d \mid x\in\mathcal{X}_A\}$ and sends $\mathcal{D}_A$ to party B.
5. Party B unblinds the received $\mathcal{Z}_B$ and obtains $\mathcal{D}_B=\{H(z/r \mod n)=H(H(x))^d \mid z\in\mathcal{Z}_B\}$.

6. Party B computes $\mathcal{D}_A \cap \mathcal{D}_B = \{H(H(x_1))^d, H(H(x_2))^d\}$ and gets common user IDs $\{x_1, x_2\}$.

$H(\cdot)$ denotes the hash function. After party B knows the overlapping users, it could choose whether to inform party A according to different scenarios.

D Modular Exponentiation & Multiplication Algorithm Optimization

D.1 Binary Exponentiation

Modular exponentiation is defined as $P = m^e \mod N$, as shown in Equation 1. In the naïve algorithm, m is multiplied by itself for e times, and the algorithm uses $e-1$ multiplications to obtain the result. Therefore, if e is a large integer, the computation time is dramatic. As a result, to optimize the computation, people usually apply binary exponentiation optimization to reduce the dramatic computation time. Algorithm D.1 shows the process of the binary exponentiation optimization algorithm.

Algorithm D.1 Binary Exponentiation

Input: m, e, N, where $N > 0$
Output: $P = m^e \mod N$

$P = 1$ ▷ Initialization
while $e > 1$ **do**
  if e is odd **then**
    $P = P \cdot m \mod N$
  end if
  $e = e \gg 1$
  $m = m^2 \mod N$
end while
return P

The idea of binary exponentiation is to reduce the number of multiplications by using the binary representation of the exponent e. As a result, we only need to compute at most $2\lfloor \log_2 e \rfloor$ multiplications, which is much smaller than $e-1$. Since the time complexity of modular exponentiation is determined by the number of multiplications, binary exponentiation can reduce its time complexity from $O(e)$ to $O(\log e)$. Worth mentioning, the modulo computation can be performed after each multiplication because of the distribution law in modular arithmetic: $(a \mod N)(b \mod N) \equiv ab \mod N$.

Summary: By using the binary exponentiation optimization algorithm, we can largely optimize the time complexity of modular exponentiation computation.

D.2 Montgomery Modular Multiplication

After applying the binary exponentiation optimization algorithm as shown in §D.1, we lower the time complexity of modular exponentiation computation by reducing the number of multiplications. However, after each multiplication, we have to perform one modulo operation. Although we can implement modulo operation on hardware with Cyclic Reduction and Barrett Reduction algorithms [32], the performance of these algorithms is still not satisfying because of the division operations used in these algorithms. Therefore, FLASH utilizes Montgomery Modular Multiplication [65] to replace the modulo operation with a bit-shifting operation, which is more hardware-friendly.

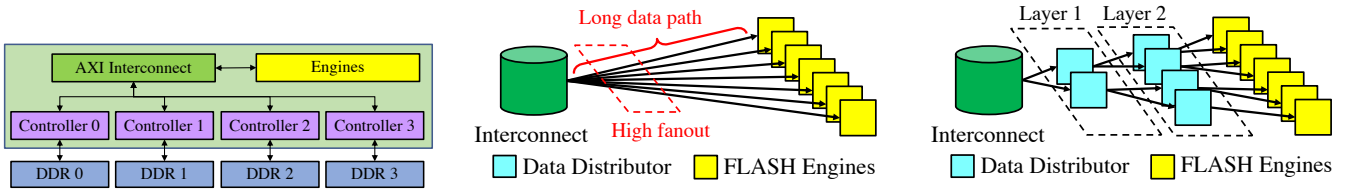
Algorithm D.2 Montgomery Modular Multiplication

▷ Given three input numbers X, Y and N, the Montgomery Modular Multiplication outputs $Z = X \cdot Y \cdot R^{-1} \mod N$, where R is a power of 2 and $\lfloor \log_2 R \rfloor = \lfloor \log_2 N \rfloor$.
Input: $X = (X_{d-1}, ..., X_0)$, $Y = (Y_{d-1}, ..., Y_0)$, $N = (N_{d-1}, ..., N_0)$, N', where
  $N' = (-N)^{-1} \mod r$, ▷ N' is pre-computed in S1
  $r = 2^w$, $d = \lfloor \log_r N \rfloor + 1$, ▷ r and d is used to split data
  $\gcd(N, r) = 1$, with $N \times N' \equiv -1 \mod r$
Output: $Z = \text{ModMult}(X, Y, N) = X \times Y \times R^{-1} \mod N$

$Z = (Z_{d-1}, ..., Z_0) = 0$ ▷ Initialization
for all $i = 0, 1, ..., d-1$ **do** ▷ Loop on Y
  $q = (Z_0 + X_0 \times Y_i) \times N' \mod r$
  $C = 0$
  for all $j = 0, 2, ..., d-1$ **do** ▷ Loop on X
    $S = Z_j + X_j \times Y_i + q \times N_j + C$
    if $j > 0$ **then**
      $Z_{j-1} = S \mod r$
    end if
    $C = S \gg w$ ▷ Carry higher bits
  end for
  $Z_{d-1} = C$
end for
if $Z \geq N$ **then**
  $Z = Z - N$
end if
return Z

The process of applying Montgomery Modular Multiplication includes three major steps: (1) converting the data into Montgomery space, (2) computing the modular multiplication in the Montgomery space, and (3) converting the data back from Montgomery space. Before going into details, we will first describe Algorithm D.2. This algorithm implements efficient $A * B * R^{-1} \mod N$ in a hardware-friendly way. The key optimization of the algorithm is the introduction of the divider $R = r^d$. Thus the division can be easily implemented by bit-shifting since r is a power-of-2 integer, and after the division, the result is an integer within $[0, 2N)$ and no more modulo operation is needed. Afterward, we will show details of each step in the following sections.

Converting the data into Montgomery space: Before applying Montgomery Modular Multiplication, the input numbers should be converted to Montgomery space. The conversion formula is $A = a \cdot R \mod N$. It can also be written as $A = a \cdot R^2 \cdot R^{-1} \mod N$, so we can leverage Algorithm D.2 to efficiently calculate it. In the formula, a is one of the multiplicands of modular multiplication. A is the Montgomery space of a. N is the modulus. R is a power of 2 and it satisfies the condition that $\lfloor \log_2 R \rfloor = \lfloor \log_2 N \rfloor$.



(a) AXI Interconnect. (b) Naïve data distribution. (c) Hierarchical data distribution.

Figure D.1: FLASH adopts hierarchical data distribution to enable efficient data exchange between on-chip and off-chip memory.

Computing the modular multiplication in the Montgomery space: We can directly use Algorithm D.2 to efficiently calculate the modular multiplication.

Converting the data back from Montgomery space: To convert a number out of Montgomery space, the conversion formula is $p = P \cdot R^{-1} \mod N$. p is the result of modular multiplication. P is the Montgomery format of p. Similarly, we can leverage Algorithm D.2 to complete the computation.

Parameter Computation: In the above steps, we notice that if N, which is usually the public key in cryptosystems, remains unchanged, $R^2 \mod N$ (*e.g.*, used in converting the data into Montgomery space) and $(-N)^{-1} \mod r$ (*e.g.*, line 3 and 8 in Algorithm D.2) remain constant values. We call these constant values parameters in this paper, and we show that we can compute these parameters in advance and avoid duplicate calculations to improve the performance further.

Summary: By applying the Montgomery Modular Multiplication, we mainly replace the modulo operation with a hardware-friendly bit-shifting operation, which improves the performance of modular multiplication/exponentiation on hardware.

E The Montgomery Modular Multiplication Circuit Design

In this section, we will describe how our Montgomery Modular Multiplication Circuit works (shown in Figure 5).

According to Algorithm 1, the outer loop iterates over operand Y. Therefore, the circuit sequentially reads different Y_i from RAM Y and performs execution over them. The workflow of our Montgomery Modular Multiplication circuit contains four steps. Steps 1 and 2 in the following introduction focus on the workflow for a fixed Y_i while steps 3 and 4 show how we bridge the operations between iterations with different i and obtain the final result. We use Reg to represent the register group.

1. X_0 and Y_i are sent to Mul 1 to get a $2w$-bit multiplication result. The higher w bits of the result are cached in Reg r_1. The lower w bits, denoted as α in Algorithm 1, are cached in both Reg α and r_7. The numbers stored in Reg α are used for the calculation of β via Add 5. With β and N' available, their multiplication result q can be obtained from the output of Mul 2. After that, q is sent back to the input of Mul 2 and multiplied with N_0. Similarly, the higher w bits of the multiplication result are cached in Reg r_3 and the lower w bits are sent to Reg r_8. Combining data cached in r_7 and r_8, it is straightforward to get δ_1 and δ_2 with several addition units. The results of addition, Z_0 and C, are cached in Reg Z_0 and Reg C for subsequent operations.
2. Following step 1, the inner loop for j begins. Different iterations of the inner loop are fully pipelined, which implies the jth iteration is executed by the circuit just one cycle after the $(j-1)$th iteration. At the beginning of the pipeline, Mul 1 and Mul 2 simultaneously calculate $X_j \times Y_i$ and $q \times N_j$. The higher w bits and lower w bits of the results are separately cached in different registers. Please note that we use Reg r_5 and r_6 to register the higher w bits in the circuit for one more cycle compared to the lower w bits so that δ_1 can be calculated through the addition between higher w bits from the $(j-1)$th iteration and lower w bits from the jth iteration (*i.e.*, line 12 in Algorithm 1). After the calculation of δ_1, the subsequent calculation of δ_2 can also be simply executed by the adders in the pipeline. The intermediate results, Z_{j-1} and C, are cached in RAM Z and Reg C.
3. We begin the execution of the $(i+1)$th iteration of the outer loop when all the multiplications in the ith iteration accomplish. Although some operations like addition are still in progress, the multipliers are free to start the multiplications in Step 1 for the $(i+1)$th iteration.
4. After accomplishing the outer loop, the result Z should be stored in RAM Z. If $Z < N$, we directly output Z. Otherwise, we calculate $Z - N$ as the final output.

F Hierarchical Data Distribution

While the design in §4.2.3 is efficient, it also introduces a practical problem. As shown in Figure D.1a, FLASH adopts AXI interconnect to manipulate the external memory [68, 72]. However, as the on-chip memory units are placed near each engine for low latency, naïvely distributing data from the AXI interconnect to these memory units, as shown in Figure D.1b, leads to high fan-out near interconnect and long data paths. These two issues will cause (1) large design difficulties for circuits placement because there are too many long paths to be placed near interconnect; (2) degraded performance because long paths cause large delay for the circuits.

To solve the problem, we design a hierarchical data distribution mechanism as shown in Figure D.1c. Instead of directly sending data to all engines, FLASH distributes data

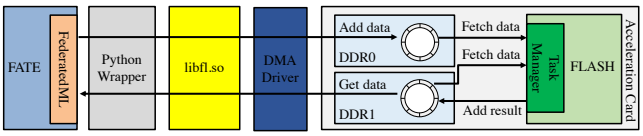


Figure G.2: FLASH integrates with cross-silo FL frameworks by providing an integrated software package.

at multiple layers. At each layer, the data distributors receive data from the previous layer and further distribute data to the data distributors/engines in the next layer. Suppose we have m engines and n layers, the fan-out of each data distributor is $\sim \log_n m$, which is much smaller than m. As a result, FLASH achieves a much smaller fan-out and shortened logical data path. These two advantages first reduce the design complexity because a small fan-out with short logical paths will make the circuits' placement much easier. Furthermore, they also improve performance because they allow a high operating frequency by restricting the delay of all logic paths. In our FPGA implementation, the delay of all logic data paths is within 3.3ns, thus we can achieve a high FPGA operation frequency of 300MHz [20].

G Software Stack Architecture

Figure G.2 illustrates how FLASH integrates with the cross-silo FL software. As introduced in §4.4, FLASH's software stack contains Xilinx DMA Driver [21], `libfl.so` library and its corresponding Python wrapper.

On Modular Learning of Distributed Systems for Predicting End-to-End Latency

Chieh-Jan Mike Liang[‡] Zilin Fang[*] Yuqing Xie[°] Fan Yang[‡]
Zhao Lucis Li[⋆] Li Lyna Zhang[‡] Mao Yang[‡] Lidong Zhou[‡]
[‡]*Microsoft Research* [*]*CMU* [°]*Tsinghua University* [⋆]*University of Science and Technology of China*

Abstract

An emerging trend in cloud deployments is to adopt machine learning (ML) models to characterize end-to-end system performance. Despite early success, such methods can incur significant costs when adapting to the *deployment dynamics* of distributed systems like service scaling-out and replacement. They require hours or even days for data collection and model training, otherwise models may drift to result in unacceptable inaccuracy. This problem arises from the practice of modeling the entire system with *monolithic* models. We propose Fluxion, a framework to model end-to-end system latency with *modularized learning*. Fluxion introduces learning assignment, a new abstraction that allows modeling individual sub-components. With a consistent interface, multiple learning assignments can then be dynamically composed into an *inference graph*, to model a complex distributed system on the fly. Changes in a system sub-component only involve updating the corresponding learning assignment, thus significantly reducing costs. Using three systems with up to 142 microservices on a 100-VM cluster, Fluxion shows a performance modeling MAE (mean absolute error) up to 68.41% lower than monolithic models. In turn, this lower MAE allows better system performance tuning, e.g., a speed up for 90-percentile end-to-end latency by up to 1.57×. All these are achieved under various system deployment dynamics.

1 Introduction

Predicting cloud system performance is critical for improving the end-user experience. While this problem has been traditionally addressed with analysis and handcrafted performance models, an emerging trend is to incorporate machine learning (ML) techniques for performance modeling [7,9,23]. Such approaches [2,3,6,13,14,20,21,34,41] typically use *monolithic* ML models, to predict the performance (e.g., user request latency), given configurable knobs of the system component (e.g., cache size) and observable states (e.g., request rate).

In recent years, the microservice architecture has gained popularity in building distributed cloud systems [1,12,18,25,40]. Its per-service flexibility enables continuous integration and continuous delivery (CI/CD), and per-service horizontal scaling (e.g., replication) and vertical scaling (e.g., capacity adjustments) can handle load dynamics. Interestingly, the monolithic approach of modeling the entire system could still be applicable to such distributed systems, and doing so frees operators from explicitly modeling service dependencies.

Unfortunately, our first-hand experience at Microsoft suggests inherent limitations in effectively learning distributed system's *end-to-end* performance. There is a need to continually adapt performance models to the *deployment dynamics*. As services are independently scaled and replaced over time, deployment updates become frequent operations.

Continually updating monolithic models can incur significant time costs, especially if deployment dynamics are handled in an ad-hoc way. First, collecting a sufficient amount of training data can be time-consuming, as new system dynamics can take minutes and even hours to be fully warmed-up and stable [39]. Second, designing and training a new model can also be time-consuming, even with the help of automation tools [4,19,27]. For example, a system evaluated in §5 has 142 microservices, and requires a performance model that considers 1,034 service knobs and states. Collecting sufficient data points to train such a complex monolithic model takes us ∼46 hours, and model training takes additional ∼24 hours. Such a practice hinders the practicality of monolithic approach, for performance tuning in the real world.

Given the above challenges, we advocate *modularized learning* for microservice-based distributed systems. Our key observation is the locality of deployment dynamics, where changes happen at the granularity of system components (e.g., microservices). So, modularized learning models the performance of each service individually, and carefully composes these models on-demand to follow deployment dynamics.

Fluxion is a framework that realizes modularized learning

This work was done when Zilin Fang, Yuqing Xie, and Zhao Lucis Li were interns at Microsoft Research.

to model end-to-end latency while handling system deployment dynamics efficiently and effectively. Fluxion introduces *learning assignment*, an abstraction to model each service in a distributed system. Learning assignment is instantiated on each service instance to model its performance metrics (e.g., latency). It can accommodate any service and different ML modeling techniques (e.g., DNN or Gaussian process). Learning assignments can be composed into an *inference graph* to model a large complex system, similar to how services are composed into an end-to-end system. Changes in a service of the system only induce modeling errors in the corresponding learning assignments. Since the configurations and internal logic of other services remain the same, their corresponding learning assignments along with their modeling accuracy also remain unchanged. Therefore, Fluxion only needs to update the learning assignments corresponding to the changed service, thus significantly reducing the costs.

Three unique characteristics enable learning assignment to handle system deployment dynamics effectively. First, to capture the impact from other services, learning assignment defines external performance dependencies as part of modeling inputs. This enables the composability — multiple assignments can be composed into an inference graph, to follow service dependencies of a complex distributed system on the fly. Second, instead of considering only the performance metric of interest (e.g., p90 latency), learning assignments can take in a spectrum of performance metrics from upstream assignments (e.g., p50–p99 latencies) in inference graph. This allows a learning assignment to better observe (and capture) the impacts of system deployment dynamics from upstream assignments. Finally, to capture the temporal system dynamics, a learning assignment can host one or more ML models trained in different time periods and scales.

In summary, this paper makes the following contributions. ***(1)*** We propose a modular approach to modeling end-to-end latency for complex and dynamic distributed systems, exemplified by microservice systems. ***(2)*** The abstraction of learning assignment and the resulting inference graph effectively capture intrinsic system dynamics, as well as dependencies among services. ***(3)*** We conduct comprehensive experiments to demonstrate the significantly superior performance of Fluxion over existing approaches, under various system deployment dynamics. In some microservice systems spanning 100 VMs, Fluxion's performance model exhibits up to 68% lower MAE (mean absolute error). In turn, this enables better system performance optimization, or p90 latency speed up by up to 1.57× over the use of baselines. At the same time, Fluxion reduces the model training time by up to 99.98%.

2 Background and Motivations

2.1 Performance Prediction and Modeling

Performance models predict the *end-to-end system performance* (e.g., user request latencies), given *observable states* (e.g., per-service request rates) and *configurable knobs* (e.g., per-service cache size and thresholds). Performance models can be analytically constructed with mathematical formulations, but doing so does not scale well with the size or complexity of large-scale distributed systems. A typical microservice system can have hundreds (and even thousands [40]) of services, and requests can traverse 40+ services [25].

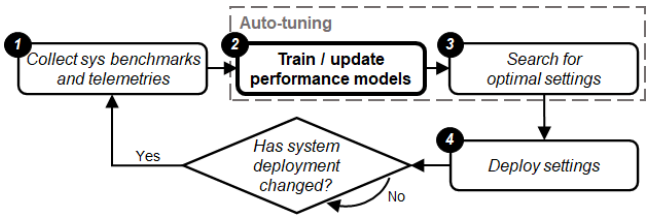


Figure 1: General workflow of auto-tuning. Fluxion focuses on *step #2*, or the efficiency in adapting performance models to system deployment dynamics.

Learned performance model. Advances in machine learning (ML) enable performance modeling to be learned, with regression techniques such as Gaussian process and DNN. Like previous efforts that model monolithic systems [2,3,9,13,21,23], it is possible to treat an entire microservice system as a black-box. To match system deployment scale, ML models can monolithically grow in size (e.g., adding DNN neurons). Furthermore, black-box modeling eliminates the need to explicitly consider service interactions and dependencies.

For training performance models, each training data point consists of inputs (i.e., per-service knob settings and observable states) and an output (i.e., a system performance measurement). Data are collected through benchmarks in controlled environment (e.g., testbeds or isolated sections in production), or telemetries in production. Trained models may be evaluated with testing data, which are collected in the same fashion. One common evaluation metric for performance modeling is MAE (mean absolute error) [37], or the average magnitude of errors in a set of test predictions made by the model.

Performance auto-tuning scenario. As Fig 1 illustrates, performance models can drive auto-tuning [2, 3, 21–23]. The goal is to guide non-system-experts to set system knobs, to optimize for a performance metric. Auto-tuning relies on optimizers (*step #3*), which iteratively search for global optimum in the modeled space. Each iteration algorithmically selects new knob setting (for performance model to predict), based on performance predictions from previous iterations.

2.2 High Costs to Maintain ML Models

We observe significant time costs in continually keeping ML-based performance models updated to system deployment dynamics. Deployment dynamics make performance models drift over time and impact modeling accuracy, hence auto-tuning outcomes (c.f. §5). Main sources of deployment

dynamics include ***(1)*** system scaling, i.e., replicating or reclaiming service instances, and ***(2)*** continuous integration and delivery (CI/CD), i.e., replacing existing services.

We discuss the breakdown of time costs below.

***Problem #1*: Collecting training data points for performance modeling can be time-consuming.** As the model complexity grows, the number of training data points necessary also grows. However, each benchmark requires the system to be fully warmed-up and stable [39]. In our cases, one benchmark can take up to 15 minutes, and collecting sufficient data points can require ∼46 hours.

The problem exacerbates as distributed cloud systems require a significantly higher model complexity than previously considered. Unlike monolithic systems with ∼20 knobs and states to consider, this number can quickly add up to hundreds and thousands for microservice-based systems. For example, Train-Ticket [31] has 41 services. At initialization, it has 242 model inputs: 148 configuration knobs (e.g., MongoDB's `eviction_dirty_target`), 94 states (e.g., Docker's `cpu-limit` and per-service requests per second). As Train-Ticket scales-out by a factor of 6, there is a total of 210 service instances, and 1,020 model inputs.

***Problem #2*: Designing and training new models for performance modeling can be time-consuming.** Keeping performance models updated goes beyond simply fine-tuning models with recent benchmarks. In many cases, the required changes lie in the model structure. One motivating example is how replicating services essentially alters the deployment, with respect to the available knobs, service states, and service dependencies. As a result, we need a new model of different input dimension and even different modeling technique.

Although AutoML toolkits can automate this process to some extent, our experience suggests that it can take at least 20 hours to produce a reasonably accurate ML model for performance modeling. Furthermore, it is not feasible to pre-train all monolithic models for all possible deployment setups. Since services can be independently replaced and arbitrarily scaled, the number of possible deployment setups is unbounded.

2.3 Modularized Learning

The principle of modularity has been proven in engineering scalable and elastic systems. It provides opportunities to realize ML-based performance modeling in an agile and accurate way. Instead of monolithically modeling the end-to-end latency of distributed systems with one performance model, we propose *modularized learning* that breaks down this model to align with a deployment's modular units.

Challenges. To practice modularity, it is natural to independently model each system component, e.g., a microservice. Given system components can have vastly different configuration knobs and states, different types of ML models can be chosen for individual system components. The key challenges are: ***(1)*** to represent different system components, possibly using different ML techniques, with a *consistent interface*; and ***(2)*** to have a *composability criteria* that combines these component representations into an end-to-end ML model for performance modeling.

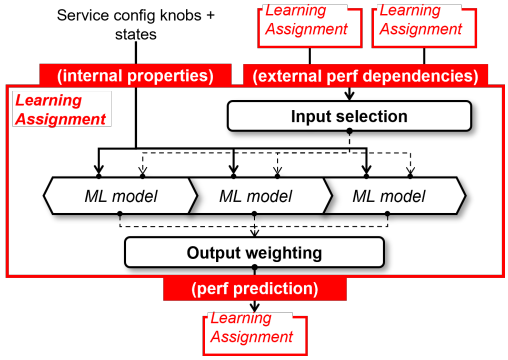


Figure 2: Learning assignment is a wrapper for ML models. It has a consistent interface for composability.

3 Fluxion Framework

Fluxion is a framework that realizes modularized learning for distributed systems. To address the two challenges discussed in §2.3, Fluxion introduces *learning assignment* to abstract away model and component heterogeneity and provide a unified interface to model service-level latency (§3.1). Moreover, Fluxion presents *inference graph* to dynamically compose assignments into an end-to-end performance model (§3.2).

3.1 Learning Assignments

A learning assignment is a basic modeling unit. It hosts one or more ML models that collectively model a modular unit in distributed systems. Since system deployment dynamics typically happen at the unit of services (e.g., scaling and replacing services), assignments are instantiated on a *per-service-instance* and *per-performance-metric*[1] basis. When deployment dynamics happen, this mapping of modular units allows Fluxion to localize changes to some learning assignments. Fig 2 illustrates the internal structure of an assignment, and we elaborate the details next.

Interface. The learning assignment interface is designed to abstract service-level performance for ML models. Modeling individual services is different from modeling the entire system monolithically. The former needs to take service dependencies into account. E.g., a service's observed latency inherently includes the latency of its downstream services [10].

Therefore, in addition to internal configuration knobs and internal observable states, the assignment inputs further include external performance dependencies (e.g., downstream service latency). The assignment output is a service-level

[1] E.g., p50 and p90 latencies require two learning assignments.

performance metric. This interface addresses *challenge #1* — the consistent service-level performance metrics as input and output, together with the heterogeneous internal states and knobs, are sufficiently general to model different services and to host ML models of different modeling techniques, and a learning assignment's output can be connected to another assignment as an external performance dependency.

Exposing a spectrum of performance metrics. To predict the end-to-end latency more accurately, a learning assignment may expect the external performance dependencies to be a spectrum of performance metrics from dependent services. For example, to predict end-to-end p90 latency, a learning assignment may require p50–p90 latencies of the downstream services (and even their CPU utilization and disk throughput), so as to decide which metrics are appropriate for consideration. This allows a learning assignment to better observe the extent of impacts that system deployment dynamics impose on its downstream services.

However, not all performance metrics are *highly* relevant to the one being predicted. An example is the bottom-percentile latencies *vs.* the top-percentile latencies. Including irrelevant performance metrics incurs additional costs. The first is the training costs. As unnecessary inputs add noises to the training dataset, some ML models would need more data points to distinguish and learn from the relevant inputs. And collecting training data points can be time-consuming (c.f. §2.2). The second is the unnecessary learning assignments introduced to predict these irrelevant metrics.

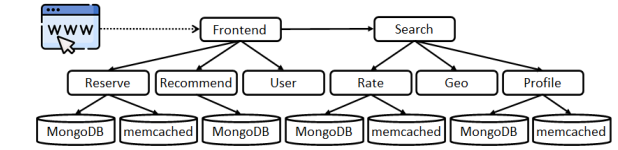
To this end, learning assignment introduces "input selection" (shown in Fig 2) to prune unnecessary metrics and the corresponding learning assignments. The problem of input selection can be formulated as follows. Given a set of k learning assignments (m_k) as inputs to a specific learning assignment, we want to find the k-dimensional binary weight vector (w_k^*). w_k^* should minimize the prediction error of an weighted sum of m_k, over a batch of n data points (inputs X, and outputs Y). This is formulated as the following equation:

$$w_k^* = \underset{w_k \in W}{\operatorname{argmin}} \left(\sum_{i=1}^{n} Y_i - f(w_k, m_k(X_i))\right)^2. \quad (1)$$

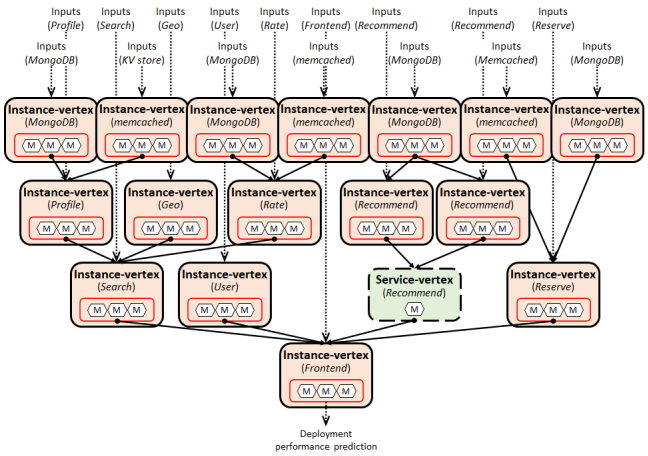
The goal is to search for w_k^*, or the optimal k-dimensional binary vector. f is a predefined function that aggregate outputs of k learning assignments, and it can be hand-written code or non-linear functions such as neural networks. We can expand f as follows:

$$f(w, m(X)) = \frac{\sum_{i=1}^{k} w^{(i)} m^{(i)}(X)}{\sum_{i=1}^{k} w^{(i)}}. \quad (2)$$

In practice, k can be large. For example, RocksDB has $k = 99$ performance metrics. To select three relevant metrics need to search 941,094 possible combinations. To find a solution effectively, §4.1 presents one generic approach in the current implementation of Fluxion.



(a) Execution graph



(b) Inference graph

Figure 3: A simplified inference graph of a microsevice application, Hotel Reservation. Vertices represent services, and directed edges capture performance dependencies among services. Graph inputs include configuration knobs and observable service states. (M) represents models. In this example, the Recommend service is scaled out to two instances.

Capturing system's temporal dynamics. A learning assignment can host multiple ML models trained in different time periods and scales. Doing so promotes the reuse of previously learned models, in order to better capture temporal dynamics and predict recurring patterns. An example is the daily variations in incoming request rates. In this case, we can train a new model with data points collected each day.

Learning assignment introduces "output weighting" (shown in Fig 2) to weight over outputs from different time periods/scales and reduce all models' outputs to one succinct value. The formulation of output weighting is similar to that of input selection, except that k assignments are changed to k internal models and w_k^* here is a k-dimensional vector of continuous numbers between 0 and 1 inclusively. And the output is computed as the weighted sum of all k models' outputs.

Similarly, to find a good combination of weights is computationally expensive. §4.1 presents one generic approach in the current implementation of Fluxion.

3.2 Inference Graph of Learning Assignments

Inference graph is a set of interconnected learning assignments. It represents the performance modeling of an end-to-

end system. From external user's perspective, inference graph has the same input/output semantics as monolithic models. In other words, inference graph exposes the following performance modeling inputs: all services' configuration knobs and observable states. Its output predicts for an end-to-end performance metric: the end-to-end tail latency in our case. Inference graph addresses *challenge #2*, i.e., the composability criteria to compose learning assignments.

3.2.1 Inference Graph Construction

Fig 3 illustrates a simplified inference graph for a microservice system, Hotel Reservation [11].

Graph vertices. An inference graph has two types of vertices: instance-vertices and service-vertices (c.f. Fig 3b). Instance-vertices correspond to the learning assignments modeling the performance of service instances. They expose learning assignment inputs (e.g., service configuration knobs, observable states, and external performance dependencies) and output (e.g., a service-level performance metric). Service-vertices aggregate instance-vertices that model the same service and performance metric. In Fig 3b, the Recommend service (marked in dash-line) is scaled out to two instances.

Graph edges. Graph edges are directed dataflows, and they satisfy services' external performance dependencies. To draw the edges correctly, we leverage the observation that the performance dependency of two services is the *reverse* of their execution dependency. Since an upstream service invokes RPC calls to its downstream service, the upstream service's latency would depend on the latency of downstream service. Hence an edge should be connected from the learning assignment (i.e., vertex) representing the downstream service to the one representing the upstream service (i.e., the reverse direction of the service execution order).

Handling deployment dynamics through inference graph updating. Since graph vertices and edges have a strong correspondence to services in the system deployment, orchestration-induced dynamics can be localized to certain regions of the inference graph. This implies that other regions can remain unchanged. Fluxion provides APIs to update graph for common orchestration operations (c.f. §4.1.2). First, scaling-out a service is conceptually equivalent to replicating the corresponding instance-vertices, to match the number of deployed instances. Similarly, scaling-in a service removes some of the corresponding instance-vertices. Second, upgrading a service (or even a migration from MySQL to PostgreSQL) is conceptually equivalent to replacing old service's instance-vertices with new service's.

3.2.2 Graph Inferencing to Predict End-to-End Latency

Graph inferencing is performed through graph traversal. The traversal starts from graph vertices that do not have external performance dependencies, or services that do not invoke any downstream services (e.g., the top vertices in Fig 3b). At each vertex, the learning assignment output metric is computed with its ML models. Following graph edges, the output is then passed to subsequent vertices as an external performance dependency. The traversal stops at the last vertex in the graph, or typically the gateway service in a deployment. The output of this last vertex is the output of the graph, and it predicts the end-to-end system performance.

3.2.3 Inference Graph Re-training

Inference can be performed immediately after graph is composed, but in cases where the prediction error rate is high, re-training can mitigate the problem. Fluxion identifies two major error sources. And, it can effectively reduce the re-training costs by taking advantage of the inherent modularity in the graph, rather than re-designing and re-training the entire monolithic model. Particularly, graph prediction errors can be traced back to some subsets of vertices.

Graph error source #1: New learning assignments. New learning assignments are required when new services are deployed to microservice systems or existing services are being updated. From our experience, their high MAE is typically due to insufficient training, especially by non-ML-experts. This case can be mitigated with the use of AutoML toolkits. Another possibility is service-vertices. Since they aggregate instance-vertices, scaling out/in a service requires them to have a new model with a new input dimension. Localizing new assignments is trivial, and the information is available through recording graph manipulations over time.

Graph error source #2: Unforeseen prediction inputs. Unforeseen prediction inputs can happen when the system receives unforeseen request types, or unexpected request ratios. This is a situation monolithic models also have to handle. As different requests stress different service-to-service execution paths, a service can observe unfamiliar states (e.g., requests per second and CPU utilization) or even downstream service performance. In the monolithic approach, the solution is to retrain the monolithic models. Modularized learning gives new optimization opportunities. Fluxion only needs to retrain the learning assignments being impacted to better handle these prediction inputs.

Identifying the impacted assignments, i.e., vertices in the inference graph, to address *error source #2* can be non-trivial. This step is not simply about identifying vertices whose learning assignments have the largest prediction MAE. A well-trained learning assignment can still output unexpected predictions if it receives erroneous inputs from another. The reason is that local errors of individual assignments can propagate. This is an artifact of how the inference graph output is a function of all its models. As graph traversal passes the prediction of a learning assignment to another assignment,

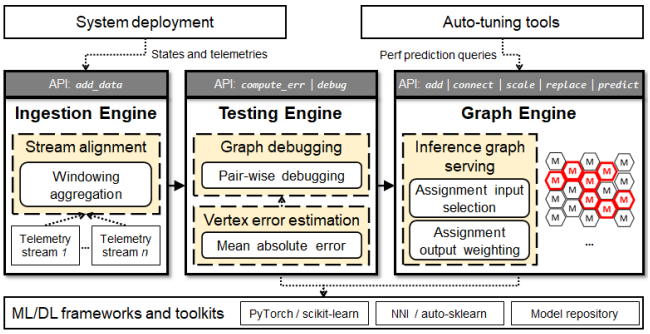


Figure 4: Fluxion architecture overview. It is implemented as three engines. Graph Engine hosts inference graphs. Testing Engine implements features for graph uncertainty estimation and graph debugging. Ingestion Engine ingests and buffers streams of telemetries, for ML model training and testing.

prediction errors propagate and accumulate.

Unfortunately, it is not trivial to analytically derive formulas describing the error accumulation. As ML models apply non-linear transformations to inputs, this non-linearity also transforms input errors to prediction errors. Furthermore, a learning assignment can have multiple model inputs. Not only can different model inputs be weighted differently, but they can also be of predictions from different assignments.

Pair-wise debugging approach. Fluxion implements a pair-wise debugging approach for *error source #2*. This is an iterative process, and each iteration selects one vertex. The core idea is to evaluate the likelihood that the vertex's prediction error is due to its own learning assignment or parent vertices.

System operators provide a test dataset, which contains recent deployment benchmarks. Fluxion then computes the test MAE for each graph vertex. And, it computes the Pearson correlation, or $r(mae_{parent}, mae_{child})$, for each directly connected pair of graph vertices. With all rs computed, the debugging procedure follows a depth-first traversal. It starts from the last graph vertex (i.e., the vertex that produces the graph output), and performs the following steps — ***(step #1)*** with respect to the current vertex, we rank all parent vertices by their r in descending order. ***(step #2)*** If the top-ranked parent vertex has an r larger than 0, we traverse the edge to it and repeat step #1. ***(step #3)*** Otherwise, if the top-ranked parent vertex does not have an r larger than 0, we stop and return the current vertex as the debugging result.

4 Implementation

Current implementation has 13,012 SLOC, supports PyTorch and scikit-learn models, and integrates NNI [27]. Fig 4 shows an overview. A model repository stores models in serialized form and retrieves them with the unique model ID.

4.1 Graph Engine (GE)

GE serves inference graphs. During auto-tuning, the optimizer queries GE for performance predictions, just like how it would query monolithic models. GE implements the input selection strategy and the output weighting strategy (c.f. §3.1), and it offers APIs to manipulate inference graphs.

Input selection strategy. Our implementation is based on Thompson sampling with Beta distribution [30]. For a learning assignment, each external performance dependency has a Beta distribution, to estimate the probability of being selected for w_k^*. The probability density is governed by two variables, α and β. A larger α increases the mean probability, and a larger $(\alpha+\beta)$ decreases the probability variance.

αs and βs are initialized to 0, or system operators can manually specify a larger α to favor certain performance metrics. Then, the input selection strategy repeatedly updates αs and βs as follows. Each round starts by randomly generating a combination of models — specifically, we generate random numbers from each β-distribution and select k performance metrics with the largest number. This combination is then evaluated for the prediction accuracy. If the current round results in a higher accuracy then the first round, we increment each performance metric's α value, otherwise the β value.

This process of updating α and β repeats for a user-defined number of rounds, or if the random selection converges for several rounds. Upon termination, we again generate random numbers from each β-distribution and select k performance metrics with the largest number.

Output weighting strategy. Our implementation is based on differential evolution for stochastic minimization [33]. It makes no assumptions on the search space distribution, and it is easy-to-use due to few hyperparameters. Differential evolution is initialized with a population of starting points in the search space. In rounds of mutation-recombination-selection, it moves towards the optimum.

A subset of the initial population can be based on cached w_k^*s. So, the N initial population $(w_1^0, w_2^0, \ldots w_N^0)$ consists of uniformly random candidates and previously computed w_k^*. At each round G, differential evolution creates mutant vectors storing combinations of individuals (w_i^G, w_j^G, w_k^G) that are randomly chosen from the current population. Mutant vectors are then mixed with a pre-determined population candidate, to produce a new trial candidate. With data points selected by the Ingestion Engine, we then evaluate this trial candidate. If the new trial candidate yields a better prediction accuracy, it is added to the population.

This process of updating population repeats for a user-defined number of rounds, or if the population converges for several rounds. Upon termination, the population candidate that best maximizes Equation 2 is returned.

4.1.1 Learning Assignment APIs

`LA.init(X_names, y_name)` sets the labels for assignment inputs and output.

`LA.add(X, y, del)` adds a new model, and trains the new model with input `X` and output data `y`. `del` specifies whether existing models should be deleted.

`LA.input_selection(X, y)` runs the input selection strategy with the inputs `X` and output `y`.

`LA.output_weighting(X, y)` runs the output weighting strategy with the inputs `X` and output `y`.

`LA.predict(X)` predicts for `X`.

4.1.2 Graph Update APIs

`GE.add(s_name, p_name, a_ptr)` adds a new instance-vertex in the graph, to represent one instance of the service `s_name` for the performance metric `p_name`. `a_ptr` points to the learning assignment instance.

`GE.connect(s1_name, s2_name)` specifies the performance dependency from the service `s1_name` to `s2_name`. Since a service can have multiple performance metrics, `GE.connect` adds directed edges for all combinations of `s1_name`'s and `s2_name`'s metrics. It matches learning assignments' input and output labels to make the connection.

`GE.scale(s_name, num_inst)` replicates/removes instance-vertices to match `num_inst`, for all `s_name`'s performance metrics. And, for each performance metric, it adds a service-vertex to aggregate instance-vertices. These service-vertices are initialized to an input dimension of `num_inst`, and they can be trained by invoking `GE.fit`.

`GE.replace(s_name, p_name, a_ptr)` updates instance vertices to reference the new learning assignment `a_ptr`. Service-vertice needs to be initialized by `GE.fit`.

`GE.fit(s_name, p_name, time_window)` creates a learning assignment, for the service-vertex of `s_name` service and `p_name` metric. Then, it retrieves data points within the last `time_window` seconds from IE, and invokes `LA.add`.

4.2 Testing Engine (TE)

TE implements functionalities to support graph testing. First, `TE.compute_err` computes the per-vertex test error, with the test dataset given in the argument. The current implementation uses the mean absolute error. The test dataset is in a tabular format; each row represents one system benchmark, and columns record service config knob settings and performance measurements. Second, `TE.debug` starts the graph debugging strategy and returns a learning assignment's name.

4.3 Ingestion Engine (IE)

IE ingests and buffers per-service telemetry streams for ML model training and testing. A stream contains one time-series data type, which can be a performance metric, a configuration knob, or an observable state. Data are published to IE by `IE.add_data`. They are in JSON format with the following fields: `stream_uri`, `type`, `seq_num`, and `val`. The `type` field can be "continuous", "discrete", or "choices". The `seq_num` field is an incrementing integer such as the Unix timestamp.

5 Evaluation

We evaluate and demonstrate the superior performance of Fluxion, with three complex microservice systems on up to 100 VMs, under deployment dynamics like service scale-out and replacement. Our major results include:

(1) Fluxion consistently maintains a lower performance modeling MAE (mean absolute error). Considering the case of gradually scaling Hotel Reservation from 15 to 142 services, Fluxion's MAE is 29.14% lower on average and up to 68.41% lower than comparison baselines.

(2) Fluxion's lower MAE enables better end-to-end system latency. Considering the case of switching from baselines to Fluxion, auto-tuning optimizers achieve a speedup of 1.24× on average (and up to 1.44×), for TrainTicket's 90th-percentile latency.

(3) Using a 30-day Azure trace, results show that Fluxion can capture system dynamics in the temporal dimension. By recognizing and adapting to the recurring patterns, the daily training time is reduced by up to 99.98%.

5.1 Microservice Systems

We evaluate the effectiveness of Fluxion, by measuring both the performance modeling accuracy improvement (or MAE *reduction*), and the resulting latency improvement for microservice-based systems. Our evaluations are based on case studies — as microservice systems are orchestrated to exhibit deployment dynamics, we run auto-tuning to continually optimize their tail latency, i.e., the 90th-percentile latency.

Microservice system setup. We deploy three systems: ***(1)*** TrainTicket [31], with 41 unique services, ***(2)*** Hotel Reservation, with 15 unique services from DeathStarBench [11], ***(3)*** Boutique, with 11 unique services from Google [15]. Services are managed by Kubernetes, and they can be replicated and replaced. KubeDNS is used for round-robin load-balancing. Services log per-request latencies for all remote procedure calls, and measurements are centrally stored in an InfluxDB. Appendix lists each system's knobs. For databases, we select top knobs that have been identified to impact read/write latency in production, by Microsoft engineers.

Experiment setup. Our comparison baselines are performance models of monolithic Gaussian process (GP) and multilayer perceptron (DNN) models. These baselines are common in recent performance optimization efforts [2, 3, 6, 7, 9, 13, 14, 20, 21, 23, 34, 41]. GP uses Matern(5/2) kernel [3]. We

use NNI [27] to tune DNN hyper-parameters: number of hidden layers, hidden layer size, and initial learning rate. We construct Fluxion graphs with APIs in §4.1.1, and learning assignments use GP. Externally, baselines and Fluxion graph have the same inputs and output (c.f. Appendix).

Our testbeds are 3 clusters on Azure — 100-VM cluster (with Intel E5-2673 CPU and 54GB RAM) for Train Ticket and Hotel Reservation; 6-VM cluster (with Intel 8272CL CPU and 8GB RAM) for Hotel Reservation; a 9-VM cluster (with Intel 8171M CPU and 16GB RAM) for Boutique.

Methodology. We send workloads of requests, with wrk2 [38] and Locust [24] (c.f. Appendix). We periodically induce the following stresses to trigger orchestrations, hence deployment dynamics. First, we change the requests per second (RPS) or ratio of request types, to stress different services and paths. This stress then triggers Kubernetes' HPA (Horizontal Pod Autoscaler) to replicate or reclaim multiple services, to maintain an average service CPU utilization of 60%. Second, we replace a service. Third, we scale-out the entire system.

After each orchestration operation (i.e., deployment dynamics), we first ensure all modeling approaches' inputs match system knobs. This step involves training new baselines, and also re-composing Fluxion's graph. In addition, to evaluate how different approaches would improve with further training, we collect new training dataset. Each iteration performs one random benchmark and measures per-request latencies for ∼10 minutes. To compare prediction MAE (mean absolute error), we collect an additional 100 random benchmarks as the testing dataset. MAE is computed as the average error between a benchmark's actual latency and predicted latency.

5.1.1 Performance Modeling Error Reduction

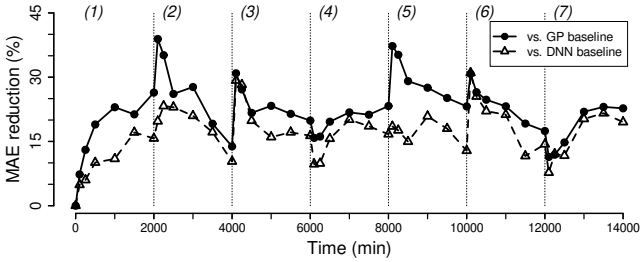
We evaluate how well Fluxion reduces the MAE of predicting the end-to-end latency, as compared to baselines. In the presence of deployment dynamics, a consistently lower MAE suggests a more robust performance modeling.

For TrainTicket, Fig 5b shows that Fluxion consistently maintains a lower MAE (i.e., MAE reduction is always greater than 0) for predicting 90th-percentile latency. It achieves 7.30–38.92% and 4.88%–29.22% lower MAE than monolithic GP and DNN baselines, respectively; this translates to an average MAE reduction of 2,181.89 μs and 1,547.27 μs. Similarly, for Hotel Reservation on the 100-VM cluster, Fig 7b shows that Fluxion achieves 34.82–60.05% and 27.24–57.39% lower MAE than monolithic GP and DNN baselines, respectively; this translates to an average MAE reduction of 7,298.78 μs and 6,858.30 μs. For Hotel Reservation on the 6-VM cluster, Fig 6b shows that Fluxion achieves 10.04–68.41% and 10.87–66.14% lower MAE than baselines; this translates to an average MAE reduction of 2,814.09 μs and 2,205.70 μs. Finally, Fig 8b shows Fluxion's lower MAE, for Boutique.

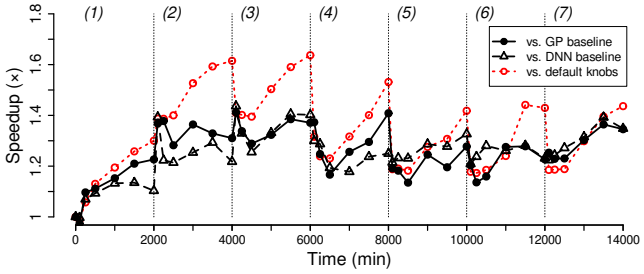
Right after deployment dynamics happen (i.e., orchestration operations), the MAEs of all approaches increase. However, compared to baselines, the *relative* MAE reduction of Fluxion becomes significantly higher after an orchestration operation. Since monolithic baselines require entirely new models, their modeling accuracy can improve only after collecting sufficient data points and training. For example, as we gradually scale-out Hotel Reservation from 15 to 142 services, we need new GP and DNN baselines to accommodate the input dimension that grows from 63 to 1,034. Furthermore, Table 6a shows an operation (Step #6) that replaces all Memcached services by Redis. Although the input dimension here does not change, we need new baselines because Redis

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=50)	Deploy "TrainTicket"	49	242
(2) Stress RPS (=100)	Scale-out *Station* (2×)	50	247
(3) Change req ratio	Scale-out *Station* (2×), *Route* (2×)	51	252
(4) Change req ratio	Scale-out *Station* (2×), *Route* (2×), *Order* (2×)	53	262
(5) Stress RPS (=250)	Scale-out *Station* (5×), *Route* (6×), *Order* (4×)	62	307
(6) Stress scale	Scale-out *all* services (3×)	105	510
(7) Replace services	*TiDB* replaces *MySQL*	49	242

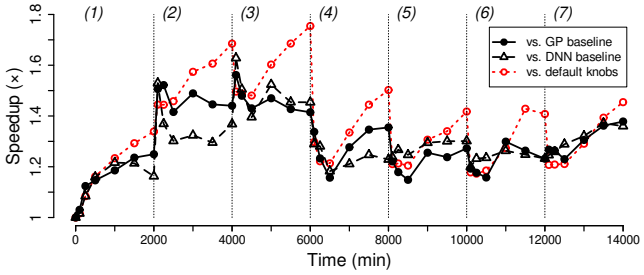
(a) Microservice orchestrations



(b) Perf modeling error reduction for predicting p90 latency



(c) System p90 latency speedup (w/ random optimizer)



(d) System p90 latency speedup (w/ Metis optimizer)

Figure 5: TrainTicket on 100-VM cluster (Intel 2673).

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=50)	Deploy "Hotel Reservation"	15	63
(2) Stress RPS (=500)	Scale-out *Reservation* (2×)	17	72
(3) Change req ratio	Scale-out *Reservation* (2×), *Rate* (2×)	19	80
(4) Stress RPS (=800), change req ratio	Scale-out *all* services (3×)	37	139
(5) Stress scale	Scale-out *all* services (4×)	48	177
(6) Replace services	*Redis* replaces *Memcached*	15	63

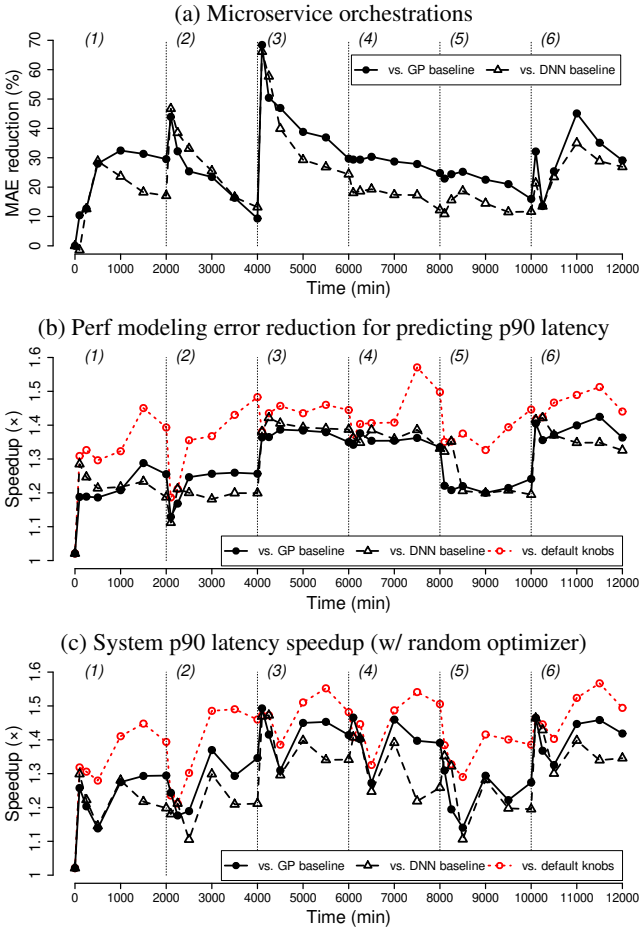


Figure 6: Hotel Reservation on 6-VM cluster (Intel 8272CL).

brings a different set of configuration knobs to performance modeling.

On the other hand, Fluxion is able to localize the inference graph regions (or some learning assignments) that require updating, without changing the rest of the graph. For example, when Hotel Reservation scales-out from 15 to 142 services, graph updates replicate all services' instance-vertices and train their service-vertices. The former incurs no costs, and the latter represents only 15 of the 157 learning assignments in the inference graph. Similar observations can be made for TrainTicket (c.f. Fig 5) and Boutique (c.f. Fig 8).

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=50)	Deploy "Hotel Reservation"	15	63
(2) Stress RPS (=1,200), change req ratio	Scale-out *all* services (6×)	70	253
(3) Stress scale	Scale-out *all* services (6×), *all Memcached* (10×)	142	1,034

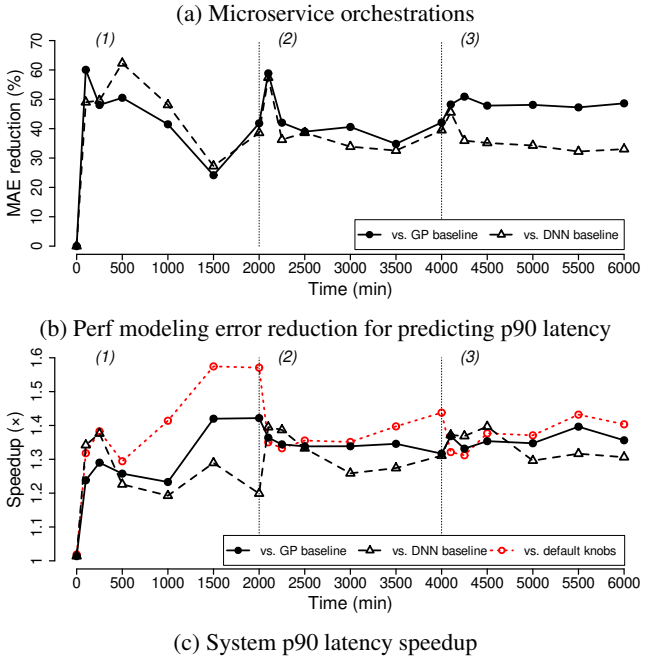


Figure 7: Hotel Reservation on 100-VM cluster (Intel 2673).

5.1.2 Model Adaptation Time Reduction

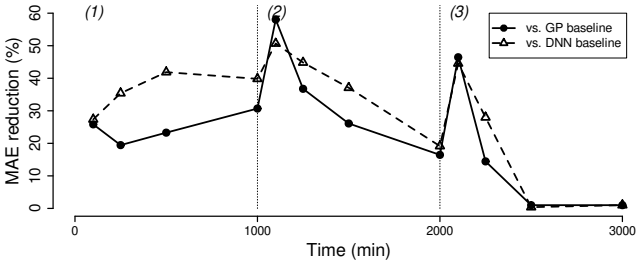
After each orchestration operation, all modeling approaches are updated to match inputs to system knobs. While baselines need to be re-trained, Fluxion minimizes the overhead by localizing updates to some inference graph regions. If none or only a small number) of regions need updating, graph can immediately achieve low prediction MAE, without collecting training data points. Since collecting data points can be time-consuming (c.f. §2.2), if more training data points are necessary, the modeling accuracy will take longer to improve.

We take a deep dive into TrainTicket. After an orchestration operation, both monolithic GP and DNN baselines generally need at least 300–400 data points, in order to train new monolithic models that have an MAE close to what Fluxion can achieve with only 10–25 data points. If each system benchmark takes ∼10 minutes, this is a reduction of 2,900–3,750 minutes (or up to 30× reduction).

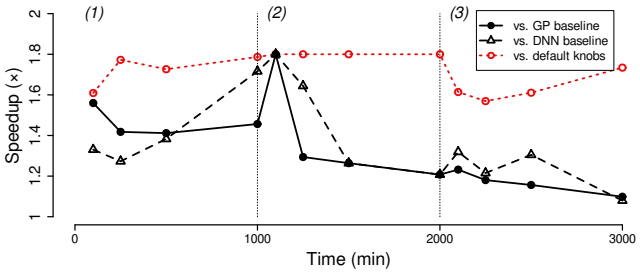
Furthermore, we highlight the case where Hotel Reservation is scaled-out to 142 services. With 500 data points, monolithic GP and DNN models achieve an MAE of 29,576.57μs and 22,575.57μs, respectively. On the other hand, with only 10 data points, Fluxion can already achieve an MAE of 17,716.25μs. Since Fluxion needs to update only a small subset of the learning assignments in the graph, it requires

Step	Triggered orchestration	Service	Knob+state
(1) Init RPS (=100)	Deploy "Boutique"	11	63
(2) Stress RPS (=200), stress scale	Scale-out *all* services (2×)	22	126
(3) Stress RPS (=300), stress scale	Scale-out *all* services (3×)	33	189

(a) Microservice orchestrations



(b) Error reduction for predicting p90 latency



(c) System p90 latency speedup

Figure 8: Boutique on 9-VM cluster (Intel 8171M).

much fewer training data points.

5.1.3 End-to-end System Latency Speedup

Intuitively, a better performance model should enable better performance optimization. Based on Fluxion's MAE reduction shown in §5.1.1, this subsection now quantifies how it can better reduce end-to-end system latencies. To do so, we couple performance models with auto-tuning optimizers (c.f. §2.1): ***(1)*** a random optimizer that selects the best knob setting from randomly generated 100,000 settings, and ***(2)*** Metis [21].

Fig 5c and 5d show TrainTicket's 90th-percentile latency speedup from using Fluxion, over baselines. With Fluxion, the random optimizer achieves a speedup up to 1.41× and 1.43×, over GP and DNN baselines, respectively; the Metis optimizer achieves a speedup up to 1.52× and 1.62×. While the choice of optimizer can impact the auto-tuning outcome, using Fluxion can result in better performance optimization. Even as we introduce deployment dynamics, the speedup is always greater than 1. We note that figures also plot the speedup over default knobs, to demonstrate the benefits of performance optimization.

Fig 7c shows similar observations for Hotel Reservation on the 100-VM cluster — with Fluxion, the random optimizer achieves a 90th-percentile latency speedup up to 1.43× and 1.40×, as compared to relying on GP and DNN baselines, respectively. Even for the last orchestration step where Hotel Reservation is scaled-out to 142 services, the speedup can be up to 1.40× and 1.39×. Furthermore, compared to the default knob setting, Fluxion achieves a maximum speedup of 1.57×.

Fig 6c and Fig 6d illustrate the results for Hotel Reservation on the 6-VM cluster. Fluxion helps the Metis optimizer to achieve a speedup up to 1.49× and 1.47×, over GP and DNN baselines, respectively. Fig 8c shows Boutique, where the random optimizer achieves a speedup up to 1.81× and 1.79×, over GP and DNN baselines, respectively.

5.2 Microbenchmarks

5.2.1 Exposing a Spectrum of Performance Metrics

We evaluate the benefits of exposing a spectrum of metrics for external performance dependencies. To do so, we compare the following inference graphs that predicting Hotel Reservation's 90th-percentile latency. In the first inference graph, all services' learning assignments consider (50, 80–99)th-percentile latencies as external performance dependencies from downstream services. In the second inference graph, they consider only the target performance metric, or the 90th-percentile latency. The last inference graph considers only (50, 85, 90, 95)th-percentile latencies, which are suggested by Fluxion's input selection strategy.

We delve into the case when Hotel Reservation is scaled-out by a factor of 6. If we consider all (50, 80–99)th-percentile latencies, the graph MAE is 9,722.53μs. This is a 10.87% lower MAE, as compared to the second inference graph's MAE of 10,779.22μs. Even across a sequence of orchestrations on the 6-VM cluster, the first inference graph MAE is at least 1.39% lower than the second inference graph. We note that the trade off is the inference graph size — the first graph has 2,170 vertices and 17,647 edges, but the second graph has only 110 vertices and 167 edges. Since each vertex references a learning assignment, this trade off can have a significant implication in terms of the training costs, i.e., 660 more learning assignments to train.

The third latency graph tries to include only highly relevant metrics. In the case of scaling-out Hotel Reservation above, this graph achieves a MAE of 10,091.87μs, or 6.81% lower than considering only the 90th-percentile latency. Furthermore, compared to considering all (50, 80–99)th-percentile latencies, the third latency graph reduces the graph size to 419 vertices and 1,055 edges. Although there is a modest 3.80% increase in MAE, the number of necessary learning assignments reduces by 561.

5.2.2 Capturing System's Temporal Dynamics

We evaluate how well learning assignments can re-use previously trained models to adapt to recurring patterns in temporal dynamics. Our evaluations are based on a case study, which

auto-tunes per-service VM resource allocations by predicting incoming VM request's max CPU utilization and lifetime. Particularly, as the VM utilization pattern changes over time, models are trained and added to learning assignments, which then use output weighting to compute outputs.

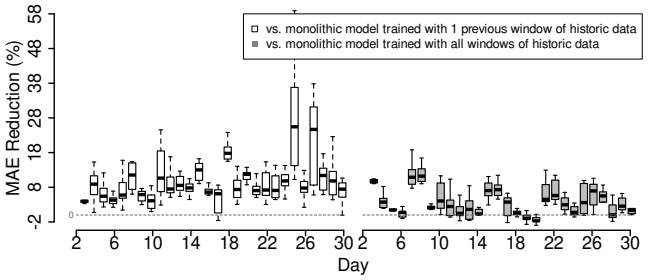
Experiment methodology. We use the 30-day Azure VM utilization trace as the workload, and this production trace was recorded in 2019 [26]. It records resource utilization measurements (e.g., 5-min CPU utilization and VM lifetime) and VM metadata (e.g., VM size and encrypted subscription/deployment ID) for 2,695,548 VM requests.

Following Cortez et al. [7], our baselines are monolithic random forest and extreme gradient boosting tree (XGBoost), for modeling VM's max CPU utilization and lifetime, respectively. And, these metrics are bucketized. Model inputs include encrypted subscription/deployment ID, requested VM size/category, hour of the day, day of the week. We re-train monolithic baselines at the beginning of each day in the trace, with data points from the previous day or all past days. Furthermore, to ensure comparison baselines are properly trained, we tune their hyper-parameters with NNI [27].

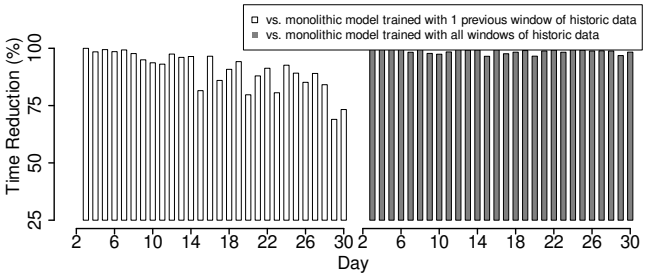
For Fluxion, we set up two learning assignments to represent VM's max CPU utilization and lifetime. We then discretize the continuous trace into non-overlapping batches by days. At the end of each day in the trace, Fluxion evaluates the mean absolute error (MAE). A new model is trained and added only if this MAE is below 85%. The learning assignment gradually accumulates models for future re-uses, and its output is the weighted sum of all its models' predictions. We re-compute the weights by invoking `GE.fit()` every two hours, with recent data points.

Performance modeling error reduction. Fig 9a and Fig 10a suggest that Fluxion can significantly reduce the daily modeling MAE. One reason is that `GE.fit()` can quickly re-adjust weights with the output weighting strategy, rather than going through expensive model training. In summary, compared to baselines trained with the previous day of data, the daily MAE reduction is 2.04%–11.25% and 3.97%–25.43%, for predicting max CPU utilization and lifetime, respectively. Compared to baselines trained with all historic data, the daily MAE reduction is 0.64%–6.49% and -1.16%–11.20%, for predicting max CPU utilization and lifetime, respectively. We note that there are days (e.g., day #20) where Fluxion has a slightly higher MAE than baselines. The reason is that these days exhibit a pattern that significantly drifts from previous days.

Model adaptation cost reduction. Fig 9b and Fig 10b suggest that Fluxion significantly reduces the daily training time, i.e., the time spent on model training and `GE.fit()`. We note that this reduction varies by days, as Fluxion does not need to train new models every day. By re-using models, it trains only a total of 20 and 23 models for predicting VM's max CPU utilization and lifetime, respectively. Furthermore, `GE.fit()` is relatively lightweight — invoking `GE.fit()` 12 times a day takes ∼1,620 and ∼1,480 seconds, for modeling VM's max CPU utilization and lifetime, respectively. So, compared to monolithic random forest and XGBoost baselines trained with the previous day of data, Fluxion reduces the daily training time by 34.93–99.97% and 68.99–99.98%, respectively. Compared to monolithic random forest and XGBoost baselines trained with all historic data, Fluxion reduces the daily training time by 64.55–99.96% and 96.51–99.98%, respectively.



(a) Perf modeling error reduction



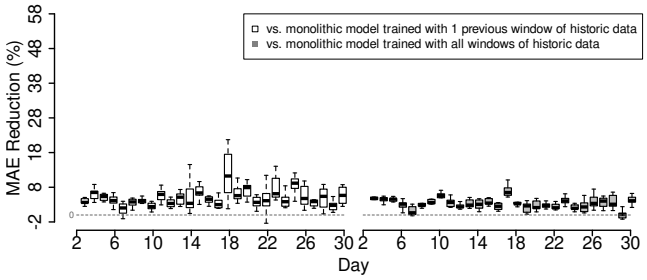
(b) Adaptation time reduction

Figure 9: Benefits of learning assignments in predicting VM lifetime in the Azure 30-day trace, compared to baselines. The output weighting strategy promotes the re-use of models trained at different time periods. We evaluate the prediction error every two hours to produce daily boxplots.

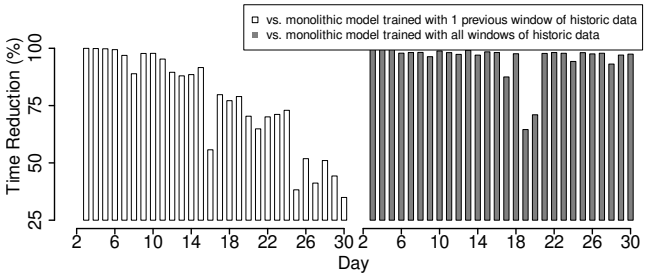
5.2.3 Graph Re-training

As mentioned in §3.2.3, there are two error sources. As previous evaluation has shown Fluxion's benefit on *error source #1* (new learning assignments), this section focuses on *error source #2* (unforseen prediction inputs). As the incoming request pattern changes, different service-to-service execution paths are stressed. We evaluate how well Fluxion can identify the learning assignments at fault in the inference graph.

We conduct experiments by altering the ratio of four request types in the wrk2 workload generator: `search`, `recommend`, `reserve`, and `user`. After we scale-out Hotel Reservation by a factor of 6, we increase the ratio of `search`, `recommend`, and `user` requests from 10% to 30%. At this point, the graph MAE for predicting the 90th-percentile latency is 12,444.63μs. Then, the first iteration of graph debugging identifies MongoDB's learning assignment. After adding a new model trained with recent data points, the graph MAE reduces to



(a) Perf modeling error reduction



(b) Adaptation time reduction

Figure 10: Benefits of learning assignments in predicting VM's max CPU utilization in the Azure 30-day trace, compared to baselines. Output weighting promotes the re-use of models trained at different time periods. We evaluate the prediction error every two hours to produce daily boxplots.

11,024.66μs (or a 11.41% reduction). Subsequent iterations identify the following services: User, Recommendation, and Geo; updating these services reduces MAE to 10,190.89μs (or a 18.11% reduction), 9,483.48μs (or a 23.79% reduction), and 9,352.04μs (or a 24.85% reduction). Beyond this point, the graph MAE reduction starts to exhibit a diminishing return.

6 Related Work

Auto-tuning. Performance tuning for distributed systems is a problem that has continuously received attentions. Instead of relying on heuristics, previous efforts have demonstrated the feasibility of ML-based auto-tuning, for systems ranging from databases [2, 20, 21, 36], storage [6], VM instances [3, 7, 13, 14, 41], cloud services [23], and big data analytics [34].

The focus of this paper is not to apply auto-tuning to new system scenarios, nor to propose new ML techniques. Rather, we are motivated by the limitations of driving auto-tuning with monolithic performance models, especially in the presence of deployment dynamics. We take the first step at abstractions and pieces to systematically bring the concept of modularity to performance modeling. Furthermore, one key question addressed is how this process should be standardized and generalized, without being coupled to specific modeling techniques and systems.

Ensemble of models. The system community has proposed model ensemble as a research opportunity to improve the development speed and adoption in the real world. Stoica et al. [32] describe this opportunity as composable AI systems. Their goal is to query multiple models in different patterns to balance the tradeoff between accuracy, latency, and throughput of a model serving system. In contrast, Fluxion focuses on providing performance modeling for modern systems.

Ensemble learning is a popular machine learning approach that combines multiple models to achieve a higher prediction accuracy on a given dataset [16, 17, 28]. Representative efforts include bagging [5], boosting [8] and so on. Unlike Fluxion, these techniques do not consider system deployment dynamics and the inherent modularity of ML-based performance model for large complex distributed systems.

Previous research efforts have also applied ensemble learning, to realize incremental learning [29, 35]. They inspire our design for learning assignments to keep a list of models.

7 Discussion

We discuss overarching issues regarding modularity level. Fluxion's current design closely follows the system modularity of services, but a finer or coarser modularity level might also seem viable. For Fluxion, the main difference would be in the number of learning assignments. Having said that, we choose the modularity level of services, in order to align with what deployment orchestrations typically operate on. While developers could carefully craft a monolithic system that outperforms service-based counterpart, doing so would complicate everyday CI/CD orchestrations in production. Therefore, we advocate developers to follow the well-known principle of engineering cohesive and loosely coupled services. And, for training, these services should expose appropriate knobs and performance feedback.

8 Conclusion

We report the design and implementation of Fluxion. Fluxion applies the principle of modularity to make performance modeling practical for distributed systems such as microservices. Even under deployment dynamics, empirical results show that Fluxion consistently maintains a higher performance modeling accuracy than monolithic models. This in turn enables auto-tuning tools to better reduce end-to-end system latencies.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Ravi Netravali, for their extensive comments and suggestions.

References

[1] Adam Gluck. Introducing Domain-Oriented Microservice Architecture, 2020.

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. ACM, 2017.

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*. USENIX, 2017.

[4] auto-sklearn. auto-sklearn. http://github.com/automl/auto-sklearn.

[5] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[6] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards Better Understanding of Black-box Auto-tuning: A Comparative Analysis for Storage Systems. In *ATC*. USENIX, 2018.

[7] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understandingand Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*. ACM, 2017.

[8] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

[9] Silvery Fu, Saurabh Gupta, Radhika Mittal, and Sylvia Ratnasamy. On the Use of ML for Blackbox System Performance Prediction. In *NSDI*. USENIX, 2021.

[10] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical Scalable ML-Driven Performance Debugging in Microservices. In *ASPLOS*. ACM, 2021.

[11] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. Unveiling the Hardware and Software Implications of Microservices in Cloud and Edge Systems. *IEEE Micro*, 2020.

[12] Giulio Santoli. Microservices Architectures: Become a Unicorn like Netflix, Twitter and Hailo, 2016.

[13] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *SIGKDD*. ACM, 2017.

[14] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *CNSM*. IEEE, 2010.

[15] Google. Online Boutique. http://github.com/GoogleCloudPlatform/microservices-demo.

[16] L.K. Hansen and P. Salamon. Neural Network Ensembles. In *Transactions on Pattern Analysis and Machine Intelligence*. IEEE, 1990.

[17] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive Mixtures of Local Experts. In *Neural Computation*. MIT, 1991.

[18] Jeremy Cloud. Decomposing Twitter: Adventures in Service Oriented Architecture, 2013.

[19] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-Keras: An Efficient Neural Architecture Search System. In *KDD*. ACM, 2019.

[20] Feifei Li. Cloud-native Database Systems at Alibaba: Opportunities and Challenges. In *VLDB*, 2019.

[21] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly Optimizing Tail Latencies of Cloud Systems. In *ATC*. USENIX, 2018.

[22] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, and Lidong Zhou. The Case for Learning-and-System Co-design. In *SIGOPS Operating Systems Review*. ACM, 2019.

[23] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, and Wenjun Dai. AutoSys: The Design and Operation of Learning-Augmented Systems. In *ATC*. USENIX, 2020.

[24] Locst. Locust - A Modern Load Testing Framework. https://locust.io/.

[25] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC*. ACM, 2021.

[26] Microsoft. Azure Public Datasets. http://github.com/Azure/AzurePublicDataset.

[27] Microsoft. NNI. http://github.com/Microsoft/nni.

[28] Robi Polikar. Ensemble Based Systems in Decision Making. *IEEE Circuits and Systems Magazine*, 2006.

[29] Robi Polikar, Lalita Udpa, Satish S. Udpa, and Vasant Honavar. Learn++: An Incremental Learning Algorithm for Supervised Neural Networks. In *Transactions on Systems, Man, and Cybernetics: Systems*. IEEE, 2001.

[30] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A Tutorial on Thompson Sampling, 2017.

[31] Software Engineering Laboratory of Fudan University. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket.

[32] Ion Stoica, Dawn Song, Raluca Ada Popa, David A. Patterson, Michael W. Mahoney, Randy H. Katz, Anthony D. Joseph, Michael Jordan, Joseph M. Hellerstein, Joseph Gonzalez, Ken Goldberg, Ali Ghodsi, David E. Culler, and Pieter Abbeel. A Berkeley View of Systems Challenges for AI. Technical report, Berkeley, 2017.

[33] R Storn and K Price. Differential Evolution - a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 1997.

[34] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*. USENIX, 2016.

[35] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining Concept-Drifting Data Streams Using Ensemble Classifiers. In *KDD*. ACM, 2003.

[36] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. UDO: Universal Database Optimization using Reinforcement Learning. *VLDB*, 2021.

[37] Cort J. Willmott and Kenji Matsuura. Advantages of the Mean Absolute Error (MAE) over the Root Mean Square Error (RMSE) in Assessing Average Model Performance. In *Climate Research*. Inter-Research, 2005.

[38] wrk2. wrk2. http://github.com/giltene/wrk2.

[39] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. When is the Cache Warm? Manufacturing a Rule of Thumb. In *HotCloud*. USENIX, 2020.

[40] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *SoCC*. ACM, 2018.

[41] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *SoCC*. ACM, 2017.

A Appendix

This appendix provides further information regarding our experiment setup in §5.

- Table 1, 2, and 3 list features (i.e., configuration knobs and states) that we use as performance model inputs, for our three microservice systems. These tables break down these features by microservices.
- Our Fluxion inference graphs use Gaussian Process (GP) models in learning assignments. These GP models use the Matern(5/2) kernel.
- We use NNI to automatically tune hyperparameters for the DNN baseline: the number of hidden layers (3–7), each hidden layer size (100–2,048), and the initial learning rate (0.001–0.1). We budget 24 hours of NNI for each baseline.
- We rely on scripts provided by microservice systems, to generate different request payloads. These requests are then sent by wrk2 or Locust, as recommended by each system. They are available here: Hotel Reservation (https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation/wrk2), Boutique (https://github.com/GoogleCloudPlatform/microservices-demo/tree/main/src/loadgenerator), and TrainTicket (https://github.com/FudanSELab/train-ticket/issues/131).

Service type	Configuration knob	Observable state
"*Frontend*"	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All MySQL	`innodb_thread_concurrency` (8–128) `innodb_buffer_pool_size` (512–3,072) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All MongoDB	`eviction_dirty_target` (10–99) `eviction_dirty_trigger` (1–99) `cache` (50–200) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All TiDB	`storage.scheduler_worker_pool_size` (2–32) `rocksdb.write_buffer_size` (64–256) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All other services	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS

Table 1: Performance model inputs, for TrainTicket.

Service type	Configuration knob	Observable state
"*Frontend*"	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All Redis	`hash_max_ziplist_entries` (32–4,096) `maxmemor_samples` (1–10) `maxmemory` (1–16) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS (read) RPS (write)
"*Email*"	`max_workers` (1–20) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
"*Recommend*"	`max_workers` (1–20) `max_response` (1–5) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
"*Ad*"	`max_ads_to_serve` (1–10) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All other services	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS

Table 2: Performance model inputs, for Boutique.

Service type	Configuration knob	Observable state
"*Frontend*"	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS
All Memcached	`memory-limit` (30–100) `threads` (1–16) `slab-growth-factor` (1.1–2.2)	RPS (read)
All MongoDB	`eviction_dirty_target` (10–99) `eviction_dirty_trigger` (1–99) `cache` (50–200) `net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304)	RPS (read) RPS (write) RPS (update)
All other services	`net.ipv4.tcp_rmem` (4,096–6,291,456) `net.ipv4.tcp_wmem` (4,096–4,194,304) `cpu.cfs_quota_us` (100–300)	RPS

Table 3: Performance model inputs, for Hotel Reservation.

SelfTune: Tuning Cluster Managers

Ajaykrishna Karthikeyan[1] Nagarajan Natarajan[1] Gagan Somashekar[3] Lei Zhao[2]
Ranjita Bhagwan[1] Rodrigo Fonseca[2] Tatiana Racheva[2] Yogesh Bansal[2]

[1]*Microsoft Research* [2]*Microsoft* [3]*Stony Brook University*

Abstract

Large-scale cloud providers rely on cluster managers for container allocation and load balancing (e.g., Kubernetes), VM provisioning (e.g., Protean), and other management tasks. These cluster managers use algorithms or heuristics whose behavior depends upon multiple configuration parameters. Currently, operators manually set these parameters using a combination of domain knowledge and limited testing. In very large-scale and dynamic environments, these manually-set parameters may lead to sub-optimal cluster states, adversely affecting important metrics such as latency and throughput.

In this paper we describe SelfTune, a framework that automatically tunes such parameters in deployment. SelfTune piggybacks on the iterative nature of cluster managers which, through multiple iterations, drives a cluster to a desired state. Using a simple interface, developers integrate SelfTune into the cluster manager code, which then uses a principled reinforcement learning algorithm to tune important parameters over time. We have deployed SelfTune on tens of thousands of machines that run a large-scale background task scheduler at Microsoft. SelfTune has improved throughput by as much as 20% in this deployment by continuously tuning a key configuration parameter that determines the number of jobs concurrently accessing CPU and disk on every machine. We also evaluate SelfTune with two Azure FaaS workloads, the Kubernetes Vertical Pod Autoscaler, and the DeathStar microservice benchmark. In all cases, SelfTune significantly improves cluster performance.

1 Introduction

Large cloud services depend upon cluster managers such as Protean [37], Borg [65], Twine [61], and Kubernetes [6] for job scheduling [32, 33, 39, 47, 53], virtual machine preprovisioning [43], and resource autoscaling [42, 50, 51]. Cluster managers employ algorithms or heuristics to improve metrics such as throughput, latency, and resource utilization. Often, these algorithms rely on multiple configuration parameters that critically influence their behavior, that we call *cluster manager parameters*. For instance, Kubernetes exposes parameters `cpu-histogram-decay-half-life` and `recommender-interval` to help the autoscaler [8] react promptly to changes in cluster utilization without reacting to extremely ephemeral changes in utilization.

Every cluster manager relies on developers[1] to manually set these configuration management parameters to "suitable" values. Table 1 gives examples of such parameters (not exhaustive) for different cluster managers. Typically, developers set these values using a combination of domain-knowledge and a limited set of manually-run tests or canaries [38, 60, 62]. While using domain knowledge is a step in the right direction, limited testing has many disadvantages. First, the tests may not widely explore different values of these parameters in different environments. Second, the search space of feasible values explodes exponentially when multiple interdependent parameters can be tweaked simultaneously. Third, cluster usage can change with time, and the best parameter values would therefore change with time as well. Consequently, clusters with manually tuned parameter values may result in reduced throughput, high request latencies or low resource utilization. For instance, we find that using the default values for `cpu-histogram-decay-half-life` and `pod-recommendation-min-cpu` parameters of the Kubernetes autoscaler drops the system throughput to nearly 50% when the workloads arrive in short, heavy bursts (Section 7).

To address this problem, we observe an interesting similarity between cluster manager algorithms and reinforcement learning (RL) algorithms. Cluster managers (Table 1) often use *state reconciliation*: periodically, they observe the current state of a cluster in terms of health and utilization metrics, compare it to a desired state, and take action to move the observed state closer to the desired state [27]. For instance, the Kubernetes autoscaler [8] continuously determines how to update container sizes, by maintaining a histogram of recent resource utilization values. RL algorithms are also iterative

[1]For brevity, we refer to anyone developing, deploying or monitoring cluster managers – developers, operators, service engineers – as "developers".

Cluster manager	Parameter	Description	Default
Kubernetes (Vertical Pod Autoscaler)	`cpu-histogram-decay-half-life`	How long to wait before halving the weights of past CPU measurements	24 hours
	`recommendation-margin-fraction`	Fraction of usage added as the safety margin to the recommended request	0.15
	`pod-recommendation-min-cpu`	Minimum CPU recommendation for a pod	25 millicores
	`history-length`	Window length for CPU utilization histogram	24 hours
	`pod-recommendation-min-memory`	Minimum memory recommendation for a pod	250 MB
	`memory-histogram-decay-half-life`	How long to wait before halving the weights of past memory measurements	24 hours
	`memory-aggregation-interval`	Window length for memory utilization histogram	24 hours
	`recommender-interval`	How often the resource utilization metrics should be fetched	1 minute
Azure FaaS (App manager)	`prewarm`	Time to wait before pre-loading function code	5
	`keepalive`	Time to wait before retiring the loaded VM	99
Azure Protean (VM allocator)	`num-aa`	Number of rule-based VM allocation agents	
	`k`	*k*-highest quality clusters for VM placement	[8,16]

Table 1: Key numerical configuration parameters of popular cluster management frameworks.

in nature, and use "rewards" to periodically improve and converge a system to an optimal state. Hence we observe that cluster managers are naturally amenable to RL techniques for tuning configuration parameters.

In this paper, we propose SelfTune, a framework that automatically tunes such configuration parameters *in deployment*, rather than through testing. Three key aspects of our framework are: (i) SelfTune piggybacks *solely* on cluster manager's periodic metric measurements, to help tune the cluster manager parameters, so that both tuning and the cluster state reconciliation can occur simultaneously with the same goal of moving the cluster *continuously* towards optimal state; (ii) SelfTune provides a light-weight API for the developers to augment the cluster manager code specifying which parameters to tune (as we illustrate with an example in Section 3), and an objective, e.g., average CPU utilization should be $\geq 60\%$ but $\leq 90\%$; and (iii) SelfTune uses a principled algorithm called Bluefin, based on theoretically-founded ideas for time-varying rewards [35, 52], to optimize the developer-specified objective; it gradually explores choices for the cluster manager parameters, observes the cluster state, and iteratively tunes the parameters to achieve the objective (Section 4).

We have deployed SelfTune on WLM, a scheduler which manages background job scheduling for many Microsoft M365 services including Exchange Online. WLM runs on hundreds of thousands of machines, of which about a third currently use SelfTune's parameter tuning. Our deployment has been running for the last six months. We find that SelfTune has improved cluster throughput by 15%–20% in multiple clusters, while simultaneously improving the resource health in some cases. Based on this, operators are in the process of rolling out SelfTune on the entire fleet of machines.

Despite the simplicity of the Bluefin algorithm, SelfTune is successful *and* has low sample complexity (i.e., number of iterations to converge to the desired cluster state) across applications (Sections 5, 6, 7). This stems primarily from the fact that SelfTune does not learn a single complex model or "policy" for the various scenarios (e.g., high/low workloads) and states (e.g., resource utilization levels, failures) of the deployment environment, unlike standard RL techniques used in systems [45], to tune parameters. Instead, SelfTune relies on pre-determined "scoping" of scenarios (by developers, which is easy in practice) to learn optimal parameters per scope (e.g., one model per machine in our WLM deployment). This scoping, along with light-weight parameter updates (Bluefin) within each scope, makes our solution sample efficient, requiring only about 20 iterations to converge in all our case-studies.

This paper makes the following contributions.
(1) We present SelfTune, a framework that developers can use to automate parameter search for their cluster manager via a minimal interface (Section 3).
(2) We use a novel algorithm, Bluefin, based on rigorously-studied ideas in online learning [35, 52], which allows multiple parameters to be tuned quickly and jointly (Section 4). SelfTune, with Bluefin, enables systems to converge to their objective, i.e., their most desired state faster than previous systems that use Bayesian Optimization [55] and standard RL algorithms [26] (Sections 2.1, 7). We have open-sourced an implementation of SelfTune with Bluefin [14].
(3) We describe our deployment of SelfTune on WLM and show results from multiple clusters where SelfTune achieves up to 15%–20% improvement in the throughput (Section 5).
(4) To the best of our knowledge, ours is the first developer-centric framework for automated tuning of parameters of online systems, not just cluster managers, with large-scale deployments. We show SelfTune's generality in the contexts of (a) resource management for Azure FaaS with production workloads [54] (Section 6) yielding significant improvement in resource efficiency and (b) container rightsizing with Kubernetes and DeathStar benchmark [36] (Section 7), yielding significant improvements in tail latency and throughput.

2 Related Work

Optimally configuring systems is a long-studied research problem in both systems and machine learning [34, 40, 44, 63, 64, 68]. In this section, we describe how SelfTune improves upon previous work, in terms of both the core algorithm it uses, and the framework it provides to the developer.

2.1 Algorithm

Commonly-used techniques for tuning or learning system parameters are variants of Bayesian Optimization [25, 55], reinforcement learning [20, 67], and heuristic search [30].

Bayesian Optimization (BO): CherryPick [23] uses BO to pick the best cloud configuration for big data analytics while Metis [40] uses it to improve performance metrics like tail latency by tuning key system parameters. BO is meant for settings where one seeks global optima of a *fixed* reward function, and requires users to specify a model for the function. For example, CherryPick and Metis use Gaussian Processes to model the prior function. In contrast, Bluefin focuses on online settings where the reward function may change with time, and so does the optimal solution. Also, BO algorithms have high sample complexity, i.e., the number of parameter deployments needed to converge to an optimal solution is large, especially when the number of parameters is also large. Thus BO is not ideal for tuning parameters in deployment or online. Our evaluation in Section 7 confirms this.

Reinforcement learning (RL): RL solutions for tuning database systems [67] or learning scheduling algorithms [45] support continuous parameters but require the system to explicitly provide state information in addition to reward values. For instance, Decima [45] needs every node to specify state in terms of a feature vector consisting of average task duration, number of servers assigned to a node, etc. Defining and implementing states needs domain expertise and engineering effort which is hard to scale across diverse systems. In contrast, Bluefin works with just the reward values and does not need the system to explicitly define such state.

Heuristic search: Using branch-and-bound [30] based techniques for large combinatorial spaces, or domain-specific deductive search for high-dimensional spaces [68] are primarily meant for systems where the goal is to obtain the best configuration parameters for a fixed reward function, and a fixed set of workloads. Often, these techniques do not apply to online settings for the same reason as BO (discussed above); also, heuristic search space modeling lacks the generality of RL techniques like contextual bandits [20, 26] and Bluefin.

SelfTune's Bluefin algorithm addresses the concerns in both BO and state-of-the-art RL techniques. It is a principled gradient-descent based algorithm which (a) needs no modeling, ML expertise, or non-trivial engineering effort, (b) works seamlessly with large real-valued and discrete parameter spaces, and (c) converges to local (or global) optima, with fewer samples than previous approaches.

2.2 Framework

MLOS [31] is a framework to automatically tune configuration parameters using BO; thus, its applicability is limited as discussed above. OpenTuner [24] provides a meta-framework using which domain-specific tuners can in turn be built. CGPTuner [29] considers contextual data, e.g., workload information, for DBMS tasks and uses BO to guide tuning. BestConfig [24] finds good configuration settings using heuristic search and sampling techniques. As discussed in Section 2.1, these techniques do not generalize to dynamic environments unlike SelfTune, where the rewards observed change with time, and in turn the optimal configurations themselves.

OtterTune [64, 66] is a framework for tuning DBMS configuration parameters. Though it also uses a variant of BO for tuning, it incorporates a novel technique to mitigate the risk of using stale configurations for new workloads. It builds ML models for selecting an appropriate workload (from a workload repository) that best represents the current workload, and uses the selected workload to estimate the effect of parameters on the current workload. In contrast, our online setting is much more dynamic, where it is extremely challenging to characterize and maintain such repositories.

AutoPilot [51] reduces resource wastage for containerized workloads using ML techniques for setting job-specific resource limits based on resource utilization. SelfTune is orthogonal to such cluster management frameworks and solutions — in fact, we show how SelfTune helps tune the key parameters of the open-source version of AutoPilot, called Vertical Pod Autoscaler [8], that is part of Kubernetes, in Section 7.

State-of-the-art RL frameworks, e.g., Microsoft's Decision Service [21] are suited for settings where the parameter ("action") space is discrete or categorical, as they rely on "multi-arm bandit" formulations [20]. Extending these techniques to multiple numerical parameters results in very large action spaces which makes it much more challenging to learn (as we see in Section 7). RL frameworks like SmartChoices [28] naturally support numerical parameters, but rely on providing explicit reward separately for each parameter. We, on the other hand, do not require such disambiguation — our problem formulation, and Bluefin, work with a single reward value (i.e., the desired system state objective) for tuning several, possibly inter-dependent, parameters together.

3 SelfTune Overview

We provide an overview of SelfTune and, using a simple example, explain how a developer uses it. Then we describe SelfTune's main system components and their functions.

3.1 SelfTune Interface

To use SelfTune, a developer augments their cluster manager code in four ways. First, they specify the set of parameters SelfTune should tune. Second, they either initialize a fresh SelfTune instance or connect to an existing one. Third, they specify at what point in the code and at what frequency SelfTune should update the values of these parameters. Finally, they use the current state of the cluster to determine a *reward*, which captures the difference between the desired

state and the current state of the cluster. SelfTune's algorithm (in our implementation this is Bluefin) uses this reward to set the parameter values in the next iteration. One of the main insights of this work is that the cluster manager already computes the current state of the cluster, and already has a notion of the desired state of the cluster. Hence SelfTune simply piggybacks on existing code to determine the reward, which is essential for any reinforcement learning platform. The example in Figure 1 shows how a simple token-based job scheduler uses SelfTune to tune the frequency with which it makes scheduling decisions. Using this example, we now describe the SelfTune-specific additions to code in detail.

```
1: public const double optLoad = 0.80;
2: // UpdateCycle = new TimeDelta("00:00:05");
3: Config UpdateCycle = new Config("UpdateCycle",
4:   (1) Specification                "TimeDelta",
5:                                    "00:00:01-00:00:30",
6:                                    "00:00:05");
7: SelfTune st = new SelfTune.Create(UpdateCycle);
8: st.Connect();   (2) Creation
9: // This is the scheduler loop
10: var currentLoad = 0.0;
11: while(1)
12: {
13:    if (currentLoad < optLoad)
14:    {
15:       int numTokens = GenerateTokens(currentLoad);
16:       GrantTokensToJobs(numTokens);
17:    }
18:    Guid callId;   (3) Prediction
19:    UpdateCycle = st.Predict(callId, "UpdateCycle");
20:    sleep(UpdateCycle);
21:    currentLoad = CalculateLoad();   (4) Feedback
22:    st.SetReward(callId, currentLoad - optLoad);
23: }
```

Figure 1: Token-based scheduler augmented with SelfTune to tune the frequency with which its main algorithm runs — the highlighted lines show the three basic additions for SelfTune.

Specify Tunable Parameters: For each parameter, the developer specifies its data type, and optionally, initial value, a range of permissible values, and step-size (e.g., `TimeDelta` data type with values in multiples of 5 seconds). Line 3 in Figure 1 says that SelfTune should tune the `UpdateCycle` parameter, which determines the time between consecutive iterations of the main scheduler loop. Here, the developer has specified that this parameter can lie between 1 second and 30 seconds. They also specify 5 seconds as its initial value.

The developer has determined that `UpdateCycle` should be tuned because if the scheduler waits too long between iterations, it will not react fast enough to changes in cluster state, hence causing the cluster resources to be used sub-optimally. If, on the other hand, the scheduler iterations run very frequently, the scheduler may react prematurely to extremely transient changes to system state, thereby causing sub-optimal resource usage. Note that though this example shows SelfTune tuning only one parameter, one instance of SelfTune can tune any number of parameters simultaneously.

Initialize and Connect to SelfTune Instance: Line 7 in Figure 1 starts a new SelfTune instance. In a cluster-wide deployment, the developer decides how many instances of SelfTune to set up. In our WLM deployment, each machine initializes a separate SelfTune instance. However, if needed, cluster managers can reuse the same instance of SelfTune across various machines, simply by connecting to an existing SelfTune instance (Line 8).

Get Parameter Values: Lines 11 to 23 show the main scheduler loop. Lines 13 to 17 capture the main algorithm of the scheduler. The developer measures the current cluster state as `currentLoad` (set to 0 in Line 10 and updated by the function `CalculateLoad()` in Line 21). The developer states the desired cluster state, i.e. `optLoad`, in Line 1. If the current load of the system `currentLoad` is less than the specified optimal load `optLoad`, it generates a number of tokens proportional to the difference between the optimal load and the current load.

After this, in Line 19, the scheduler invokes SelfTune's **`Predict`** function to determine the value of `UpdateCycle`, and sleeps for `UpdateCycle` seconds. Without SelfTune, the scheduler loop would have slept for a fixed value of 5 seconds, as the commented Line 2 shows.

Set Reward Function: SelfTune needs the developer to specify a domain-specific function to determine the outcome of tuning the specified parameters. Note that the developer's code already defined both `optLoad` and `currentLoad` since the core scheduling algorithm uses them both. The developer reuses this pre-existing code: in Line 22, the developer inputs the difference between `currentLoad` and `optLoad` to SelfTune's **`SetReward`** function as the reward value.

Every reward is a result of a certain set of parameter values. So, the code associates the calls to **`Predict`** and **`SetReward`** using the same `callId`. The Data Collector stores this information for later use (details in Section 3.2).

3.2 SelfTune Components

We now describe the different components SelfTune needs to support the functions in Section 3.1. Figure 2 depicts the four main components: the Client API (which supports the **`Predict`** and **`SetReward`** functions), the Learning Engine, the Data Collector, and the Reward Tracker. Appendix A discusses the specifics of the client API. We describe the rest of the components here.

Learning Engine: The learning engine implements the necessary optimization algorithms such as Bluefin. While SelfTune primarily uses Bluefin, the framework itself is generic and can therefore include other algorithms, e.g., Azure Decision Service's Contextual Bandits.

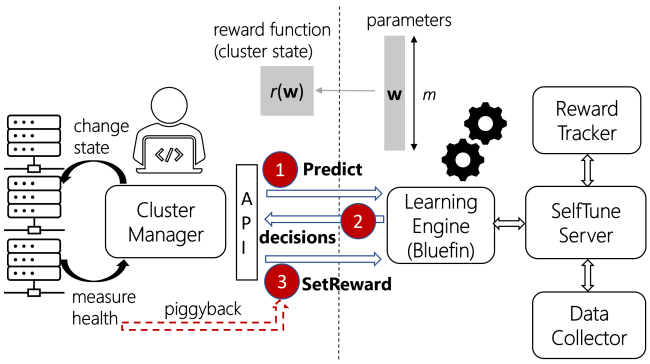


Figure 2: SelfTune architecture. The cluster manager interacts with the SelfTune server via client API. The learning happens on the server side, and it is transparent to the cluster manager.

Section 4 describes Bluefin algorithm that uses a variant of gradient descent. It determines the next value of the parameter based on how the cluster reacted to past parameter choices. For instance, in the example explained in Figure 1, Bluefin observes past values of `UpdateCycle` and the corresponding reward, and then determines the next value with the objective of getting the load as close as possible to `optLoad`.

Data Collector: The data collector is a background service that maintains the history of all parameter values and the corresponding reward for each SelfTune instance. In the example of Figure 1, whenever the client code makes a call to **`Predict`** and a subsequent call to **`SetReward`**, the data collector associates the parameter values and reward using the `callId` and stores this as the tuple $(\texttt{callId}, a, r)$ where a is the value for `UpdateCycle` that the client code obtained by calling `Predict()` and r is the resulting reward. The learning engine uses this data to set future parameter values as described in Section 4. The data collector also stores references to each SelfTune instance so that cluster managers can lookup existing SelfTune instances and connect to them. In our SelfTune implementation, since we use the Bluefin algorithm (Algorithm 1), the space requirement for Data Collector is negligible — it needs to persist only one $(\texttt{callId}, a, r)$ tuple (the most recent), per SelfTune instance (see Section 5).

Reward Tracker: In practical settings, the reward computation may have to happen asynchronously off the critical path; there may not be a natural place in the main control flow to call **`SetReward`**, unlike in the example of Figure 1. In fact, the actual implementation of WLM discussed in Section 5 is such a setting. To facilitate this scenario, SelfTune supports another background service, called the Reward Tracker, which computes rewards periodically, at a frequency determined by the developer, and pushes the values to the data collector.

4 The Bluefin Algorithm

This section describes the Bluefin algorithm used by SelfTune's learning engine. We first define a "round", that is essential to explaining the algorithm. Next, we describe two characteristics essential for making SelfTune generic as well as lightweight. Finally, we describe the algorithm in detail and explain how it achieves both the essential characteristics.

Definition of a round: Standard reinforcement learning (RL) proceeds sequentially in "rounds" between the learning engine and the system whose parameters are tuned. We define a round in the context of tuning deployed systems as the duration for which the system executes with a particular set of parameter values as returned by the calls to **`Predict`**. The client code terminates a round when it calls **`SetReward`**. In Figure 1's example, the developer may introduce an `if` statement around Line 22, checking the last time the reward was set, and setting the reward only if more than a day has passed since. In this case, each day constitutes a round. Alternatively, the developer may share the same SelfTune instance across multiple machines and call **`SetReward`** only after all machines have had a chance to call **`Predict`**; here, a round completes only when all machines have called **`Predict`**.

Characteristic 1: Bluefin **uses One-point Feedback.** Cluster state is the result of a complex combination of parameter values and external factors such as sudden bursts in workloads and time-of-day effects. Therefore, the reward, which is a function of the cluster state, also changes with time. Modeling this behavior using a fixed function is extremely difficult, if not impossible.

Bluefin (like any other RL approach) uses rewards only at the parameter values that the cluster manager obtains by calling **`Predict`**. It does not assume any other information about the inherent, unknown function that determines the reward. In other words, following standard practice in RL literature [21, 56], Bluefin assumes only bandit-feedback or zeroth-order access to the reward function. This constraint is referred to as "one-point feedback" [35], as against multi-point feedback [22] in learning theory. Techniques such as Bayesian Optimization, branch-and-bound heuristics [30], and genetic algorithms [46, Chapter 1.6] need to compute the reward for multiple parameter values that may not have been deployed in the system. Hence they need a model to represent the potentially complex and unknown reward function. Thus, these techniques are much more suited to offline tuning than to our setting of tuning *in deployment* to optimize cumulative time-varying rewards.

Characteristic 2: Bluefin **has Low Sample Complexity and Low Engineering Overhead.** Our goal is to reach the optimal parameters that maximize the cumulative reward over

time. The metric of efficiency is *sample complexity*, i.e., the number of rounds it takes to converge to the optimal values. Each round can be very resource-intensive (as we discovered in SelfTune's deployment on WLM, explained in Section 5), so the fewer the number of rounds the less the overhead of the parameter tuning framework itself. SelfTune reduces the engineering overhead and makes tuning highly sample efficient by letting developers statically identify suitable "scopes" for tuning. That is, rather than learning a single global model to account for all the complex behaviors of the underlying system being tuned, it allows developers to instantiate a SelfTune instance per scope (e.g., WLM uses machine as the scope, in Section 5). Each instance executes Bluefin to learn optimal parameters within its scope, thereby solving a relatively easier problem. Second, Bluefin algorithm (in each SelfTune instance) can be thought of as learning a model of size equal to the number of parameters tuned, unlike standard RL techniques that use sophisticated models with orders of magnitude more parameters to capture system states and behaviors. Thus, both the sample and the engineering complexities of Bluefin is much lower than the standard RL approaches.

Algorithm: We first define a few terms used to explain the algorithm. Say the developer wants to tune m parameters. In each round, the cluster manager receives an m-dimensional vector $\mathbf{a}^{(t)}$ when it calls **`Predict`**, and as a result of setting these values, it measures cluster state and reports back a reward value $r_t(.) : \mathbf{a}^{(t)} \mapsto \mathbb{R}$. SelfTune then uses this reward to update the parameter values.

Algorithm 1 presents the core function of Bluefin, which leverages ideas from the rigorously-studied derivative-free online optimization [35, 52] in the machine learning theory community. There are two key challenges in our tuning setting. First, if we know the exact reward function, r_t, then we can apply the standard online gradient descent techniques [69]. However, in a deployed cluster, we do not have any information on r_t other than the one-point black-box access to it. Second, standard gradient-descent style updates are derived for real-valued parameters. However, cluster manager parameters can be discrete as well as real-valued.

To tackle the two challenges, we leverage the derivative-free optimization ideas studied in learning theory [35, 52]. They showed that we can reliably estimate the gradient of the black-box reward function by randomly perturbing the parameters *once*, albeit under some assumptions on the function. In particular, the theory requires that the problem be continuous, i.e., parameters are all real-valued. In practice, we often have to tune discrete-valued parameters. To this end, Bluefin introduces a function g, which it appropriately defines during the **`Create`** call, to map the real-valued parameters and the generic data-types that can be deployed in the system. In other words, Bluefin executes the well-studied online gradient descent updates in a suitably transformed parameter space. We discuss the details next.

Algorithm 1 Online tuning of parameters in SelfTune

1: **procedure** Bluefin (radius $\delta > 0$, learning rate $\eta > 0$)
2: Initialize the parameters $\mathbf{w}^{(0)} \in \mathbb{R}^m$ // **`Create`**
3: Initialize $g(\cdot)$ // **`Create`**
4: **for** $t = 0, 1, 2, \ldots$ **do**
5: Uniformly sample $\mathbf{u} \in \mathbb{R}^m$ from $\{\mathbf{u} : \|\mathbf{u}\|_2 = 1\}$.
6: Compute *perturbed* parameters $\widetilde{\mathbf{w}}^{(t)} := \mathbf{w}^{(t)} + \delta\mathbf{u}$
7: Client receives *perturbed* decisions $\mathbf{a}^{(t)} := g(\widetilde{\mathbf{w}}^{(t)})$ // **`Predict`** calls
8: Receive feedback $r^{(t)} := r_t(\mathbf{a}^{(t)}) \in \mathbb{R}$ // **`SetReward`**
9: Do "one-point" gradient-ascent update (to maximize the reward): $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \frac{1}{\delta} \cdot \eta \cdot r^{(t)} \cdot \mathbf{u}$

Initialization (Line 2). The algorithm works with a real-valued parameter vector $\mathbf{w} \in \mathbb{R}^m$, where m is the total number of parameters to tune. If the developer does not give an initial value for parameter i, the algorithm samples w_i uniformly at random from the specified range (suitably scaled, see below). If the developer has not provided a range, it initializes w_i to 0.

Defining g (Line 3). If the developer specified a step-size, g appropriately scales the corresponding components of $\mathbf{w}$. For instance, if the developer specifies that the ith parameter is an integer that needs to have a step-size of 5, then $g(w_i) = 5 * \texttt{round}(w_i)$, where w_i is the real value that the algorithm manipulates, and `round` is the round-to-the-nearest-integer function. Similarly, if the developer specified range constraints on the parameter, then g appropriately projects w_i to lie within the specified bounds. **`Predict`** applies the g function before returning the parameter values, as in Line 7 of the Algorithm.

Update parameters (Lines 5, 6, 9). To update $\mathbf{w}$, we use the technique of [35], where we estimate the gradient of r_t with respect to $\mathbf{w}^{(t)}$ by a random perturbation of $\mathbf{w}^{(t)}$. Line 6 effectively samples a vector $\widetilde{\mathbf{w}}^{(t)}$ from the hyper-sphere centered at $\mathbf{w}^{(t)}$ with a radius δ (input to the algorithm, appropriately set as discussed below). Line 9 computes a gradient-*ascent* style update (to *maximize* the cumulative reward) in the direction of the random vector $\mathbf{u}$ scaled appropriately by the learning rate η, and the observed reward value r_t at the perturbed vector. In some cases, such as in simulation settings, one may be able to perturb the vector more than once and make reward measurements. It turns out that with just two-point feedback, we can get a very accurate estimate of the gradient (in lieu of Line 9) as noted in the following remark.

Remark 1 ("Two-point feedback")**.** *The accuracy of gradient estimation, and in turn the sample complexity of Algorithm 1, can be further improved [57] in settings (e.g., simulations in Section 6) where it is possible to obtain reward $r_t(\cdot)$ at two*

different **a** *values. In that case, the gradient estimator in Line 9 of Algorithm 1 can be replaced with:*

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \eta \frac{1}{2\delta}\big(r(g(\mathbf{w}^{(t)} + \delta\mathbf{u})) - r(g(\mathbf{w}^{(t)} - \delta\mathbf{u}))\big)\mathbf{u}$$

Setting radius δ, learning rate η. In general, choosing a single real number δ can be tricky especially when the parameters have different scales. But, decoupling the deployed parameters **a** (that may have very different scales) from the weights **w** (that are in a normalized scale) via $g(\cdot)$ mitigates this issue in practice. Given $\|\mathbf{w}^{(t)}\|_2 = O(1)$ and $\|\mathbf{u}\|_2 = 1$, we set $\delta = O(1)$ and $\eta = O(\delta^2)$, so that $\mathbf{w}^{(t+1)}$ retains the scale after the update in Line 9 of Algorithm 1.

5 Large-scale Workload Scheduling System

In this section, we describe our experiences deploying SelfTune with WLM (short for "workload manager"), the background task scheduler for Substrate, a large data management engine used by many of Microsoft's services. We first describe Substrate and WLM, and then the deployment of SelfTune with WLM, and finally present evaluation.

5.1 Substrate

Substrate is a large-scale data management engine at Microsoft which hosts data for several of Microsoft's enterprise services such as Exchange Online, an enterprise email service, and SharePoint, an online collaboration platform. Substrate stores data in a local database on each machine. Substrate runs upon hundreds of thousands of machines worldwide and hosts billions of data items.

In Substrate, compute and storage are tightly coupled. Each machine runs many *user-facing* tasks, such as reading emails, writing documents, and searching through data. These tasks are latency-sensitive and need to complete within a few milliseconds. Simultaneously, Substrate runs a vast range of *background tasks* on the same machines such as data indexing, data analytics, machine learning, and data defragmentation. More than 70% of all tasks that run on Substrate are background tasks. An example background task analyses a customer's mailbox to provide daily to-do lists [10]. Most tasks are defined to finish very quickly (e.g., process one mailbox and return), in the order of a few seconds.

5.2 WLM

To ensure that background tasks do not interfere with user-facing tasks, Substrate uses a background task scheduler called WLM which regulates these tasks' access to disk,[2] CPU, memory, and network on that machine. WLM continuously polls the background task queues, granting the tasks access to resources when permissible (to ensure high throughput), while trying to keep resource utilization on the machine within a specified range (to ensure room for user-facing tasks).

[2] majority of Substrate data is hosted on cost-effective HDD media

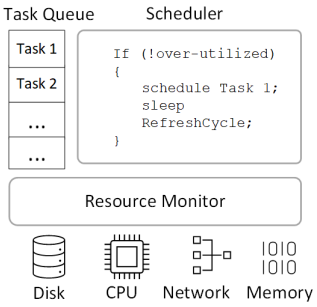


Figure 3: WLM service architecture.

Figure 3 depicts WLM's scheduling algorithm. WLM's resource monitor continuously tracks CPU, disk, network, and memory usage (IO latency for disk, utilization % for CPU and memory, and a function of bandwidth utilization and ping losses for network). For each resource, developers specify a lower and a higher usage threshold. If the resource's utilization is under the lower threshold, the resource is said to be *under-utilized*. Similarly, if the resource's utilization is over the higher threshold, WLM considers it to be *over-utilized*.

Configuring WLM**:** Every few seconds, determined by a configuration parameter called `RefreshCycle`, WLM updates a state variable called `MaxConcurrency`. `MaxConcurrency` determines the maximum number of background tasks that can run on a machine simultaneously. WLM operates an Additive Increase, Multiplicative Decrease (AIMD) algorithm to determine `MaxConcurrency`: every `RefreshCycle` seconds, it determines resource usage. If all four resources are under-utilized, WLM increments `MaxConcurrency` by 1. Even if even one of the resources is over-utilized, WLM cuts `MaxConcurrency` to half its current value. For instance, the developer may set the higher threshold for CPU usage to 60%, the idea being to reserve 40% for the more important user-facing tasks. If WLM observes that background tasks are using more than 60% CPU, it decreases `MaxConcurrency` to half the current value, thereby going into a mode of rejecting tasks until the usage comes down sufficiently. WLM thus gradually schedules more tasks and increases resource utilization, while also checking that no resource is over-utilized.

The ideal value of `RefreshCycle` depends on machine type and workload characteristics. A less powerful machine might benefit from a larger `RefreshCycle`. A smaller value of `RefreshCycle` may help machines with variable workloads. In the absence of an automated tuning framework, WLM's developers have set up different versions of this parameter such as `CPU-RefreshCycle`, `machine-type-A-RefreshCycle`, etc. to control it in dif-

ferent contexts. This approach increases the number of configuration parameters, hence management overhead, as well as the developer burden to continuously check cluster state in these various contexts and manually tweak parameters.

Our deployment of SelfTune automatically and continuously tunes only one configuration parameter – `RefreshCycle` – for every machine independently, which is the scope identified by the domain experts. Developers can now stop using the context-specific `RefreshCycle` parameters, and also stop the continuous manual monitoring of the parameter value and its effect on cluster state.

Performance metrics: WLM measures its performance using two metrics, and hence SelfTune uses either one of these as its reward metric. The first, called a resource's *Healthy Utilization Percent* (HUP), measures the fraction of time the resource is neither over-utilized nor under-utilized. The ideal value of HUP is 1. WLM usually calculates HUP for every hour and for every resource.

The second metric, *grant ratio* (GR), measures the ratio of the total number of tasks that WLM runs in a given time-period to the total number of tasks that were submitted to it in the same time-period. A grant-ratio of 1 implies that WLM did not reject any task. Thus ideally, WLM needs to drive the cluster to have HUP=1 and GR=1. We use the same metrics, aggregated over a day, as the reward function for SelfTune in our deployments.

While these are the two primary metrics that WLM directly exerts influence over, there are other workload-specific metrics, that are outside the scope of WLM, instrumented by the teams who rely on the scheduler. For instance, background task developers use a higher-level metric, i.e., *background task throughput*, to determine how promptly WLM schedules their tasks. This is measured as the total number of background tasks successfully completed within one day. While SelfTune does not use this as a reward metric, we use this metric to determine if SelfTune does indeed help improve the efficiency of the system (Section 5.3).

Integrating SelfTune: We integrate SelfTune with WLM to tune `RefreshCycle` separately for every machine. While the WLM code-base consists of tens of thousands of lines of code, we required less than 50 lines of code to integrate SelfTune, most of which is replacing parameter usage with **`Predict`**, and setting up the Reward Tracker service (to invoke **`SetReward`** asynchronously, as discussed in Section 3.2) with the appropriate reward function.

We look at aggregate metrics over a subset of machines for a month to set the scale of δ (which helps exploration) and η appropriately. We find that a single, fixed choice of δ and η works across clusters; we do not shrink these parameters to 0 with increasing iterations, which is needed in theory. This helps prevent stagnation when tuning in deployment.

Minimal overhead of running SelfTune: Each Substrate machine runs its own local SelfTune (i.e., its component services) instance; so **`Predict`** calls (executing Steps 5 and 6 of Bluefin) are just like any other local function calls in the WLM code-base. Parameter updates (Step 9 of Bluefin) are extremely light (at most 5 FLOPS) and are made once a day when the reward arrives. To enable debugging, the Data Collector (introduced in Section 3) persists a history of $(\texttt{callId}, a, r)$ tuples from the previous 30 days; this takes at most a few hundred KBs space per instance in production. Overall, there is minimal overhead to operationalizing SelfTune in production, in terms of both compute and space.

We enable parameter tuning with SelfTune on individual production servers via *flights*, a mechanism used for gradually deploying any code change in production. Deployment starts with a few hundred servers, and then slowly expands to more servers. This helps us perform controlled experiments.

5.3 Evaluation

In this section, we first describe our evaluation methodology. Then, we describe our experiments and results.

Evaluation Methodology: A significant challenge we faced while evaluating SelfTune is that resource HUP varies widely week over week in Substrate. Figure 4 shows the disk HUP over six weeks in Aug-Sep 2021 for two randomly chosen machine sets in a representative cluster in South America consisting of 450 servers. The sets contained 225 machines each, and were completely disjoint. The figure shows that, for the same machine set, utilization changes significantly from one week to the next. Hence we cannot evaluate the efficacy of SelfTune simply by observing HUP on the machine set before deploying SelfTune, and comparing it to HUP after deploying SelfTune. However, we also observe that the distributions of disk HUP computed on the two disjoint machine sets are very similar (e.g., relative difference between HUP P50 percentiles of the two sets was $\leq 0.5\%$ for all weeks). Therefore, to evaluate SelfTune, we deploy it on one machine set, called the *Treatment Group*, and compare this machine set's HUP values after deployment to the HUP values on the other machine set within the same cluster, which is the *Control Group*. Similarly, we evaluate grant ratios across the two groups (for the same duration, the relative difference between GR P50 percentiles of the two sets was $\leq 3.0\%$).

Results: We ran three large-scale experiments to evaluate SelfTune. We chose three clusters with sub-optimal values of resource HUP and GR: (1) We chose Cluster 1 because, despite being under-utilized (and thus having low values of HUP), it also had low GR. Developers were submitting background tasks to WLM but a significant fraction of them were surprisingly getting rejected despite low resource utilization. Developers thus reported a trouble-ticket for Cluster 1, and

Cluster	Control Size	Treatment Size	Experiment Duration	Reward	Metric Improvement HUP P25	HUP P50	HUP P75	GR P25	GR P50	GR P75	RefreshCycle Value (minutes) P25	P50	P75
1	144	144	July 1–July 30	GR	SI	SI	SI	214%	178%	169%	5.05	6.00	6.08
2	1000	1000	Aug 25–Oct 12	GR	SI	SI	SI	34%	37%	25%	5.02	10.19	15.11
3	1950	1950	Oct 11–Nov 17	(CPU) HUP	2%	1%	3%	18%	18%	20%	0.016	0.043	0.071

Table 2: SelfTune experiment details, resulting performance improvement (SI=Statistically Insignificant) & `RefreshCycle` values.

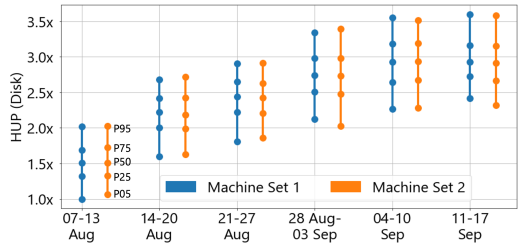


Figure 4: Disk HUP for a cluster in South America (of 450 servers) during Aug-Sep 2021: The (normalized) percentiles drift across weeks ($1\% - 32\%$) significantly; but they vary much less ($1\% - 2\%$) across the two disjoint server sets.

this made it a good candidate for SelfTune. (2) Cluster 2, with predominantly disk-intensive workloads, faced heavy disk throttling and consequently had poor GR. (3) Cluster 3, with predominantly CPU-intensive workloads, had low CPU HUP (CPUs were mostly in the over-utilized state), and consequently, low GR (recall that the CPU `MaxConcurrency` will quickly drop to 1, when it is in an over-utilized state for a short amount of time).

Our objective was to see if SelfTune, by tuning `RefreshCycle` on each machine in the cluster, could improve GR for Cluster 1 and Cluster 2, and CPU HUP for Cluster 3. Thus, in Cluster 1 and Cluster 2, we set up SelfTune with the Grant Ratio (GR measured over a period of one day) as the reward metric. For Cluster 3, we set up SelfTune with CPU HUP as the reward. In Cluster 1, we initialized `RefreshCycle` to 20 minutes since it was the default value used for the cluster. For Cluster 2 and Cluster 3, we initialized `RefreshCycle` to the default value of 6 seconds that was already in use.

(1) SelfTune *improves utilization metrics in all three clusters significantly.* Table 2 describes the duration of the experiments, sizes of the control group and treatment group, and the impact on the performance metrics using SelfTune. In particular, for each cluster, it shows the improvements in the resource HUP and the GR metrics. Given confidentiality requirements, we are unable to present absolute numbers, but present the percentage improvements. Since SelfTune separately tunes `RefreshCycle` on every machine, we present improvement in utilization in terms of 25th%-ile (P25), 50%-ile and 75%-ile of metric values across all machines in the treatment group relative to the corresponding percentile values in the control group (during the deployment period). For all the results, we ensure statistical significance using the standard t-test, at a p-value of 0.05.

From Table 2, we observe significant improvements in GR, between 18% and 178% improvement in the median, across all three clusters. We see drastic improvements in the GR metric in Cluster 1, chiefly due to the sub-optimal and obsolete choice of `RefreshCycle` value used in this cluster (reflected in the Control Group). In Cluster 3, where SelfTune employed CPU HUP as the reward, the improvement in the median CPU HUP was around 2% (also see Figure 5 that shows relative values for confidentiality reasons). Even though the improvement in HUP is small (2%–3%), it is statistically significant; importantly, even a 2% improvement in the median HUP implies several minutes to an hour of better resource utilization *per* machine *per* day for at least 50% of the machine-days in the cluster. The actual impact is magnified manyfolds by the number of machines in the cluster over weeks and months. Furthermore, the small improvement in HUP led to significant improvements (18%–20%) in the GR metric.

(2) SelfTune *has to tune* `RefreshCycle` *separately and continuously for each cluster.* Table 2 gives the P25, P50 and P75 values of `RefreshCycle` that SelfTune converged to in each cluster. We find that Cluster 1's `RefreshCycle` values converged to a median value of about 6 minutes, Cluster 2's median value was about 10 minutes, whereas Cluster 3's median value was much lower, i.e., 2.6 seconds. Additionally, in some cases, there is a significant spread of converged values within a cluster, as the P25 and P75 values show. Such differences in the ideal values of `RefreshCycle` are due to various reasons, such as varying workload characteristics and provisioned hardware even within the same cluster. Moreover, these workload and hardware characteristics also change with time, which means SelfTune should continuously run on every cluster for WLM to be able to react appropriately and quickly to such changes. Figures 11, 12, and 13 in Appendix B show how `RefreshCycle` converges differently for the three clusters over the course of deployment duration.

(3) SelfTune *significantly improves background task throughput.* SelfTune uses either resource HUP or GR as reward metrics since WLM already calculates these metrics. Ultimately, however, background task developers want a high background task throughput. We therefore evaluate how SelfTune improves this metric. Figure 6 shows the improvement in the task throughput when SelfTune was enabled in the

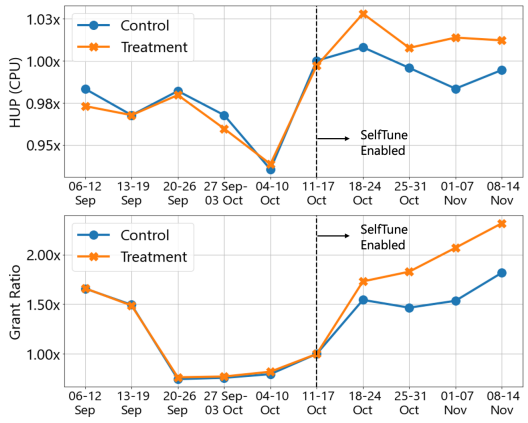


Figure 5: Both HUP (CPU) and GR (weekly P50 values) are significantly better after SelfTune was enabled in Cluster 3, with a 1% to 3% relative improvement over the control set in utilization and a 12% to 34% improvement in GR.

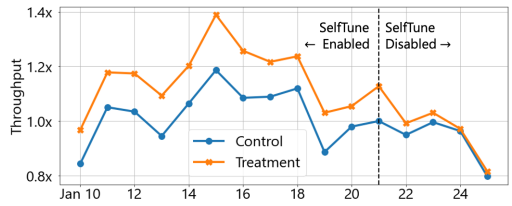


Figure 6: Background task throughput (normalized w.r.t. Jan 21st) clearly improved when SelfTune was enabled.

first week of Jan 2022, in a random half of a 750-machine cluster in the Asia-Pacific region. Before we enabled SelfTune, the throughput of both control and treatment groups were similar. Once enabled, SelfTune improves the background task throughput by as much as 17%. We disabled SelfTune on Jan 21, and the treatment group's throughput went back to being the same as that of the control group. This shows that by improving resource HUP and/or GR, SelfTune significantly improves the background task throughput as well.

Since SelfTune has shown significant improvements in multiple metrics in our experiments, starting January 2022, we have enabled SelfTune in all Substrate clusters in North America, consisting of tens of thousands of machines.

6 Serverless Scheduling in the Cloud

Customers are increasingly using serverless computing, or "Functions as a Service" (FaaS), for deploying applications on the cloud [1, 3–5]. Previous work has proposed informed resource management strategies to use cluster resources efficiently for FaaS applications [54]. We evaluated SelfTune with this work and observed significantly improved resource usage with minimal to no performance loss. This section describes the problem, experiments, and results.

6.1 FaaS Resource Usage

Cloud providers charge FaaS-based applications for the number of functions executed, and not for the resources that the applications use. Hence, to maximize their benefit, providers seek to offer good function performance to customers with the least resources assigned to run the customers' functions.

To achieve good function performance, the provider should load the customer's application into memory before the customer invokes the function (warm start), as opposed to loading it from persistent storage only after the customer invokes the function (cold start). However, keeping all applications in memory at all times is prohibitively expensive. Ideally, the provider should pre-load the customer code *just before* the function is invoked. This approach will minimize the resources that the provider assigns to this application and yet provide good performance.

To achieve this, Shahrad et al. [54] have proposed a policy that predicts two key parameters for a FaaS platform: 1) `prewarm`: The time the policy waits, since the last execution, before it loads the application image expecting the next function invocation. A large value of `prewarm` reduces resource usage but may cause cold starts. 2) `keepalive`: The time for which an application is kept in memory after it has been loaded in memory. A larger `keepalive` can reduce cold starts but will also waste resources. Therefore, the challenge is to predict suitable values of `prewarm` and `keepalive` that will provide good function performance and, at the same time, reduce resources used.

To determine these parameters, Shahrad et al. maintain a histogram of time between function invocations for each application, called the *Idle Time (IT) histogram*. Based on an empirical study, they suggest using `keepalive` = 99th percentile[3] and `prewarm` = 5th percentile of the IT values in the histogram for all applications.

6.2 Evaluation Setup and Goals

We hypothesize that it may be sub-optimal to set `prewarm` and `keepalive` to the same value for all applications. Moreover, the IT histogram can change with time, and therefore these parameters should be set not once, but periodically. In this section, we seek answers to the following two questions:

1. Per-application Tuning: Can SelfTune set `prewarm` and `keepalive` for each application (i.e., use application as the scope for SelfTune instance) based on its invocation patterns, to achieve a better performance trade-off for the cloud provider? (Section 6.3)

2. Time-varying Tuning: Can SelfTune periodically tune these parameters to improve the trade-off, as the invocation patterns could change over time? (Section 6.4)

[3] henceforth, we write `keepalive` = 99, dropping the percentile

Simulation setup: We use the Python-based simulator used in [54] which replays real function invocation traces (obtained from Azure as described in Sections 6.3 and 6.4), and infers if each invocation creates a warm or cold start. The simulator also keeps track of when each application image is loaded in order to aggregate the wasted memory time for the application (i.e., the time the image is kept in memory without executing any functions). Following [54], we simulate (a) function execution times equal to 0 to quantify the worst-case wasted resource time, and (b) all applications use the same amount of memory (as memory data is only partially available).

Performance metrics: We focus on two metrics, following the analysis presented in [54]: (i) distribution (in particular, P75) of percentage of cold start invocations per application (i.e., what fraction of invocations of the app during the time period were cold starts), and (ii) wasted memory time (as defined above). We normalize (ii) w.r.t. a baseline policy of using *no* prewarm and a fixed 10-minute `keepalive` (absolute value, unlike the percentile values used throughout this section). We use the same metrics as reward for SelfTune.

6.3 Per-application Tuning

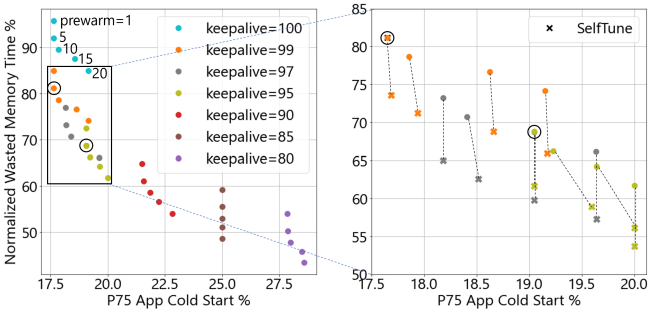


Figure 7: Performance of the VM management policy [54] on AzureFaaS data: (left) sweep of `prewarm` and `keepalive`, fixed for all apps; circled dots are the choices recommended in [54]; (right) with SelfTune for tuning the two parameters for each app, and memory wastage as reward; the dots are the starting points for SelfTune, and the corresponding crosses indicate the performance at convergence.

To answer the first question, we use the AzureFaaS dataset [2] consisting of 14 days of function invocation traces for about 22,000 applications running on Azure Functions.

Optimal global parameters: We first obtain the pareto-optimal trade-off frontier for the two parameters when they are fixed to the same value for all applications. To obtain this frontier, we did a simple grid-search with 7 `keepalive` values (100, 99, 97, 95, 90, 85, 80) and 5 `prewarm` values (1, 5, 10, 15, 20), i.e., we ran 35 simulations which took under three hours on a standard 64-core machine for this dataset (obviating the need for clever optimization/search algorithms). Figure 7 (left) plots normalized wasted memory time vs P75 app cold start percent. We see that one metric improves at the expense of the other metric, for various choices of `prewarm` and `keepalive` parameters. Our findings here align with [54], and the choices circled in black are indeed their recommendations: `prewarm` = 5, and `keepalive` = 99 that favors cold starts; or `keepalive` = 95 that reduces memory wastage by 15% at a small cost ($<$ 9%) of cold starts, relative to `keepalive` = 99.

Optimal application-specific parameters: Doing a grid-search to determine application-specific parameters is very expensive since there are tens of thousands of applications. So we leverage SelfTune to determine *per-application* values of `keepalive` and `prewarm`. On one week of data, every time a function is invoked in the trace, we call **`Predict`** to determine the values of `keepalive` and `prewarm` for the application. The reward metric used is either wasted memory time or number of cold starts. We then evaluate the converged per-application parameter values on the second week of data.

Figure 7 (right) plots wasted memory time vs cold start percent when using SelfTune. We first observe that SelfTune, with memory wastage as the reward, reduces memory wastage by nearly 10% relative to the fixed optimal global choices (indicated in circled dots on both the right and left plots) *without* worsening cold starts. Second, application-specific tuning yields strictly better choices than the global frontier — the crosses (corresponding to the converged parameters) lie below the dots (initial values). We made similar observations when we used number of cold starts as the reward.

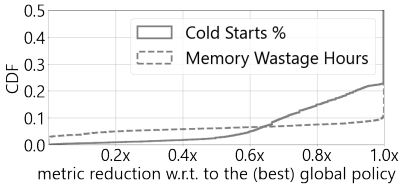


Figure 8: CDF of app-wise reduction in the cost metrics relative to the best global policy (circled in Figure 7) achieved via SelfTune on AzureFaaS. All improvements come from $\lesssim$ 20% of the apps (axis curtailed for clarity).

Figure 8 shows that the overall cost reduction with SelfTune can be attributed to less than 20% of the apps. SelfTune is able to exploit the behavior of a fraction of apps to find better choices of parameter values, while for the other apps, the default global policy parameters already work quite well.

6.4 Time-varying Tuning

To answer the second question, i.e., whether SelfTune's periodic parameter tuning helps reduce resource usage over time, we collected a much larger set of traces from the Azure FaaS

infrastructure between July 15 and Oct 31, 2019 and used them to drive the simulator. As in the previous section, in this large-scale study, we divide the traces into pairs of consecutive weeks, use SelfTune to tune parameters *per-application* on the first week, and evaluate the impact on the second week. Since we use 14 weeks of data, we have 7 such pairs.

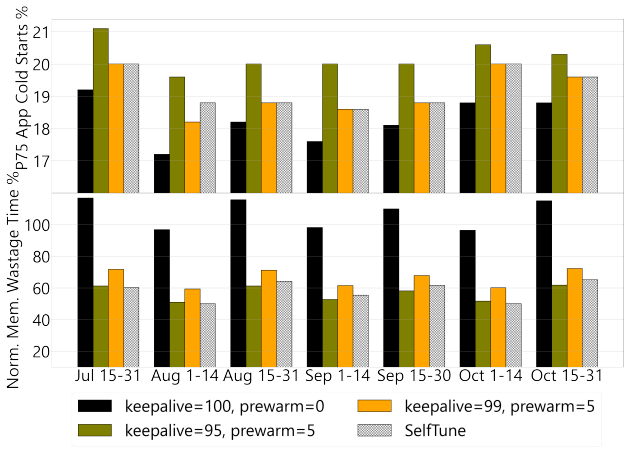


Figure 9: Performance of VM management policy [54] with app-specific tuning of parameters via SelfTune on the large Azure dataset: SelfTune is consistently superior or competitive w.r.t. the baselines along both the metrics, across weeks.

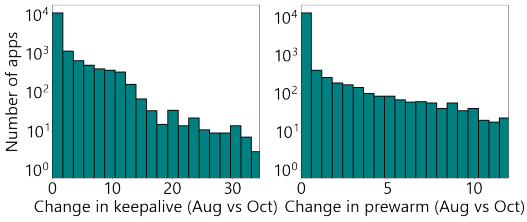


Figure 10: Distribution of differences in converged values, for the two parameters, over 3 months. SelfTune picked significantly different values in October vs. August, for over 25% apps, reflecting the temporal shift in the invocation patterns.

Figure 9 shows the value of P75 application cold starts and normalized wasted memory time with SelfTune and three baselines. We have included a baseline policy that achieves the best possible cold starts (`prewarm` = 0, `keepalive` = 100) for calibration. For SelfTune, we use multiple initial values as in Figure 7, and pick the best results obtained. Relative to the `keepalive` = 95 policy, on average, SelfTune reduces the cold starts by 5%, while incurring a 2.1% larger memory wastage. Also, relative to the `keepalive` = 99 policy, SelfTune yields 12.5% less memory wastage for a small (0.5%) increase in cold starts.

Figure 10 shows how SelfTune changes parameter values (for a random subset of apps) in October compared to August. SelfTune picks significantly different values, up to 300% relative change, for over 25% of the applications. This underscores the importance of continuously tuning the parameters.

7 Container Rightsizing

In this section, we show how SelfTune can be integrated with microservices architecture and Kubernetes to improve (a) cluster resource utilization, and (b) tail latencies of microservices-based cloud applications. We also present comparisons with BO and RL techniques.

Simulation setup: We use the social networking application in the DeathStar microservices benchmark [36]. We set up a cluster with 4 servers, each with 24 cores, 40GB of memory and 250GB of disk space. We restrict monitoring services to one server to avoid interference and deploy the microservices on the other three servers based on the functionality (e.g., all backend microservices are on one server). We simulate a diurnal workload, with short traffic bursts. Following [58], the workload generator [16] issues GET (read timeline), POST (create new post) requests continuously for 15 minutes at 500 requests per second, in the ratio 9 (GET):1 (POST).

Configuration parameters: We tune two types of parameters: (i) the first 4 CPU-related parameters listed in Table 1 for the Kubernetes VPA (Vertical Pod Autoscaler) [8], which impact the efficiency of autoscaling and throughput, and (ii) about 85 key numerical configuration parameters (2–5 parameters per microservice) for the 28 microservices in DeathStar (as identified in [58]), which impact the application latency.

Compared methods: We compare SelfTune's Bluefin with three standard techniques: (i) Bayesian Optimization — the Gaussian Process (GP) method [25], implemented in [15], and used in [23, 58, 66], (ii) Contextual Bandits [26] RL technique — the ε-greedy algorithm implemented in [19], and used in [20, 21], and (iii) Deep Deterministic Policy Gradient (DDPG) [41], a popular deep RL technique for continuous action spaces used by prior works to tune system parameters [49, 67]. For all the experiments, we initialize Bluefin and BO (GP) with the default parameter values as well as random values, and report the best results. We note that, in this scenario, the initialization does not have a significant effect on the algorithms' convergence. For both the algorithms, the difference in performances yielded by the best configurations obtained with either initialization is around 2%–4%. Each 15-minute peak workload constitutes a sample (a round). We fix a budget of 50 samples for all the methods for fair comparison. We configure the ε-greedy and DDPG algorithms to explore for the first 25 rounds and then exploit for 25 rounds.

7.1 Results

Optimizing throughput: We now demonstrate the significance of tuning Kubernetes VPA parameters. We set up a barebones version of DeathStar application, where Nginx microservice with two replicas serves static content for the GET requests. We use one of the servers in the cluster as

Metric	Bluefin	BO (GP)	ε-greedy	DDPG
Throughput %	**86.1 ± 2.2**	83.9 ± 3.1	71.2 ± 4.3	73.4 ± 5.4
# Samples	**12**	14	13	50

Table 3: Tuning key parameters of Kubernetes VPA.

the controller node and another as the worker node [7]. As the requests are light-weight, we ramp the workload up to 10000 rps, and see how quickly Kubernetes autoscales to catch up with the workload. In general, it has been found that default configuration for the Kubernetes VPA can hurt system performance [17]. For instance, with the default value of `recommendation-margin-fraction` = 0.15, Kubernetes will add a margin of 0.15 * computed CPU recommendation to allow the container to adapt to sudden changes in the workload. This ramp up can be quite slow at such high workloads. On the other hand, setting the parameter to a very large value might help quickly catch up with the heavy workload, but will lead to severe resource wastage once the peak dies.

A natural question is if we can tune the VPA parameters (the CPU parameters from Table 1) to help improve resource utilization. We use the throughput attained (over the 15-minute peak workload), with a penalty on the `cpu-histogram-decay-half-life` value as the reward function, to minimize wastage during off-peak hours.

Table 3 shows the best throughput achieved (mean and std. dev. over 5 deployments of the best parameters) and the number of samples needed by each of the methods to attain the best value. We find both BO and Bluefin converge, fairly quickly, yielding over 75% better throughput relative to the default configuration; Bluefin achieves the best throughput overall (statistically significant), an absolute improvement of 2.2% compared to BO. At convergence, Bluefin sets `recommendation-margin-fraction` to 1.5, and `pod-recommendation-min-cpu` to 850 millicores (see Table 1). This helps Kubernetes auto-scale the containers sufficiently quickly (compared to the default values of 0.15 and 250 millicores respectively) and serve the peak workload of 10000 rps. At the same time, Bluefin (and the other methods) converges to a small value (about 45 seconds) of `cpu-histogram-decay-half-life`, which is ideal for short bursts of workloads: Kubernetes evicts the worker containers right after the peak, thereby freeing up resources.

In what follows, we show how we can also tune the configuration parameters of microservices (running in containers) themselves, in order to improve application latency.

Optimizing tail latency: Microservices that are deployed in containers have multiple configuration parameters [9, 11–13, 18] that influence their performance. For instance, the number of threads of performance-critical microservices (e.g., *compose-post-service* in DeathStar) is known to significantly improve latency [58, 59]. We tuned 85 key numerical parameters of the microservices in DeathStar with P95 latency as the reward for all the methods.

Metric	Bluefin	BO (GP)	ε-greedy	DDPG
P95 latency (ms)	**19.5**	19.9	20.0	20.2
# Samples	**8**	41	30	50
P50 iter. cost (ms)	**20.5**	23.3	29.2	20.6
P75 iter. cost (ms)	**21.1**	33.0	33.2	22.1
P95 iter. cost (ms)	**28.3**	76541.9	67640.3	148543.1

Table 4: Tuning parameters of microservices in DeathStar: The second row indicates the number of samples (i.e., rounds) it took for each method to attain the best P95 latency reported in the first row. The last three rows show the spread of the latencies *while tuning* over 50 rounds.

Effectiveness of Bluefin *in high dimensions*: Table 4 shows the best tail (P95) latency attained by each of the methods and the number of samples they took to achieve the same. We deployed each parameter setting three times, and report the median number. This high-dimensional tuning setting clearly brings out the superiority of Bluefin over the popular techniques in terms of sample complexity. Even though there are 85 parameters, there are only a few parameters that critically influence the reward value. Indeed, Bluefin quickly converges to 19.5ms P95 latency (starting from 31.1ms, corresponding to the default values), with just 8 samples; in contrast, BO and ε-greedy algorithms take 3-5 times as many samples to attain similar latencies. The multi-arm bandits approach (ε-greedy) treats the parameter values as categorical choices and does not exploit continuity of the problem or correlations across the parameters. On the other hand, the deep RL method, DDPG, does exploit, but it has a much higher sample complexity.

We also show the *iteration cost*, i.e., the latency incurred through each round of tuning (which matters in deployments). The spread of the iteration costs for SelfTune indicates convergence close to 20ms. Even though all the compared algorithms eventually converge to statistically similar latency values, they incur several orders of magnitude worse P95 iteration costs than Bluefin. This is strong evidence of the effectiveness of Bluefin for tuning in live deployments, where the reward function can be highly ill-conditioned and can vary wildly in some regions of the explored parameter space.

8 Conclusion

This paper presents SelfTune, an RL-based framework using which cluster managers can tune parameters to improve cluster performance. We have deployed SelfTune with a large-scale task scheduler at Microsoft and show how it has improved overall system throughput. We show that SelfTune significantly improves system performance with experiments on Azure FaaS workloads, Kubernetes's Vertical Pod Autoscaler, and the DeathStarBench microservice benchmark.

References

[1] AWS Lambda. https://aws.amazon.com/lambda/.

[2] Azure FaaS Public Dataset. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md.

[3] Azure Functions. https://azure.microsoft.com/en-in/services/functions/.

[4] Google Cloud Functions. https://cloud.google.com/functions/.

[5] IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[6] Kubernetes. https://kubernetes.io/.

[7] Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/.

[8] Kubernetes Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/pkg/recommender/main.go.

[9] memcached(1). https://linux.die.net/man/1/memcached.

[10] Microsoft Viva Insights. https://docs.microsoft.com/en-us/viva/insights/personal/teams/viva-insights-home#microsoft-to-do.

[11] MongoDB Server Parameters. https://docs.mongodb.com/manual/reference/parameters.

[12] Nginx core functionality. https://nginx.org/en/docs/ngx_core_module.html.

[13] Redis configuration. https://redis.io/topics/config.

[14] SelfTune implementation. https://github.com/microsoft/selftune.

[15] The `scikit-optimize` library: Bayesian Optimization using Gaussian Process. https://scikit-optimize.github.io/stable/modules/generated/skopt.gp_minimize.html.

[16] `wrk2`: HTTP benchmarking tool. https://github.com/giltene/wrk2.

[17] Tuning CPU half-life decay parameter. https://github.com/kubernetes/autoscaler/issues/3684.

[18] Tuning Nginx for Performance. https://www.nginx.com/blog/tuning-nginx/.

[19] The Vowpal Wabbit library. https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Contextual-Bandit-algorithms.

[20] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, et al. Making contextual decisions with low technical debt. *arXiv preprint arXiv:1606.03966*, 2016.

[21] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 7, 2016.

[22] Alekh Agarwal, Ofer Dekel, and Lin Xiao. Optimal algorithms for online convex optimization with multi-point bandit feedback. In *COLT*, pages 28–40. Citeseer, 2010.

[23] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[24] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[25] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, 24, 2011.

[26] Alberto Bietti, Alekh Agarwal, and John Langford. A contextual bandit bake-off. *Journal of Machine Learning Research*, 22(133):1–49, 2021.

[27] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[28] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. Smartchoices: Hybridizing programming and machine learning. *ICML Workshop RL4RealLife*, 2019.

[29] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it

configurations under varying workload conditions. *Proceedings of the VLDB Endowment*, 14(8):1401–1413, 2021.

[30] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[31] Carlo Curino, Neha Godwal, Brian Kroth, Sergiy Kuryata, Greg Lapinski, Siqi Liu, Slava Oks, Olga Poppe, Adam Smiechowski, Ed Thayer, et al. Mlos: An infrastructure for automated software performance engineering. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, pages 1–5, 2020.

[32] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[33] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[34] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proc. VLDB Endow.*, 2(1):1246–1257, Aug 2009.

[35] Abraham D Flaxman, Adam Tauman Kalai, and H Brendan McMahan. Online convex optimization in the bandit setting: gradient descent without a gradient. In *Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 385–394, 2005.

[36] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[37] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:{VM} allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[38] Jez Humble and David Farley. Continuous delivery: Reliable software releases through build. *Test, and deployment automation. Pearson Education*, 1, 2010.

[39] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[40] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 981–992, Boston, MA, July 2018. USENIX Association.

[41] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.

[42] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[43] Chuan Luo, Bo Qiao, Xin Chen, Pu Zhao, Randolph Yao, Hongyu Zhang, Wei Wu, Andrew Zhou, and Qingwei Lin. Intelligent virtual machine provisioning in cloud computing. In *IJCAI*, pages 1495–1502, 2020.

[44] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 189–203. USENIX Association, July 2020.

[45] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19. Association for Computing Machinery, 2019.

[46] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[47] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84, 2013.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox,

and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[49] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[50] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 33–40. IEEE, 2019.

[51] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Aadirupa Saha, Nagarajan Natarajan, Praneeth Netrapalli, and Prateek Jain. Optimal regret algorithm for pseudo-1d bandit convex optimization. In *International Conference on Machine Learning*, pages 9255–9264. PMLR, 2021.

[53] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[54] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[55] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[56] Shai Shalev-Shwartz et al. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194, 2012.

[57] Ohad Shamir. An optimal algorithm for bandit and zero-order convex optimization with two-point feedback. *The Journal of Machine Learning Research*, 18(1):1703–1713, 2017.

[58] Gagan Somashekar and Anshul Gandhi. Towards optimal configuration of microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 7–14, 2021.

[59] Akshitha Sriraman and Thomas F. Wenisch. µTune: Auto-Tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[60] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.

[61] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803, 2020.

[62] Alexander Tarvo, Peter F Sweeney, Nick Mitchell, VT Rajan, Matthew Arnold, and Ioana Baldini. CanaryAdvisor: a statistical-based tool for canary testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 418–422, 2015.

[63] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[64] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[65] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[66] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J Gordon. A demonstration of the OtterTune automatic database management system tuning service. *Proceedings of the VLDB Endowment*, 11(12):1910–1913, 2018.

[67] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.

[69] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 928–936, 2003.

A SelfTune's Client API Implementation

We now formally present the syntax and the semantics of SelfTune's client API (introduced informally in Section 3, and in Figure 1).

Creation. The **`Create`** API creates an instance of the parameter learning problem for SelfTune. This API allows optional arguments that encode domain knowledge for tuning the parameters:
(a) names for the parameters to learn,
(b) (optional) initial values for the parameters,
(c) (optional) constraints on the parameters to be tuned; the API supports range constraints (`min` and `max`), type constraints (e.g., `isInt` = TRUE if a parameter takes only integral values),
(d) (optional) for user-defined types, one could specify step-size (e.g., memory sizes in multiples of 64MB), or scale (e.g., logarithmic or linear).

```
string Create(string[] params,
  Dictionary<string, double> initValue,
  Dictionary<string, Constraints> constraints,
  Dictionary<string, Type> type)
```

Connection. The **`Create`** API sets up a data store instance in the back-end for tuning the specified parameters, initializes the necessary background services to maintain/update this store. A unique identifier to this store instance is returned by the call to `Create`. The **`Connect`** API connects a parameter learning instance to a SelfTune object.

```
void Connect (int problemId)
```

Note that if a store already exists (for the parameter(s) of interest), then the client can directly connect to the instance by referencing the unique identifier to the instance, as the store instances are persistent. This also enables multiple clients (distributed spatially and/or temporally) to query the latest decisions for, as well as give feedback to, the same learning problem.

Prediction. With the **`Predict`** interface, the developer can query the *current* values for the parameters. These values are decided by the learning algorithm (presented subsequently).

```
(int, double[]) Predict (string[] params)
```

Note that **`Predict`** returns a pair of values – a unique identifier which identifies the particular invocation of **`Predict`**, and the predicted value.

Feedback. As shown in Figure 1, the **`SetReward`** interface allows the client to specify a reward value. More generally, it allows the client to associate the value with a particular invocation of **`Predict`**:

```
void SetReward(int invocationId, double
reward)
```

The invocation id helps associate the reward to the parameters (and their values) returned by previous **`Predict`** calls — in particular, the reward value applies to all the parameters that were part of all the **`Predict`** calls since the last **`SetReward`** call.

B Parameter Convergence

In this section, we provide graphs to give the reader an idea of how long it takes for `RefreshCycle` to converge in our experiments with WLM (see Figure 11, Figure 12 and Figure 13)

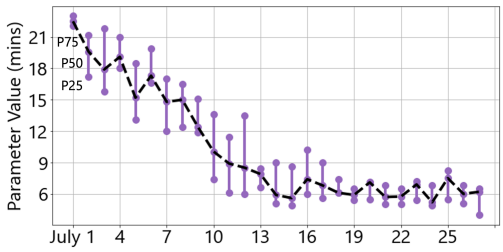


Figure 11: Convergence of `RefreshCycle` with SelfTune in the experiment using Cluster 1.

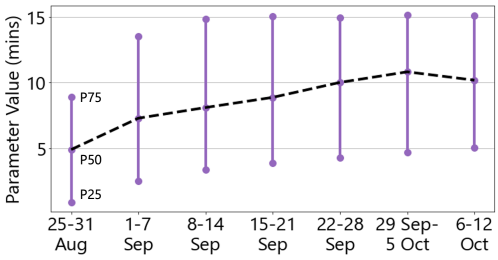


Figure 12: Convergence of `RefreshCycle` with SelfTune in the experiment using Cluster 2.

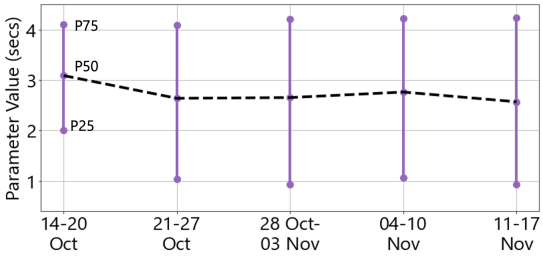


Figure 13: Convergence of `RefreshCycle` with SelfTune in the experiment using Cluster 3.

C Baselines

In this section, we discuss the implementation details of different baselines used in Section 7.

For Bayesian Optimization (BO), we used the `skopt` library [15] with `gp_hedge` as the acquisition function. The algorithm was initialized with the default configuration or with 3 random configurations (uniform sampling), and we reported the best results in Tables 3 and 4.

For Contextual Bandits (CB), we used the popular Vowpal Wabbit library [19]. Since the configuration space is too huge for the bandits formulation to handle, we restrict tuning to the four important parameters (memory limit parameter of the post-storage-memcached microservice, worker_processes and worker_connections parameters of the frontend microservice, memory limit of the post-storage-mongodb microservice) based on empirical observations and recommendations from prior work [58]. Since the algorithm expects discrete actions spaces, we suitably quantize the configuration space of each parameter. We use a *step_factor* for each parameter which yields $(upper_limit - lower_limit)/step_factor$ number of quantized values per parameter. The value of *step_factor* is chosen such that the resulting (discrete) action space is not too large. After discretizing the four parameters in this fashion, we arrived at 24960 actions for the CB algorithm. We used the "explore first" strategy in the ε-greedy algorithm via the command `-cb_explore num_actions -first num_random`, which implies that the algorithm will (only) explore the action space with uniform probability for the first `num_random` iterations.

We implemented Deep Deterministic Policy Gradient (DDPG) [41] using PyTorch [48]. DDPG is a popular policy-based Reinforcement Learning algorithm used by prior works to tune system parameters [49,67]. We use the CPU and memory utilization of microservices on the nodes where microservices are running, workload volume (requests per second), number of clients, and request composition as state features. Both the actor and the critic networks consist of 1 hidden layer. The number of inputs to the actor layer is equal to the number of state features and the output is equal to the number of actions (i.e., parameters tuned). The input and the hidden layer use ReLU as the activation function while the output layer uses Tanh. For the critic network, the number of inputs is equal to the number of state features + the number of actions while the output is just 1-dimensional.

We use 1 step for each episode (to match how the iterations of the baselines and Bluefin proceed) and run the algorithm for 50 episodes. We let the algorithm explore random points for the first 25 episodes followed by 25 episodes where the explored configurations are chosen by the algorithm. To improve the algorithm's ability to explore, we add a Gaussian noise to the action chosen which is controlled by a parameter γ ($\gamma = 0.1$ in our experiments). We update the model after every 5 steps. Once the 50 episodes are complete, we query the model to provide the best configuration for the initial state. We average the rewards over 5 such queries.

CausalSim: A Causal Framework for Unbiased Trace-Driven Simulation

Abdullah Alomar* MIT aalomar@mit.edu
Pouya Hamadanian* MIT pouyah@mit.edu
Arash Nasr-Esfahany* MIT arashne@mit.edu
Anish Agarwal MIT anish90@mit.edu
Mohammad Alizadeh MIT alizadeh@mit.edu
Devavrat Shah MIT devavrat@mit.edu

Abstract

We present CausalSim, a causal framework for unbiased trace-driven simulation. Current trace-driven simulators assume that the interventions being simulated (e.g., a new algorithm) would not affect the validity of the traces. However, real-world traces are often biased by the choices algorithms make during trace collection, and hence replaying traces under an intervention may lead to incorrect results. CausalSim addresses this challenge by learning a causal model of the system dynamics and latent factors capturing the underlying system conditions during trace collection. It learns these models using an initial randomized control trial (RCT) under a fixed set of algorithms, and then applies them to remove biases from trace data when simulating new algorithms.

Key to CausalSim is mapping unbiased trace-driven simulation to a tensor completion problem with extremely sparse observations. By exploiting a basic distributional invariance property present in RCT data, CausalSim enables a novel tensor completion method despite the sparsity of observations. Our extensive evaluation of CausalSim on both real and synthetic datasets, including more than ten months of real data from the Puffer video streaming system shows it improves simulation accuracy, reducing errors by 53% and 61% on average compared to expert-designed and supervised learning baselines. Moreover, CausalSim provides markedly different insights about ABR algorithms compared to the biased baseline simulator, which we validate with a real deployment.

1 Introduction

Causa Latet Vis Est Notissima – The cause is hidden, but the result is known. (Ovid: Metamorphoses IV, 287)

Trace-driven simulation is a widely used method for evaluating new ideas in systems. In contrast to full-system simulation (e.g., NS3 [31]), which requires detailed knowledge of system characteristics (e.g., topology, traffic patterns, hardware details, etc.), trace-driven simulation does not model all components of a system. Instead, it focuses on simulating one (or a few) components of interest, where we wish to experiment with an *intervention*, e.g., a new design, algorithm, or architectural choice. To account for the effect of the remaining components that are not simulated, we collect a trace capturing their behavior and replay it while simulating the component of interest with the proposed intervention.

The key assumption here is that the interventions would not affect the trace being replayed, which we refer to as the *exogenous trace* assumption. If this assumption does not hold, replaying the trace is invalid and could lead to incorrect simulation results. This problem has been referred to as *bias* in trace-driven (or data-driven) simulation [15, 37].

It is difficult to guarantee the exogenous trace assumption in traces collected from real-world systems. Consider, for example, trace-driven simulation of adaptive bitrate (ABR) algorithms [35, 50, 63, 75]. It is common to use network throughput traces from real video streaming sessions on Internet paths [38, 75]. However, the throughput achieved when the player downloads a video chunk is caused by certain *latent* properties of the network path (e.g., the underlying bottleneck capacity, the number and type of competing flows, etc.), as well as the particular choices made by the ABR algorithm (the bitrate chosen for each chunk). In other words, the trace data reflects the combined effect of these two causes and is biased by the ABR algorithms used during trace collection. To simulate a new algorithm, we need to tease apart the effect of the two causes, and predict how the trace would have changed under the decisions of the new algorithm.

We present CausalSim, a causal framework for unbiased trace-driven simulation. CausalSim relaxes the *exogenous trace* assumption by explicitly modeling the fact that interventions can affect trace data. Using traces collected from a *randomized control trial* (RCT) under a fixed set of algorithms, it infers both the latent factors capturing the underlying conditions of the system and a causal model of its dynamics, including the unknown relationship between latents, algorithm decisions, and observed trace data. To simulate a new algorithm, CausalSim first estimates the latent factors at every time step of each trace. Then, it uses the estimated latent factors to predict the alternate evolution of the trace, actions, and observed variables of the component of interest, under the same latent conditions that were present when the trace was collected. This two-step process allows CausalSim to remove the bias in the trace data when simulating new algorithms.

*Equal contribution

CausalSim provides two benefits: (*i*) it improves the accuracy of trace-driven simulation when the intervention could affect (in possibly subtle ways) the trace data; (*ii*) it enables trace-driven simulation of systems where defining an exogenous trace is not possible and therefore standard trace-driven simulation is not applicable. We evaluate both settings in this paper, by simulating ABR and heterogeneous server load balancing algorithms as examples for cases (*i*) and (*ii*) respectively.

CausalSim requires training data from an RCT. Large network operators have increasingly invested in RCT infrastructure to evaluate new ideas, but due to their low throughput and risk of disruptions or SLA violations [42], they can afford to evaluate only a fraction of proposed ideas in RCTs. CausalSim greatly extends the utility of RCT data by learning a model that can simulate a wide range of algorithms using traces from a fixed set of algorithms. Periodically or whenever an operator believes the underlying system characteristics have changed significantly, they can collect fresh data using an RCT (again, with the same fixed set of algorithms) to retrain CausalSim.

CausalSim's design begins with the observation that unbiased trace-driven simulation can be viewed as a matrix (or tensor) completion problem [9, 14]. Consider a matrix M of traces (it is a tensor if traces are higher dimensional), with rows corresponding to possible actions and columns corresponding to different time steps in the trace data. For each column, the entry for one action is "revealed"; all other entries are missing. Our task can be viewed as recovering the missing entries.

A significant body of work has shown that it is possible to recover a matrix from sparse observations under certain assumptions about the matrix and the pattern of missing data. Roughly speaking, the typical assumptions that make recovery feasible are that the matrix has low rank, the entries revealed are chosen at random, and that enough entries are revealed. Low-rank structure is prevalent in many real-world problems [69] and has also been observed in network measurement data [16, 43, 44, 60]. But unfortunately the other two assumptions do not hold in our problem. As we detail in §4.3, one observed entry per column is below the information-theoretic bound for low-rank matrix completion (even for rank $r=1$). Moreover, not only are the entries revealed in our problem not random, they depend on other entries of the matrix, since the actions are being taken by algorithms based on observed variables.

To overcome these challenges, CausalSim exploits two key insights. First, it assumes a causal model (§3) where the latent factors are exogenous and are not affected by the interventions we want to simulate in the component of interest. This *exogenous latent* assumption relaxes (and is therefore implied by) the exogenous trace assumption in standard trace-driven simulation. For example, in ABR, it says that underlying factors like the bottleneck link speed on a network path are not affected by a user's ABR algorithm, whereas ABR decisions can impact the trace that user *observes* (i.e., the achieved throughput).

Second, CausalSim uses a basic property of trace data collected via an RCT. Since the assignment of an algorithm to a trace is completely random in an RCT, the distribution of latent factors should be the same for the traces obtained using different algorithms, i.e., the latent distribution is *invariant* to the algorithm. We provide conditions on the RCT data (e.g., in terms of the number and diversity of algorithms) that guarantee recoverability of the low-rank matrix using this invariance property (§4.2), and we operationalize this idea in a practical learning method that exploits the invariance using an adversarial neural network training technique (§5).

We evaluate CausalSim on two use cases, ABR and server load balancing, with both real-world and synthetic datasets, and further verify CausalSim's predictions with a test in the wild on the Puffer [71] video streaming testbed. Our main findings are:

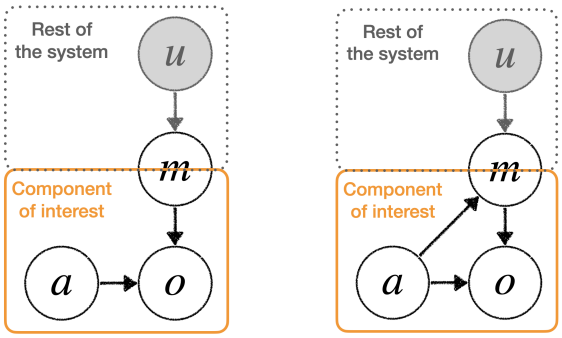
1. We use CausalSim to debug and improve an ABR algorithm, BOLA1 [53, 63]. In a ten month experiment on Puffer [71], BOLA1 exhibited high stalling compared to BBA [35], with slightly better quality. Using CausalSim, we tune BOLA1's parameters via Bayesian Optimization and deploy our improved version on Puffer. We show that it improves the stall rate of this well-known algorithm by 2.6×, achieving 0.7× the stall rate of BBA with similar perceptual quality. The expert-designed baseline simulator that ignores bias predicts the exact opposite: that the new variant should stall 1.34× the stall rate of BBA. This case study shows that removing bias is crucial to draw accurate conclusions from trace-driven simulation.
2. Evaluation of CausalSim on more than ten months of real data from Puffer shows that CausalSim's error in stall rate prediction is bounded to 28%, while expert-designed and standard supervised learning baselines have errors in the range of 49–68% and 29–187% respectively. Similar observations are also made for perceptual quality metrics and buffer occupancy levels.
3. CausalSim opens up new avenues to apply trace-driven simulation to systems where the exogenous trace assumption is invalid. Using a synthetic environment modeling a heterogeneous server load balancing problem, we show how CausalSim reduces average simulation error by 5.1×, a stark improvement compared to a baseline simulator with a median error of 124.3%.

This work does not raise any ethical issues. Our code is available at https://github.com/CausalSim/Unbiased-Trace-Driven-Simulation.

2 Motivation

2.1 Bias in Trace-Driven Simulation

Trace-driven simulation is a widely used technique to design and evaluate systems. Unlike full-system simulation, it focuses on simulating one (or a few) components of the system while capturing the effect of remaining components by replaying a trace. For example, to simulate new ABR algorithms, it is common to replay network throughput traces from real Internet



(a) Trace-driven simulation (b) CausalSim

Figure 1: CausalSim relaxes the *exogenous trace* assumption in standard trace-driven simulation.[1]

paths in a simulator modeling only the video player/server.

As we alluded to earlier, the key assumption here is that the interventions being simulated would not affect the trace being replayed; otherwise, replaying the trace would be invalid. We refer to this as the *exogenous trace* assumption, and it is central to standard trace-driven simulation. Figure 1a is a visual depiction of the exogenous trace assumption. In the figure, a represents the intervention we want to simulate; for example, the actions taken by a new algorithm. o is the observed state of the component being simulated. u represents the *latent* state of the rest of the system, which we do not observe or simulate. Finally, m is the trace, which captures the behavior of the other components.[2] The existence of each edge represents a causal effect. For example, the trace m and intervention a both affect o. Note the absence of the edge from a to m, which implies that the intervention cannot affect the trace (the exogenous trace assumption).

The simulator designer must define the trace carefully to meet this assumption. But what happens if it does not hold, i.e., there exists an edge from a to m (as in Figure 1b)? Ignoring the violation of *exogenous trace* assumption leads to biased simulation outcomes, as we will see next.

2.2 An Example Using Real-world Traces

In this section, we use more than ten months of real-world data from Puffer [71], a recently deployed system for experimenting with video streaming protocols, to illustrate the issue of bias in trace-driven simulation.

Puffer collects data from a continual Randomized Control Trial (RCT) that tests several Adaptive Bit Rate (ABR) algorithms. In the period of interest (July 27, 2020 – June 2, 2021), the tested algorithms include Buffer-Based Algorithm (BBA) [35], two versions of BOLA-BASIC (henceforth called BOLA) [63][3], and two versions of an algorithm called Fugu developed by the Puffer authors. The dataset includes more than 56 million chunk downloads from more than 230 thousand streaming sessions, totaling 3.5 years of streamed videos. For each streaming session, it provides logs of the chosen chunk sizes, available chunk sizes, achieved chunk download throughputs, and playback buffer levels.[4]

Consider a typical trace-driven simulation scenario, where we wish to simulate a new ABR algorithm using traces from previous video streaming sessions. We define such a task on the Puffer data as follows. We let one of the algorithms, say BBA, be the algorithm that we wish to simulate. We leave out the data for this algorithm and ask whether it is possible to predict its performance using the other algorithms' traces. In evaluating a new ABR algorithm, we may be interested in various performance measurements, e.g. buffer occupancy, rebuffering rate, chosen bitrates, etc. Here, we focus on predicting the behavior of playback buffer occupancy, which is one of the key indicators of an ABR algorithm's behavior [35].

The goal of trace-driven simulation is to predict the trajectory of the system (e.g., buffer, bitrates, etc.) for one algorithm in *the same underlying conditions* that were present when a trace was collected using a different algorithm. When simulating algorithm B based on a trace collected using algorithm A, we will refer to A as the "source" algorithm and to B as the "target" algorithm.

It is generally not possible to evaluate the accuracy of individual simulated trajectories using real-world data, because we do not have ground truth trajectories for the target algorithm under the same exact network conditions that were present when running the source algorithm. However, since the Puffer data was obtained using an RCT, we can evaluate predictions about *distributional* properties of the target algorithm, such as the distribution of the buffer occupancy achieved by the algorithm over the population of network paths present in the RCT.

To summarize, our task is: *predict the distribution of the buffer occupancy for the users assigned to BBA (the target algorithm) in the Puffer dataset, using only the data from the other (source) algorithms.*

2.2.1 Simulation via Expert Modeling (ExpertSim)

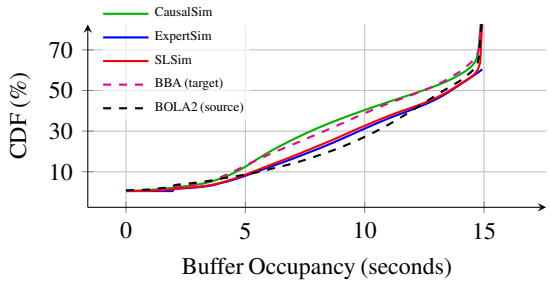
As our first strawman, we build a simple trace-driven simulator (ExpertSim) using our knowledge of how an ABR system works. ExpertSim models the playback buffer dynamics for each *step*, where a step corresponds to one ABR decision and

[1] In general, a and u can be correlated. For example, they can both depend on prior latent conditions of the system. In ABR, for instance, recent latent path conditions are correlated with current path conditions (u), and also affect the action taken by the ABR algorithm (a). Correlation of a and u, however, does not imply a causal relationship between them. In particular, our model assumes *exogenous latents*, i.e. a does not affect u.

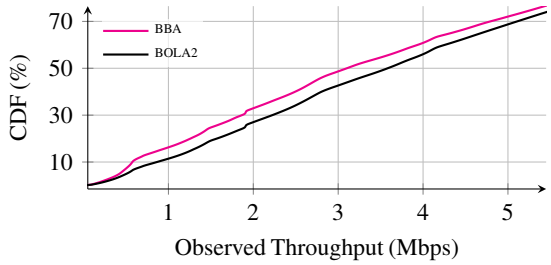
[2] Variables in Fig. 1a can be multidimensional and vary with time.

[3] BOLA1 and BOLA2 are variations on BOLA adjusted to target the SSIM quality metric instead of bitrate [53]. They pursue different objective functions and use different principles for hyperparameter adjustment.

[4] We use 'slow stream' logs (by Puffer's definition, streams with TCP delivery rates below 6Mbps) available on the Puffer website [1].



(a)



(b)

Figure 2: **(a)** CausalSim is accurate in predicting buffer level distribution of BBA users, while baseline simulators' predictions are similar to BOLA2 users. **(b)** Distribution of achieved throughput is different in BBA and BOLA2 users.

the download of a single video chunk. Let $\hat{c}_t$ be the throughput achieved in step t (for the tth chunk) of a particular video streaming session using, say, the BOLA2 algorithm. To simulate BBA for the same user, ExpertSim assumes that the user would achieve the same throughput $\hat{c}_t$ in each step under the BBA algorithm as well. In other words, it assumes that ABR decisions do not affect the observed network throughput (the exogenous trace assumption). Under this assumption, ExpertSim models the evolution of the video playback buffer as follows. Let b_t be the buffer level at the beginning of step t (before the download of chunk t), r_t be the bitrate chosen in step t, and s_t be the size of the tth chunk implied by the chosen bitrate. Then the buffer at the end of step t is derived as: $b_{t+1} = \max(0, b_t - s_t/\hat{c}_t) + T$, where T is the chunk duration.[5] Although simple, the assumption that throughput is an exogenous property of a network path is common in modelling ABR protocols. For example, both FastMPC [75] and FESTIVE [38] assume that the observed throughput does not depend on the chosen bitrate.

Figure 2a shows the true distribution of buffer level for BOLA2 and BBA users in the Puffer dataset (the two dashed lines), as well as the distribution *predicted* by running BBA on the traces collected from BOLA2 users using ExpertSim (solid blue line). The predictions are inaccurate: the buffer distribution generated by ExpertSim is more similar to the buffer distribution of BOLA2 users (the source algorithm) than the buffer distribution of BBA users (the target algorithm).

[5] The complete buffer dynamic equation is slightly more complex to handle cases with full buffers. Refer to §C.1 in the appendix for further clarification.

2.2.2 Simulation via Supervised Learning (SLSim)

Perhaps the simple model of buffer dynamics in ExpertSim does not accurately reflect the actual system behavior. As a next attempt, we turn to machine learning and try to learn the system dynamics from data. Specifically, we use supervised learning to train a Neural Network (NN) that models the step-wise dynamics of the system. This fully connected NN includes 2 hidden layers, each with 128 ReLU activated neurons. For each timestep t, the NN takes as input the buffer level before downloading the tth chunk b_t, the achieved throughput $\hat{c}_t$ for chunk t, and the chunk size s_t (which depends on the birate chosen by ABR). The NN outputs the download time of the tth chunk, and the resulting buffer level b_{t+1}. We train the NN to minimize the prediction error on our dataset. To avoid information leaking, we exclude the logs for BBA from the training data.

Figure 2a shows the predicted buffer level distribution via this approach (SLSim) for BBA. As with ExpertSim, we use the traces collected from BOLA2 users as the source algorithm. The results are similar to ExpertSim; once again, the predicted buffer distribution is closer to that of BOLA2 than BBA.

2.2.3 What Went Wrong?

To understand the limitations of ExpertSim and SLSim, we plot the distribution of achieved per-chunk throughput for users assigned to BOLA2 and BBA in Figure 2b. Since algorithm selection is completely random, we would expect inherent network path properties such as bottleneck link capacity to have the same distribution for users assigned to different ABR algorithms. However, such an invariance should not be expected for achieved throughput, because even on the same path different ABR algorithms could achieve different throughput. For example, since congestion control protocols take time to discover available bandwidth (e.g., in slow start) or converge to their fair share rate when competing against other flows, an ABR algorithm that tends to choose lower bitrates (and hence download less data per chunk) may achieve less throughput than an ABR algorithm that picks higher bitrates [34, 64]. We can see this behavior in the Puffer dataset. The achieved throughput for BOLA2 and BBA is clearly different in Figure 2b.

This confirms that ABR algorithms cause a bias in the measured throughput traces, and the exogenous trace property does not hold. To perform accurate trace-driven simulation, we need to account for this bias when simulating new ABR algorithms.

2.3 Causal Inference to the Rescue!

If the traces were the *underlying network capacity* when each chunk was downloaded (rather than the achieved throughput), the exogenous trace assumption would hold and our problem would be simple. First, we would learn the relationship between network capacity and achieved throughput for different ABR actions using our data. Then, to simulate BBA for a given trace, we would start with the network capacity

at each step of the trace and predict the achieved throughput taking into account the bitrate chosen by BBA in that step. This would then allow us to predict how the buffer evolves. This works because unlike achieved throughput, underlying capacity is an exogenous property of a network path and is not affected by the ABR actions.

However, underlying network capacity is a *latent* quantity — we do not observe it in our traces. The key challenge is therefore to *infer* such latent quantities from observational data. Concretely, in our running example, we wish to estimate the latent factors like network capacity in each step of a trace, using observations such as the bitrate, the chunk size, the achieved throughput, etc.[6]

Inferring such *latent confounders* and using them for counterfactual prediction is the core issue in the field of causal inference [57, 58]. In this paper, we develop CausalSim, a causal framework for unbiased trace-driven simulation. CausalSim relaxes the exogenous trace assumption in trace-driven simulation. It explicitly models the fact that interventions can affect trace data (the edge from a to m in Figure 1b), and infers both the latent factors and a causal model of the system dynamics. This allows CausalSim to correct for the bias in trace data when simulating an intervention. As an illustration, Figure 2a shows the predicted buffer occupancy distribution when simulating BBA on the traces of users assigned to BOLA2, using CausalSim. CausalSim matches the ground-truth distribution for BBA much more accurately than the alternatives.

3 Model and Problem Statement

3.1 Causal Model

Consider the following discrete-time dynamical model[7] corresponding to Figure 1b:

$$m_t = \mathcal{F}_{\text{trace}}(a_t, u_t), \tag{1}$$

$$o_{t+1} = \mathcal{F}_{\text{system}}(o_t, m_t, a_t). \tag{2}$$

Here, t denotes the time index, m_t is the trace, a_t is the intervention, u_t is the latent factor, and o_t is the observed state of the component of interest. The function $\mathcal{F}_{\text{trace}}$ models the effect of interventions on the trace (which traditional methods ignore), and $\mathcal{F}_{\text{system}}$ models the dynamics of the component of interest. When the intervention changes an algorithm in the component of interest, a_t can be viewed as the action taken by that algorithm at time t.

We assume that interventions do not affect the internal state of the rest of the system, i.e., that the latent factors are exogenous. This assumption is implicit in the dynamical system equations, and also visualized in Figure 1b by the absence of the edge from a to u. Note that this is a strict relaxation of the exogenous trace assumption in standard trace-driven simulation. There, the trace itself is assumed to be unaffected by intervention, which also implies exogenous latent factors.

In our running ABR example, we want to simulate the video player and server (components of interest) without precisely modeling the entire network path (the rest of the system). Each time step t corresponds to the download of a new chunk, and u_t represents latent network conditions during that transmission, e.g., bottleneck link speed, number of flows sharing the same network path, type of congestion control used by competing flows, etc. At each time step, the ABR algorithm chooses a bitrate a_t, which together with u_t generate m_t, the *achieved* throughput when downloading a chunk. Typically, latent network conditions are exogenous factors, beyond the impact of a particular user's actions. For instance, the bottleneck link speed and type of congestion control that competing flows use, are not affected by the actions of the ABR algorithm.

Note that the achieved throughput depends on the ABR action as well as the latent network conditions. Equation (1) captures this relationship and is the source of the bias induced by the ABR algorithm, which we demonstrated in §2.2.3.

When is the model applicable? The causal model applies in any trace-driven simulation setting where the trace may be impacted by interventions. Examples include:

- Job scheduling, where we wish to simulate a workload's performance under different types of machines. The trace is the job performance (e.g., runtime), interventions are the scheduling decisions, and latent factors are intrinsic properties of each job (e.g., compute intensity) or latent aspects of the machines such as collocated interfering workloads.

- Network simulation, where we wish to simulate how some aspect of network's design (e.g., congestion control, packet scheduling, traffic engineering, etc.) impacts application performance. The trace is an application's traffic pattern, the intervention is the network design, and latent factors are the internals of the application that dictate its traffic demand.

In some cases, like our running ABR example, the exogenous trace assumption may not hold exactly but still be roughly valid.[8] Here, CausalSim removes bias and improves simulation accuracy. But in certain problems, ignoring the effect of interventions is meaningless. For example, consider scheduling or load balancing on heterogeneous machines (e.g., with different hardware capabilities). Given a trace of job performance on specific machines, it isn't possible to merely replay the trace for new machine assignments. In

[6] For simplicity, we only mention network capacity here, but other latent path conditions like the number of competing flows could also affect achieved throughput and the same reasoning applies to them.

[7] This model is similar to a special type of Partially Observable Markovian Decision Processes (POMDPs) in which the unobserved part of the state is exogenous [51].

[8] Even in these cases, these subtly biased simulations can produce entirely incorrect conclusions (§6.2).

such problems, CausalSim enables trace-driven simulation by explicitly modeling the effect of interventions on the trace.

When is the model invalid? Our causal model relaxes the exogenous trace assumption but still requires *exogenous latents*, i.e. that the latents are unaffected by the intervention. This won't hold in all systems. For example, we cannot model the effect of network routing policies (e.g., BGP) on observed video streaming throughput in this way, since changing the path would change the latent network conditions that impact a video stream. Another example is simulating the effect of a CPU feature like the branch predictor on instruction throughput. Here, we can't model the state of the instruction/-data caches as an exogenous latent factor, since changing the branch predictor can change their internal state significantly.

Overall, a simulation designer needs to reason about the causal structure of observed and latent quantities to define the appropriate model in the form of Equations (1) and (2). However, the designer does not need to precisely specify the meaning of the latents or the dynamics (the functions $\mathcal{F}_{\text{trace}}$ and $\mathcal{F}_{\text{system}}$). CausalSim learns both from observational data.

3.2 Problem Formulation

We are given N trajectories, collected using K specific policies.[9] Let H_i be the length of trajectory $i \in \{1,\dots,N\}$. For trajectory i, we observe $(m_t^i, o_t^i, a_t^i)_{t=1}^{H_i}$. We assume that trajectories are generated using an RCT, i.e., that each trajectory is assigned to one of the K policies at random.

Our goal is to estimate the observations under an arbitrary given intervention (e.g., a new algorithm) for each of the N trajectories. Let $\{u_t^i\}_{t=1}^{H_i}$ be the exogenous latent factors for trajectory i. Formally, for any given trajectory i and given a sequence of actions $\{\tilde{a}_t^i\}_{t=1}^{H_i}$, starting with observation o_1^i and under the same sequence of latent factors $\{u_t^i\}_{t=1}^{H_i}$, we wish to estimate the counterfactual observations $\{\tilde{o}_t^i\}_{t=1}^{H_i}$ that are consistent with Equations (1) and (2).

This is a counterfactual estimation problem since it requires (i) estimating *latent* $\{u_t^i\}_{t=1}^{H_i}$ factors for observed trajectory i and using them along with the counterfactual actions $\{\tilde{a}_t^i\}_{t=1}^{H_i}$ to predict the counterfactual trace $\{\tilde{m}_t^i\}_{t=1}^{H_i}$ consistent with Equation (1), and then (ii) using the counterfactual trace and actions to predict counterfactual observations $\{\tilde{o}_t^i\}_{t=1}^{H_i}$ consistent with Equation (2).

For (ii), learning $\mathcal{F}_{\text{system}}$ is a supervised learning task because its inputs, (o_t^i, m_t^i, a_t^i), and output, o_{t+1}^i, are fully observed. If $\{u_t^i\}_{t=1}^{H_i}$ was observed, then (i) would also boil down to learning $\mathcal{F}_{\text{trace}}$ in a supervised manner. *It is the lack of observability of* $\{u_t^i\}_{t=1}^{H_i}$ *that makes our simulation task extremely challenging. In short, we are left with (i), the task of estimating* $\{\tilde{m}_t^i\}_{t=1}^{H_i}$ *and learning* $\mathcal{F}_{trace}$.

[9] We use policy and algorithm interchangeably in this paper.

4 CausalSim: Theoretical Insights

This section describes the theory behind CausalSim. We discuss how to operationalize this theory in a practical learning algorithm in §5. We begin by casting counterfactual estimation as a challenging variant of the matrix completion problem [14]. We then formalize conditions that allow us to complete the matrix using a certain distributional invariance property that is present in data collected in an RCT.

4.1 Counterfactual Estimation as Matrix Completion

Recall from §3.2 the task of estimating the counterfactual trace $\{\tilde{m}_t^i\}_{t=1}^{H_i}$ consistent with Equation (1). In this section, we pose this task as a variant of the classical matrix completion problem. For simplicity, let action a_t^i be one of the finitely many options $\{1,\dots,A\}$ for some $A \geq 2$. Imagine an A by U matrix M, where rows correspond to A potential actions, and columns corresponds to $U = \sum_{i=1}^{N} H_i$ latent factors (u_t^i for different choices of i and t) in the dataset. To order the columns, we may index u_t^i as a tuple $(i,\ t)$ and order these tuples in lexicographic order. The matrix M is called the potential outcome matrix in the causal inference literature [61].

At the tth step of the ith trajectory, we observe $m_t^i = \mathcal{F}_{\text{trace}}(a_t^i, u_t^i)$, which is the entry in M in the row corresponding to a_t^i and the column corresponding to u_t^i. The counterfactual quantities of interest, $\tilde{m}_t^i = \mathcal{F}_{\text{trace}}(\tilde{a}_t^i, u_t^i)$ for $\tilde{a}_t^i \neq a_t^i$, are the missing entries in M in the same column. In summary, we observe one entry per column of the matrix M and we wish to estimate the missing values in the matrix.

The task of filling missing values in a matrix based on its partially observed entries is known as *Matrix Completion* [19], a topic that has seen tremendous progress in the past two decades [18, 20, 47]. However, standard matrix completion methods do not apply to our problem (see §4.3 for details).

We use a distributional invariance property of data collected using an RCT to complete the potential outcome matrix M. The key observation is that, in an RCT, the latent factors for trajectories collected under each of the policies will have the same distribution. For example, in Puffer's RCT, incoming users are assigned to an ABR algorithm at random. Therefore each ABR algorithm will "experience" the same distribution of underlying latent network conditions, which is precisely why we can compare their performance in the RCT. The same property helps us recover the matrix M, as we show next.

4.2 Exploiting RCT for Matrix Completion

We use a minimal non-trivial example to give intuition about how we can exploit an RCT for matrix completion, before stating our main theoretical result.

Consider a simple example where $A = 2$ and $U = 2n$, and the rank of potential outcome matrix M is equal to 1. Rank 1

implies that $M = au^T$ for some $a \in \mathbb{R}^2$ and $u \in \mathbb{R}^{2n}$ with $M_{\alpha,\beta} = a_\alpha \cdot u_\beta$.[10] Suppose we have $K = 2$ policies, where each policy always chooses only one of the two actions. Furthermore, we consider an RCT setting. That is, the distribution of latent factors across trajectories assigned to both policies should be the same.

Without loss of generality, we can re-order the columns of M so that the first n columns correspond to the latent factors of the trajectories assigned to policy 1, and the second n columns are those assigned to policy 2. Then the observed entries of matrix M appear as

$$\begin{bmatrix} M_{1,1} & M_{1,2} & \dots & M_{1,n} & \star & \dots & \star & \star \\ \star & \star & \dots & \star & M_{2,n+1} & \dots & M_{2,2n-1} & M_{1,2n} \end{bmatrix}$$

where $\star$ represents the missing values.

Let us consider recovering the missing observation $M_{2,1}$. For column 1, we know the observation under the first action, i.e. $M_{1,1}$. Due to rank 1 structure, we have

$$\frac{M_{2,1}}{M_{1,1}} = \frac{a_2 u_1}{a_1 u_1} = \frac{a_2}{a_1}. \tag{3}$$

Therefore, to find $M_{2,1}$ (and by a similar argument, to find all missing entries of M), we need to estimate the ratio $\frac{a_2}{a_1}$.

Due to the distributional invariance induced by RCT, the samples $u_1, \dots, u_n$ (which correspond to the latent factors encountered by policy 1) come from the same distribution as the samples $u_{n+1}, \dots, u_{2n}$ (which correspond to the latent factors encountered by policy 2), for large enough n. Thus, their expected value should be equal:

$$\frac{1}{n}\sum_{\beta=1}^{n} u_\beta \approx \frac{1}{n}\sum_{\beta=n+1}^{2n} u_\beta \tag{4}$$

Equation (4) implies

$$\frac{\sum_{\beta=1}^{n} M_{1,\beta}}{\sum_{\beta=n+1}^{2n} M_{2,\beta}} = \frac{\sum_{\beta=1}^{n} a_1 \cdot u_\beta}{\sum_{\beta=n+1}^{2n} a_2 \cdot u_\beta} \approx \frac{a_1}{a_2}. \tag{5}$$

This provides precisely the quantity of interest in Equation (3) based on the observed entries, enabling us to complete the matrix.

Formal Result. This simple illustrative example relied on a convenient observational pattern (based on policies that always choose one action) and rank 1 structure. But the idea can be generalized. If the trace includes D measurements, $M_{\alpha,\beta,\gamma} \in \mathbb{R}^{A \times U \times D}$ becomes a tensor rather than a matrix, where α, β, and γ index the actions, latent factors, and measurements, respectively. The following theorem provides conditions where completion is possible for a rank r tensor. For more details and the proof, refer to Appendix A.

Theorem 4.1. *We can recover all entries of M by only observing one $D-$dimensional element in each column (corresponding to one latent and action) if the following is satisfied:*

1. ***(Low-Rank Factorization)*** *M is a low-rank tensor (rank $= r$), i.e., it admits the following factorization: $M_{\alpha,\beta,\gamma} = \sum_{\ell=1}^{r} a_{\alpha\ell} u_{\beta\ell} z_{\gamma\ell}$.*
2. ***(Invertibility)*** *The factorization implies existence of a linear mapping from latent encoding to trace for each action. This linear mapping is invertible.*
3. ***(Sufficient measurements)*** *$D \geq r$.*
4. ***(Sufficient, Diverse Policies)*** *The number of policies $K \geq Ar$, and the matrix $\mathbf{S} \in \mathbb{R}^{Ar \times K}$ is full-rank where $\mathbf{S}_{w.D:(w+1).D,x} = \mathbb{E}[m|action_index = w, policy_index = x]\mathbb{P}(action_index = w|policy_index = x)$. Linear independence of columns of $\mathbf{S}$ can be interpreted as diversity among policies (Appendix A).*

4.3 Discussion

Why not standard tensor completion? Tensor completion methods [26, 41, 48, 78] make several assumptions. First, the tensor M must be (approximately) low rank, which CausalSim also requires. Low-rank structure holds in many real-world problems [69] and has been observed in network measurements, e.g., in traffic matrices [16, 43, 44, 60] and network distance (i.e., RTT) [46, 52, 66]. As an example of how it emerges in the problems we study in this paper, we use a simple model of congestion control in Appendix C.4 to provide intuition about low-rank structure in ABR data.

Second, the pattern of missing entries should be *random*. If the missing patterns is not random and depends on latent factors or the entries themselves [8], standard approaches have difficulty recovering the tensor. This assumption does not hold in trace-driven simulation. Revealed entries are determined by the actions taken by the policies, which often use recent observations to make their decisions (e.g., an ABR policy may use recent throughput measurements). Hence the revealed/missing entries in a column are not random and depend on the entries in previous columns.

Third, a sufficient number of entries need to be revealed. For example, when $D = 1$ (i.e., when M is a matrix), the information theoretic lower bound to on the number of revealed entries needed to recover M is $4Ur - r^2$ [39, 70]. Thus even for rank $r = 1$, it requires 4 entries per column, whereas only one entry per column is revealed in trace-driven simulation.

Since the second and third assumptions do not necessarily hold in our setup, we cannot use *existing* tensor completion methods. However, as we argued in §4.2, exploiting the additional problem structure imposed by RCT data can make tensor completion feasible in certain conditions.

Limitations of Theorem 4.1. The proof of Theorem 4.1 (Appendix A) provides an analytical method for recovering the tensor M that generalizes the procedure described for the simple example in §4.2. While this provides a theoretical basis for why tensor recovery is possible, the analytical approach is not practical. First, it relies on M being exactly rank r; if it is approximately rank r, we have found the calculation to be

[10] Note that for readability, we are abusing notation by overloading a and u to refer to both the action and latent, and their encodings in the factorization.

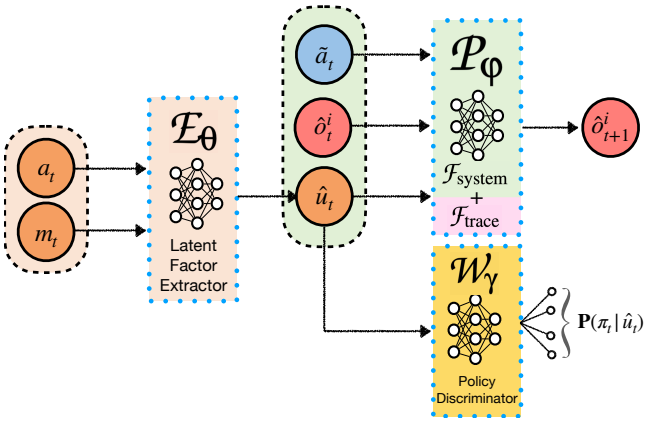


Figure 3: CausalSim Architecture

brittle. Second, it applies only to discrete action spaces. Third, it gives sufficient conditions for recovery, but they're not all necessary. One reason is that the analytical method uses only *mean* invariance, i.e. the fact that the mean of the latent factors is the same across all policies (as in Eq. (4)), even though RCT data has the stronger property that the entire *distribution* of latents does not depend on the policy. In the next section, we describe our practical implementation of CausalSim that uses learning techniques and NNs to overcome these limitations (at the expense of theoretical guarantees).

5 CausalSim: Algorithm

CausalSim builds upon the insights presented earlier but replaces the factorized model with a learning algorithm based on NNs. For ease of notation, we will drop the trajectory index for all variables in the dataset, e.g. we will refer to the latent factor $u^i_t : t \le H_i, i \le N$ as $u_t : t \le H$.

CausalSim architecture. As discussed, CausalSim aims to extract u_t and learn $\mathcal{F}_{\text{trace}}$ and $\mathcal{F}_{\text{system}}$ from observed trajectories $(o_{t+1}, o_t, m_t, a_t) : t < H$. Figure 3 summarizes CausalSim's algorithmic structure.

To extract latent factors, we use a NN that takes in a_t and m_t, and computes $\hat{u}_t$ (an estimate of u_t). To apply invariance on the extracted latents, i.e. distribution of u_t being the same regardless of the policy applied to it, we use a NN called the *Policy Discriminator*. This NN aims to predict the policy pertaining to that sample given $\hat{u}_t$, and if invariance is upheld, it will fail to do so. Unlike the analytical approach, the policy discriminator can enforce policy invariance on the entire latent distribution, potentially improving the accuracy of the estimate.

To calculate the counterfactual traces and observations, we need to learn $\mathcal{F}_{\text{trace}}$ and $\mathcal{F}_{\text{system}}$. However, we can simplify the learning problem by merging these two into one single combined function. Thus, we use a NN that takes in counterfactual actions $\tilde{a}_t$, observation o_t and estimated latent $\hat{u}_t$, and computes counterfactual observation $\tilde{o}_{t+1}$. Of course, we can explicitly use separate NNs for $\mathcal{F}_{\text{trace}}$ and $\mathcal{F}_{\text{system}}$ if we require access to the simulated trace ($\tilde{m}_t$) values.

Algorithm 1 CausalSim Training

1: initialize parameter vectors γ, θ, φ
2: initialize hyper-parameters *num_disc_it*, κ
3: initialize dataset $D \leftarrow \{(o_i, m_i, a_i, \pi_i)\}_{i=1}^m$ from an RCT
4: **for** each iteration **do**
5: ↑ Discriminator **for** *num_disc_it* **do**
6: sample minibatch $B \leftarrow \{(o_l, m_l, a_l, \pi_l)\}_{l=1}^b$
7: $u_l \leftarrow \mathcal{E}_\theta(m_l, a_l)$ for $l \in \{1, ...b\}$
8: $\mathcal{L}_{\text{disc}} \leftarrow \frac{1}{b}\Sigma_{l=1}^b \left[-\log \mathcal{W}_\gamma(\pi_l \mid u_l)\right]$
9: $\gamma = \gamma - \lambda_\gamma \cdot \nabla_\gamma \mathcal{L}_{\text{disc}}$
10: ↓ **end for**
11: ↑ Simulation Modules sample minibatch $B \leftarrow \{(o_{l+1}, o_l, m_l, a_l, \pi_l)\}_{l=1}^b$
12: $u_l \leftarrow \mathcal{E}_\theta(m_l, a_l)$ for $l \in \{1, ...b\}$
13: $\mathcal{L}_{\text{disc}} \leftarrow \frac{1}{b}\Sigma_{l=1}^b \left[-\log \mathcal{W}_\gamma(\pi_l \mid u_l)\right]$
14: $\mathcal{L}_{\text{pred}} \leftarrow \frac{1}{b}\Sigma_{l=1}^b \left[\left(o_{l+1} - \mathcal{P}_\varphi(o_l, a_l, u_l)\right)^2\right]$
15: $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{pred}} - \kappa \cdot \mathcal{L}_{\text{disc}}$
16: $\theta = \theta - \lambda_\theta \cdot \nabla_\theta \mathcal{L}_{\text{total}}$
17: ↓ Simulation Modules $\varphi = \varphi - \lambda_\varphi \cdot \nabla_\varphi \mathcal{L}_{\text{pred}}$
18: **end for**

Overall, CausalSim uses three NNs for counterfactual simulation; $\mathcal{E}_\theta$ as the *latent factor extractor*, W_γ as the *policy discriminator* and $\mathcal{P}_\varphi$ as the combination of $\mathcal{F}_{\text{trace}}$ and $\mathcal{F}_{\text{system}}$. Figure 3 depicts the structure. Training these NNs is quick; on an A100 Nvidia GPU, CausalSim's time to convergence on $56M$ data points ($230K$ streams) was less than 10 minutes, and each simulation step in inference (on CPU) takes less than $150\mu s$. A full inference run on the same volume of data takes less than 6 hours on a single CPU core and less than 20 minutes on 32 cores.

Training procedure. CausalSim's training procedure alternates between: (i) training the policy discriminator using a discrimination loss $\mathcal{L}_{\text{disc}}$; and (ii) training other modules using an aggregated loss $\mathcal{L}_{\text{total}}$. Algorithm 1 provides a detailed pseudo code of this training procedure.

Training the policy discriminator (Lines 5–10 in Algorithm 1). Distributional invariance means restricting the distribution of latent factors u to be identical across policies. To that end, we first use $\mathcal{E}_\theta$ to extract latents $\hat{u}_t$, and then search for invariance violations via a *discriminator* NN, a standard approach in the paradigm of adversarial learning [29, 68]. Specifically, the policy discriminator aims to predict the policy π_i that took action a_t from the estimated latent factor $\hat{u}_t$ (see Figure 3). Towards that, we use a cross-entropy loss to train the policy discriminator:

$$\mathcal{L}_{\text{disc}} = \mathbb{E}_B[-\log \mathcal{W}_\gamma(\pi \mid \hat{u})], \tag{6}$$

where the expectation is over the a sampled minibatch B from dataset D. We train the policy discriminator to minimize this loss, by repeating gradient decent *num_disc_it* times, as the

policy discriminator needs multiple iterations to catch up to changes in the latent factors.

Training simulation modules (Lines 11–17 in Algorithm 1). In this step, we need to impose consistency with observations, all while preserving the distributional invariance. Thus, we compute latent factors $\hat{u}_t$ with $\mathcal{E}_\theta$ and simulate the next step of the trajectory $\hat{o}_{t+1}$ with $\mathcal{P}_\varphi$. We use an aggregated loss to enforce consistency and invariance. This loss combines the negated discriminator loss with a quadratic consistency loss using a mixing hyper-parameter κ.

$$\mathcal{L}_{\text{total}} = \mathbb{E}_B\left[(o_{t+1} - \hat{o}_{t+1})^2\right] - \kappa\mathcal{L}_{\text{disc}}, \tag{7}$$

where the expectation is over the a sampled minibatch B from dataset D. Here, we used a quadratic loss function, but one could use any consistency loss fit to the specific type of variable (e.g. Huber loss, Cross entropy, ...).

Note the negative sign of discriminator loss, which means we train these NNs to maximize discriminator loss i.e., fool the discriminator to ensure policy invariance. If the extracted latent factors are policy invariant, the policy discriminator should do no better at its task than guessing at random.

Counterfactual estimation. To produce counterfactual estimates, as described above, the estimated latents $\hat{u}_t$ are extracted from observed data. Using the extracted latents factors, along with the learned combined function $\mathcal{P}_\gamma$, we start with o_1 and predict counterfactual observations $\hat{o}_{t+1}$, one step at a time.

6 Evaluation

We evaluate CausalSim's ability to do accurate counterfactual simulation (§6.1 and §6.3) using trace data from one real-world and one synthetic dataset. As a rigorous proof of concept, we debug and improve an ill-performing ABR policy with CausalSim (§6.2), and verify it through deployment on a public ABR testing infrastructure. Our baselines are as follows:

1. *ExpertSim*: Uses the analytical model described in §2.2.1.
2. *SLSim*: Uses a standard supervised-learning technique to learn system dynamics from data, as described in §2.2.2.

Finally, we show how CausalSim enables trace-driven simulation in problems where defining an *exogenous trace* is not straightforward and traditional trace-driven simulation is not applicable (§6.4). Further supporting experiments in the appendix provide more details about how CausalSim operates (§B.1, §B.2, §B.3, §B.4, §B.5, §B.7, §C.2, §C.3, §C.4 and §D.1).

6.1 Simulation Accuracy

We use CausalSim to predict the end performance of ABR policies, and compare them with ground truth data. We explore the same two metrics reported by Puffer to evaluate algorithms; 1) stall rate, which is the fraction of time a user spent rebuffering, i.e. paused and waiting for a new chunk to download; 2) average Structural Similarity Index Measure (SSIM) in decibels, which is a perceptual quality metric. Our ground truth data comes from public logs of 'slow streams' on Puffer. Whenever a client initiates a video streaming session in Puffer's website, a random ABR algorithm is chosen and assigned to that session. Sessions are logged (buffer levels, chunk sizes, timestamps, download times, etc) anonymously and the data is available for public use. Our dataset contains more than $230K$ trajectories from an RCT during July 2020 to June 2021, where five ABR algorithms (BBA, BOLA1, BOLA2, Fugu-CL, Fugu-2019) were evaluated. Exhaustive details of the setup and data can be found in §B.8.

6.1.1 Can CausalSim simulate a policy it has not seen?

We choose one of BBA, BOLA1, and BOLA2[11] as the new policy that we want to simulate, and call it the *target* policy. The remaining four policies are called *source* policies. Traces assigned to the four source policies comprise our training dataset, which we use for training CausalSim and the two baselines. The goal is to simulate the outcome of applying the target policy on trajectories assigned to any of the source policies.

Figure 4a plots the stall rate and SSIM in the simulated trajectories and ground truth, denoting each target policy with a different color. Four source policies give us four separate predictions per target policy and simulator. Each point depicts the average of these four predictions, and the intervals show the minimum and maximum among the four. For either metric, CausalSim is the most faithful to ground truth among all simulators. For instance, in stall rate, CausalSim's relative error spans $2-28\%$, while ExpertSim spans $49-68\%$ and SLSim spans $29-187\%$. CausalSim may not always predict the correct relative ordering among policies with close performance. For example, BOLA1 and BOLA2 (shown in orange and red) have similar performance in both stall rate and SSIM. CausalSim predicts that these policies are similar but it infers their relative ordering incorrectly. However, CausalSim avoids the large errors made by the baseline simulators. In absolute terms, its predictions are close to the ground truth.

CausalSim also has the most consistent predictions across different **source** policies, because it removes the biases of the source policies. As an example, we investigate all four simulation results for BOLA1 in Figure 4b. SLSim and ExpertSim's simulation results are only good when the source algorithm is BOLA2 (a similar algorithm to BOLA1 performance-wise). However, their predictions are far off from the ground truth for the other three source algorithms. CausalSim's simulation results, on the other hand, are all close to the ground truth target. Appendix §B.7 demonstrates the same observation for other target algorithms, i.e. BBA and BOLA2.

[11] We exclude Fugu as a test policy since we could not reproduce its logged actions (see §B.8).

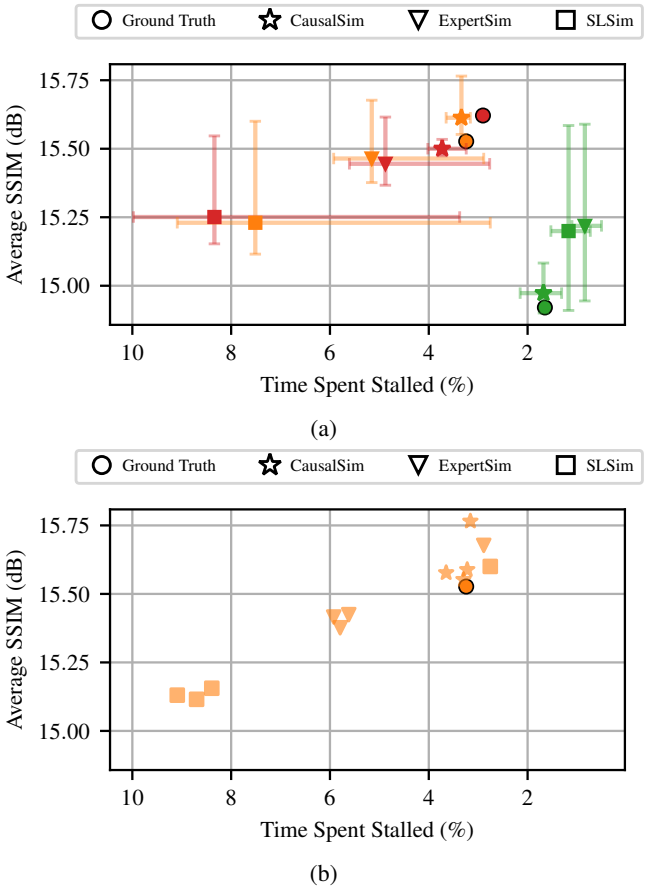


Figure 4: **(a)** In a real-world dataset of live video streaming, CausalSim is the most faithful, compared to traditional trace-driven (ExpertSim) or data-driven (SLSim) simulators. Colors indicate different **target** ABR algorithms. **(b)** Predictions for BOLA1, separated by the source policy. Each point indicates a different **source** ABR algorithm. ExpertSim and SLSim predictions carry over biases of the source data, while CausalSim mitigates the bias.

6.2 Case Study: CausalSim in the Wild

An accurate simulator allows researchers to debug and improve protocols without repeated and invasive deployments. We shall demonstrate this with CausalSim, by improving a well-known ABR policy, and verifying our findings with a real-world deployment on Puffer.

Recall that in the particular RCT we used in §6.1, five ABR algorithms (BBA, BOLA1, BOLA2, Fugu-CL, Fugu-2019) were evaluated. Figure 5 shows the result of this evaluation for BBA, BOLA1 and BOLA2, across 'slow streams'.[12] Similar to Figure 4a, the X-axis shows the stall rate, and the Y-axis is the average SSIM. BOLA1 exhibited 82% more rebuffering compared to BBA. A revised version of BOLA1, called BOLA2, was deployed alongside it, since the Puffer

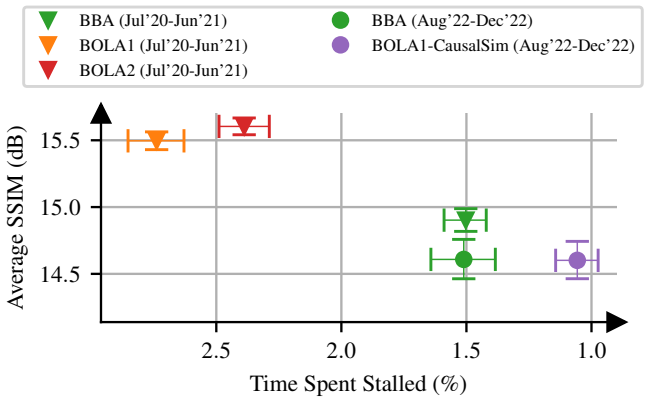


Figure 5: In an experiment preceding this work, BOLA1 exhibits high stalling. By deploying a BOLA1 variant in a later experiment **CausalSim improved the stall rate by** 2.6×, with comparable quality to BBA. User population is 'slow streams' and error bars denote 2.5%–97.5% confidence intervals.

team and the authors of BOLA believed the SSIM metric (in decibels) is incompatible with the protocol [53]. This new version had 12.8% less rebuffering and slightly higher quality, but still far too much stalling compared to BBA.

BOLA1 is an ABR policy with two hyperparameters, similar to BBA, and our hypothesis was that BOLA1 uses sub-optimal hyperparameters. To investigate this, we used the logged data pertaining to that plot along with CausalSim to exhaustively analyze the performance of BOLA1 and BBA for a range of hyperparameters. Using Bayesian Optimization[13], we explored the parameter space and created a Pareto frontier curve for each policy. During this process, we evaluated over 150 different algorithms in two days, which is achievable only in a simulator. Each curve demonstrates the trade-off between quality and stall rate in that policy. Figure 6 presents the curves, where the left and right plots show CausalSim and ExpertSim predictions. For ease of comparison, we highlight where the original BOLA1 and BBA lie. CausalSim confirms our suspicion; the curve for BOLA1 is strictly better than that of BBA. We can revise the hyperparameters in BOLA1 for an improved BOLA1 variant, henceforth called 'BOLA1-CausalSim'. We chose BOLA1-CausalSim, such that it would have better stall rate and marginally better SSIM compared to BBA.

Interestingly, ExpertSim predicts the complete opposite. It predicts that not only will BBA always improve on any BOLA1 variant in at least one metric, but also that any BOLA1 variant will stall more. This serves as a great opportunity to test CausalSim's edge compared to traditional (biased) trace-driven simulation, which is used in prior work [38,50,75]. The results of BOLA1-CausalSim's deployment can be seen in Figure 5. Considering confidence intervals, it is clear that it stalls less than BBA; in fact, BBA stalls 43% more than BOLA1-CausalSim on average. The confidence intervals for

[12] The data for this plot comes directly from Puffer [2, 3].

[13] We use a Gaussian Process prior with a Matern Kernel [54].

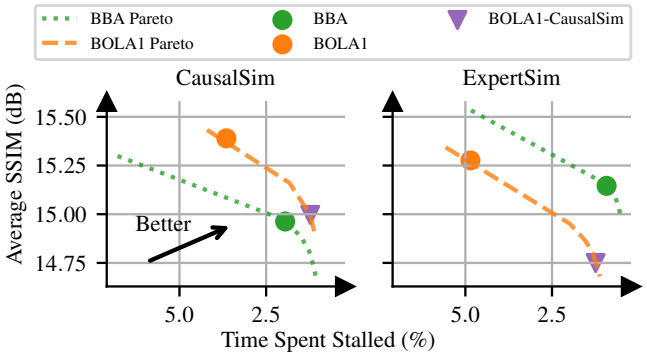


Figure 6: Pareto frontier curves for BOLA1 and BBA variants. **CausalSim correctly predicts BOLA1's potential**, while ExpertSim fails to do so.

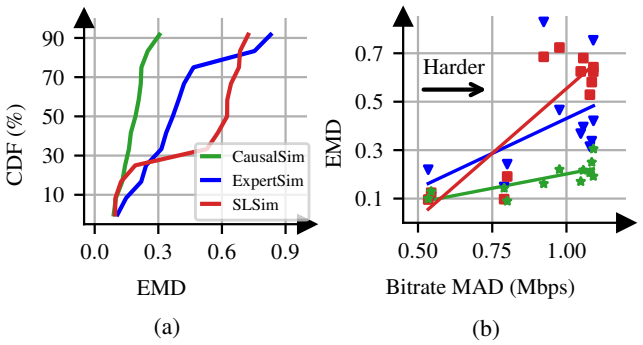


Figure 7: On average, CausalSim improves the EMD distance metric compared to ExpertSim and SLSim by 53% and 61% respectively. **(a)** Distribution of CausalSim, ExpertSim, and SLSim EMDs over all possible source/target choices. **(b)** Error (EMD) increases for baseline as simulation scenarios get harder, but CausalSim maintains good accuracy.

quality are wide and will need more data to be separable[14], but based on the ongoing trend, BOLA1-CausalSim will have similar quality compared to BBA.

Our goal was to show CausalSim's potential, and for that we targeted one of several plots on Puffer ('slow streams'). We could have chosen a different plot to optimize on, but it would not affect the takeaway. Note that our opportunities for deployment on Puffer are limited, as other researchers use Puffer as well; hence we only deployed one BOLA1 variant. Furthermore, we hoped to also compare CausalSim's prediction of stall rate and quality with the deployment results, but the client and network population has clearly changed; as shown in Figure 5, BBA achieves a different SSIM value for the two periods of time. Since CausalSim's predictions are based on data from the previous RCT, directly comparing the predicted values to results from the new RCT isn't meaningful. However, as our results show, the old RCT data allows us to compare different schemes. For example, CausalSim predicts BBA stalls 58% more than BOLA1-CausalSim on network distribution of the old RCT, which is reasonably close to the 43% observed in the new RCT (ignoring confidence intervals).

6.3 A Closer Look at Simulated Trajectories

For a deep dive in simulator accuracy, we focus on buffer occupancy level, a key indicator of ABR algorithm behavior. Ideally, we would like to compare simulated trajectories to ground truth. But this isn't possible using real trace data, since it requires us to have multiple traces of different policies running under the exact same underlying path conditions. To overcome this issue, we resort to distributional evaluation. Puffer data is collected in an RCT setting; hence the characteristics of network paths assigned to each policy is the same. If we accurately simulate the target policy on traces assigned to one of the source policies, the distribution of each variable (e.g. buffer level) must be similar in the simulated trajectory and ground truth trace assigned to the target policy. This motivates using distributional similarity as our performance metric.

To quantify the similarity of two distributions, we use the Earth Mover Distance (EMD) [62]. We can calculate EMD for one-dimensional distributions as $\text{EMD}(\mathcal{P}, Q) = \int_{-\infty}^{+\infty} |\mathcal{P}(x) - Q(x)| dx$, where $\mathcal{P}$ and Q are the Cumulative Distribution Function (CDF)s of p and q, respectively. A small EMD between two distributions implies that they are similar.

Figure 7a shows the CDF of the EMD (between actual and simulated buffer level distributions) for CausalSim and baselines, over all possible source/target policy pairs. EMD of CausalSim is smaller than EMD of baselines across almost all experiments. In terms of the average EMD across all experiments, CausalSim bests ExpertSim and SLSim by 53% and 61% respectively. Figure 2a visualized differences in buffer level distributions for the simulation scenario where BOLA2 and BBA are source and target policies, respectively. To observe buffer level distributions for all scenarios, refer to Figure 9.

In about 30% of cases, SLSim is slightly better than CausalSim. These cases are "easy" simulation scenarios where the source and target policies make similar actions (For more details see §B.3). In these cases, the EMD is low for both CausalSim and baseline simulators (< 0.15), and all perform well. For instance, Figure 9c (in the Appendix) shows source, target, and simulated buffer level distribution in an easy scenario, where BOLA2 and BOLA1 are the target and source policies respectively. In this example, all simulated distributions match the target distribution quite well.

Figure 7b shows where CausalSim most shines, i.e. hard simulation scenarios. The Y-axis is the error (EMD), and the X-axis is the mean absolute difference (MAD) between actions taken by the source policy and the target policy, in SLSim simulation. The larger the action difference, the harder the scenario (§B.3). As we move toward harder scenarios, the error increases

[14] Updated plots can be found on the 'Experimental Results' page of the Puffer website [1], under "Current experiment, full contiguous duration, slow streams only".

significantly for the baselines, while CausalSim is more robust.

6.3.1 Additional experiments

We perform further evaluations of CausalSim in the ABR environment. Due to space constraints, we summarize these results here and defer details to the appendix.

A more fine-grained evaluation. In the results above, we evaluated the performance of CausalSim and baselines using the distribution of buffer occupancy across the whole population. One way to further validate the results is to test whether they will hold on carefully partitioned sub-populations. In §B.4, we show that this is indeed the case when the sub-populations are partitioned according to the Min Round Trip Time (RTT), a network property that is independent of the selected ABR algorithm in Puffer.

Hyperparameters tuning. Counterfactual estimation (§3.2) is inherently an Out of Distribution (OOD) prediction task. Hence, typical supervised-learning hyper-parameter tuning methods do not work. In §B.5, we describe and evaluate CausalSim's hyper-parameter tuning procedure.

Ground truth evaluation. Real data never comes with ground truth counterfactual labels. As a result, we cannot evaluate CausalSim's simulations for each time step in real data, but we can do this in a reproducible synthetic environment. In §C.2, we evaluate CausalSim using ground truth counterfactual labels and show that it still outperforms baselines in the Mean Absolute Percentage Error (MAPE) metric.[15] Specfically, CausalSim achieves an MAPE of (∼5%), which is significantly lower than both ExpertSim's and SLSim's (∼10%).

6.4 A Second Example: Server Load Balancing

We now focus on simulating load balancing policies with heterogeneous servers, where defining an exogenous trace is not possible and therefore standard trace-driven simulation is not applicable. This example shows how CausalSim opens up new avenues in trace-driven simulation.

We use a synthetic environment which consists of $N = 8$ servers (and a queue for each) with different processing powers, a load balancer, and a series of jobs that need to be processed on these servers. Each job has a specific size which is unknown to the load balancer. Each server can process jobs at a specific rate $\{r_i\}_{i=1}^N$, which is also unknown to the load balancer. The load balancer receives jobs and must assign them to one of N servers. Assuming the kth arriving job has size S_k and gets assigned to server a_k, the job processing time will be S_k/r_{a_k}. If this job is not blocked by some other job being processed, its latency will equal its processing time. If it is blocked, and the jobs ahead of it in the queue take T_k to be processed, the incurred latency is $S_k/r_{a_k}+T_k$.

[15] Let $\hat{\mathbf{p}} = \{\hat{p}_i\}_{i=1}^N$ and $\mathbf{p} = \{p_i\}_{i=1}^N$ denotes the vectors of predicted and ground truth quantity of interest, respectively. Then, MAPE is defined as $\text{MAPE}(\mathbf{p},\hat{\mathbf{p}}) = \frac{100}{N}\sum_{i=1}^N \frac{|\hat{p}_i - p_i|}{p_i}$.

We generate a collection of 5000 trajectories each with 1000 steps and use 16 policies in the load balancer. For a detailed explanation of the policies, job size generation process, and server processing rates, refer to §D.2.

6.4.1 Experiment setup

The aim of this experiment is to evaluate whether we can simulate new unseen server assignment policies in this environment, using traces collected with other policies. Recall that while we observe the processing time of each job, the actual size of the job is not observed, i.e., it acts as the latent factor in this problem. For all simulators, we assume access to $\mathcal{F}_{\text{system}}$ (the queue model) and focus on the more challenging task of learning $\mathcal{F}_{\text{trace}}$ and estimating the counterfactual traces $\hat{m}_i^t$ for $i \leq 5000$, and $t \leq 1000$. Algorithmically, this translates to enforcing consistency for the observed traces (m_t), rather than the observations (o_t) (see §5). The trace we collect is the processing time when using a *source* server assignment policy. To simulate a *target* server assignment policy, we need to estimate the processing time of a job on servers other than the one where its processing time was measured (without knowing either the job size or the server processing rates).

Standard trace-driven simulation assumes an exogenous trace (job processing time), but this is the same as assuming servers have equal processing rates. This contradicts the problem setup, and standard trace-driven simulation (analogous to ExpertSim in ABR) is not applicable to this problem. Thus, we compare CausalSim with *SLSim* simulations. SLSim (realized by an NN) takes as input the observed processing time and the target server, and its output is the processing time under the targeted server. However, the observed and target processing time are always the same in training data, and hence it is impossible for SLSim to learn the true dynamics (e.g., the server's underlying processing power). CausalSim sidesteps this problem by explicitly estimating latent factors. For details regarding the network architecture and training details for both SLSim and CausalSim, refer to Table 8 in the appendix.

Performance Metric. We compare CausalSim and SLSim with the underlying ground truth using the MAPE metric.

6.4.2 Can CausalSim Faithfully Simulate New Policies?

As is done in the ABR case studies, we train CausalSim and SLSim models based on a dataset generated using all policies except one, which will be the target policy. We use the same hyper-parameter tuning approaches explained in §B.5 for CausalSim and §B.6 for SLSim. We carry out this evaluation on eight target policies. We evaluate the performance for each pair of source-target policies, as was done in §6.1. In total, we have 120 different source/target policy pairs.

In Figure 8a and Figure 8b, we show the CDF of the MAPE of estimating the processing time and the latency, respectively, using both CausalSim and SLSim. As evident in these two figures, CausalSim's error is significantly lower than that of

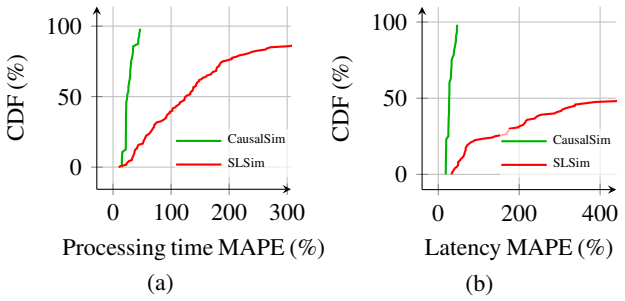


Figure 8: Distribution of CausalSim and SLSim MAPEs over all source target pairs.

SLSim for both the processing time and latency. In particular, the median MAPE when estimating processing time/latency is 24.4%/27.0% for CausalSim and 124.3%/467.8% for SLSim. For a complementary view, we compare the latent factors CausalSim extracts to the real latent job sizes and observe how closely they match, in §D.1 in the appendix.

7 Related-Work

Data-driven simulation. Traditional packet-level simulators [21, 31, 45] tend to sacrifice either scalability or accuracy when simulating large networks. MimicNet [77] and DeepQueueNet [73] use machine learning to improve simulation speed of datacenter networks. The aforementioned approaches are all full-system packet-level simulators, whereas CausalSim focuses on trace-driven simulation of a specific system component and must therefore deal with latent factors and biases present in trace data.

A very recent work, Veritas [17] (published on arXiv in Aug. 2022), models trace-driven simulation for ABR as a Hidden Markov Model (HMM) with a known emission process. This is equivalent to assuming that $\mathcal{F}_{\text{trace}}$ is known in our model (see Eq. (1)). Veritas uses the Viterbi algorithm to decode the latent factors, which are then used for counterfactual simulation. CausalSim solves a more general problem where $\mathcal{F}_{\text{trace}}$ is not known and must be learned. It therefore requires less knowledge of the system's latents and underlying dynamics to apply. On the other hand, CausalSim requires RCT data whereas Veritas does not. Comparing the fidelity of these approaches using real-world ABR data would be interesting future work (Veritas evaluates its method in a network emulator).

Panthon's calibrated emulators [72] model the end-to-end behaviour of a network path with a simple model including a handful of parameters, e.g., bottleneck link rate, constant propagation delay, etc., which are tuned to fit a collection of packet traces collected from this path using a variety of congestion control protocols. iBox [13] extends this approach by modeling cross-traffic. CausalSim does not assume any known model for the dynamics of the network. Furthermore, it has access to only a single trace from each network path.

Policy evaluation. Policy evaluation techniques such as Inverse Propensity Scoring [33] and Doubly Robust [15] aim to predict population-level performance statistics for a given intervention. WISE [67] builds a Causal Bayesian Network from the data that is able to answer interventional (what-if) queries about the future, but the method requires absence of latent confounding variables. Sage [25] uses a Causal Bayesian Network model with latent factors to diagnose performance issues in microservice applications. It answers what-if questions about how interventions like changing the resources allocated to a microservice impacts the end-to-end application latency. Trace-driven simulation is distinct from all these methods, in that it requires counterfactual predictions of how an intervention would have changed specific previously-measured trajectories rather than how it changes population-level statistics.[16]

8 Concluding Remarks

The exogenous trace assumption is central to traditional trace-driven simulation. CausalSim relaxes this key assumption, by modeling the intervention effect on the trace and learning to replay the trace in an unbiased manner. We showed how this improves the accuracy of trace-driven simulation using real-world ABR data, and how CausalSim provides insights for algorithm improvement that are in contrast with standard trace-driven simulators' predictions, which we validated in a real-world deployment. Furthermore, we showed how this expands the applicability of trace-driven simulation to problems where defining an exogenous trace is not possible by applying it to heterogeneous server load balancing. We believe CausalSim could be applied to many other system simulation tasks.

CausalSim opens up several interesting paths for future work. First, evaluating CausalSim in problems with a higher-dimensional latent factors would be interesting. Second, it is a natural next step to use CausalSim for more complex policy optimization methods, e.g., using reinforcement learning. Last, as discussed in §4.3, our theoretical analysis of CausalSim's approach, i.e. exploiting the policy invariance of latent factors distributions, is not tight, and improving it could potentially relax the assumptions of our analytical method.

9 Acknowledgement

We thank our shepherd Keith Winstein for in-depth suggestions, and our reviewers for insightful comments. We thank the Puffer team, specifically Emily Marx and Francis Y. Yan for providing us with the data we used in §6.1 and the algorithm deployment in §6.2. This work was supported by NSF grants 1751009 and 1955370, an award from the SystemsThatLearn@CSAIL program, and a gift from Intel as part of the MIT Data Systems and AI Lab (DSAIL). A. Alomar and D. Shah were supported in part by DSO-Singapore project, MIT-IBM project on Causal representation learning and NSF FODSI project.

[16] Appendix E provides a broader overview of the causal inference literature.

References

[1] Puffer: Experimental results. https://puffer.stanford.edu/results/. Accessed: 2023-2-22.

[2] Puffer: Total scheme statistics - decmeber 27th, 2022. https://storage.googleapis.com/puffer-data-release/2022-12-27T11_2022-12-28T11/duration_slow_scheme_stats_2022-12-27T11_2022-12-28T11.txt. Accessed: 2023-2-22.

[3] Puffer: Total scheme statistics - july 2nd, 2021. https://storage.googleapis.com/puffer-data-release/2021-06-01T11_2021-06-02T11/duration_slow_scheme_stats_2021-06-01T11_2021-06-02T11.txt. Accessed: 2023-2-22.

[4] A. Abadie, A. Diamond, and J. Hainmueller. Synthetic control methods for comparative case studies: Estimating the effect of californiaâs tobacco control program. *Journal of the American Statistical Association*, 2010.

[5] A. Abadie and J. Gardeazabal. The economic costs of conflict: A case study of the basque country. *American Economic Review*, 2003.

[6] Anish Agarwal, Abdullah Alomar, Varkey Alumootil, Devavrat Shah, Dennis Shen, Zhi Xu, and Cindy Yang. Persim: Data-efficient offline reinforcement learning with heterogeneous agents via personalized simulators. *arXiv preprint arXiv:2102.06961*, 2021.

[7] Anish Agarwal, Abdullah Alomar, and Devavrat Shah. On multivariate singular spectrum analysis. *arXiv e-prints*, pages arXiv–2006, 2020.

[8] Anish Agarwal, Munther A. Dahleh, Devavrat Shah, and Dennis Shen. Causal matrix completion. *ArXiv*, abs/2109.15154, 2021.

[9] Anish Agarwal, Devavrat Shah, and Dennis Shen. Synthetic interventions. *arXiv preprint arXiv:2006.07691*, 2021.

[10] Anish Agarwal, Devavrat Shah, Dennis Shen, and Dogyoon Song. On robustness of principal component regression. *Journal of the American Statistical Association*, 2021.

[11] Muhammad Amjad, Vishal Misra, Devavrat Shah, and Dennis Shen. Mrsc: Multi-dimensional robust synthetic control. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2), June 2019.

[12] Muhammad Amjad, Devavrat Shah, and Dennis Shen. Robust synthetic control. *Journal of Machine Learning Research*, 19(22):1–51, 2018.

[13] Sachin Ashok, Shubham Tiwari, Nagarajan Natarajan, Venkata N Padmanabhan, and Sundararajan Sellamanickam. Data-driven network path simulation with ibox. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–26, 2022.

[14] Susan Athey, Mohsen Bayati, Nikolay Doudchenko, Guido Imbens, and Khashayar Khosravi. Matrix completion methods for causal panel data models. *Journal of the American Statistical Association*, pages 1–15, 2021.

[15] Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 192–198, 2017.

[16] Vineet Bharti, Pankaj Kankar, Lokesh Setia, Gonca Gürsun, Anukool Lakhina, and Mark Crovella. Inferring invisible traffic. In *Proceedings of the 6th International COnference*, Co-NEXT '10, New York, NY, USA, 2010. Association for Computing Machinery.

[17] Chandan Bothra, Jianfei Gao, Sanjay Rao, and Bruno Ribeiro. Veritas: Answering causal queries from video streaming traces. arXiv/2208.12596, August 2022.

[18] Changxiao Cai, Gen Li, Yuejie Chi, H Vincent Poor, and Yuxin Chen. Subspace estimation from unbalanced and incomplete data matrices: $\ell_{2,\infty}$ statistical guarantees. *The Annals of Statistics*, 49(2):944–967, 2021.

[19] Emmanuel J Candès and Benjamin Recht. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, 9(6):717–772, 2009.

[20] Emmanuel J Candès and Terence Tao. The power of convex relaxation: Near-optimal matrix completion. *IEEE Transactions on Information Theory*, 56(5):2053–2080, 2010.

[21] Xinjie Chang. Network simulations with opnet. In *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1*, WSC '99, page 307–314, New York, NY, USA, 1999. Association for Computing Machinery.

[22] DASH Industry Form. Reference client 2.4.0, 2016.

[23] Rajeev H Dehejia and Sadek Wahba. Causal effects in nonexperimental studies: Reevaluating the evaluation of training programs. *Journal of the American statistical Association*, 94(448):1053–1062, 1999.

[24] Andrew Forney, Judea Pearl, and Elias Bareinboim. Counterfactual data-fusion for online reinforcement learners. In *International Conference on Machine Learning*, pages 1156–1164. PMLR, 2017.

[25] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 135–151, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Silvia Gandy, Benjamin Recht, and Isao Yamada. Tensor completion and low-n-rank tensor recovery via convex optimization. *Inverse problems*, 27(2):025010, 2011.

[27] Sahaj Garg, Vincent Perot, Nicole Limtiaco, Ankur Taly, Ed H Chi, and Alex Beutel. Counterfactual fairness in text classification through robustness. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pages 219–226, 2019.

[28] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[29] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[30] Ruocheng Guo, Lu Cheng, Jundong Li, P Richard Hahn, and Huan Liu. A survey of learning causality with data: Problems and methods. *ACM Computing Surveys (CSUR)*, 53(4):1–37, 2020.

[31] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527, 2008.

[32] Paul W Holland. Statistics and causal inference. *Journal of the American statistical Association*, 81(396):945–960, 1986.

[33] Daniel G Horvitz and Donovan J Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association*, 47(260):663–685, 1952.

[34] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, timid, and unstable: picking a video streaming rate is hard. In *Proceedings of the 2012 internet measurement conference*, pages 225–238, 2012.

[35] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 187–198, New York, NY, USA, 2014. Association for Computing Machinery.

[36] Guido W Imbens. Nonparametric estimation of average treatment effects under exogeneity: A review. *Review of Economics and statistics*, 86(1):4–29, 2004.

[37] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. Unleashing the potential of data-driven networking. In *International Conference on Communication Systems and Networks*, pages 110–126. Springer, 2017.

[38] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108, 2012.

[39] Maryia Kabanava, Holger Rauhut, and Ulrich Terstiege. On the minimal number of measurements in low-rank matrix recovery. In *2015 International Conference on Sampling Theory and Applications (SampTA)*, pages 382–386, 2015.

[40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[41] Daniel Kressner, Michael Steinlechner, and Bart Vandereycken. Low-rank tensor completion by riemannian optimization. *BIT Numerical Mathematics*, 54(2):447–468, 2014.

[42] S Shunmuga Krishnan and Ramesh K Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[43] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. *ACM SIGCOMM computer communication review*, 34(4):219–230, 2004.

[44] Anukool Lakhina, Konstantina Papagiannaki, Mark Crovella, Christophe Diot, Eric D. Kolaczyk, and Nina Taft. Structural analysis of network traffic flows. *SIGMETRICS Perform. Eval. Rev.*, 32(1):61–72, jun 2004.

[45] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[46] Yongjun Liao, Wei Du, Pierre Geurts, and Guy Leduc. Dmfsgd: A decentralized matrix factorization algorithm for network distance prediction. *IEEE/ACM Trans. Netw.*, 21(5):1511–1524, oct 2013.

[47] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[48] Ji Liu, Przemyslaw Musialski, Peter Wonka, and Jieping Ye. Tensor completion for estimating missing values in visual data. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):208–220, 2012.

[49] Dong Lu, Yi Qiao, P.A. Dinda, and F.E. Bustamante. Characterizing and predicting tcp throughput on the wide area network. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 414–424, 2005.

[50] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.

[51] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments, 2018.

[52] Yun Mao, Lawrence K. Saul, and Jonathan M. Smith. Ides: An internet distance estimation service for large networks. *IEEE Journal on Selected Areas in Communications*, 24(12):2273–2284, 2006.

[53] Emily Marx, Francis Y. Yan, and Keith Winstein. Implementing bola-basic on puffer: Lessons for the use of ssim in abr logic, 2020.

[54] Bertil Matérn. *Spatial variation*, volume 36. Springer Science & Business Media, 2013.

[55] Cross-Disorder Group of the Psychiatric Genomics Consortium et al. Identification of risk loci with shared effects on five major psychiatric disorders: a genome-wide analysis. *The Lancet*, 381(9875):1371–1379, 2013.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[57] Judea Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, USA, 2nd edition, 2009.

[58] Jonas Peters, Dominik Janzing, and Bernhard Schölkopf. *Elements of causal inference: foundations and learning algorithms*. The MIT Press, 2017.

[59] James M Robins, Miguel Angel Hernan, and Babette Brumback. Marginal structural models and causal inference in epidemiology, 2000.

[60] Matthew Roughan, Yin Zhang, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices (extended version). *IEEE/ACM Transactions on Networking*, 20(3):662–676, 2012.

[61] Donald B Rubin. Causal inference using potential outcomes: Design, modeling, decisions. *Journal of the American Statistical Association*, 100(469):322–331, 2005.

[62] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271)*, pages 59–66. IEEE, 1998.

[63] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[64] P. C. Sruthi, Sanjay Rao, and Bruno Ribeiro. Pitfalls of data-driven networking: A case study of latent causal confounders in video streaming. In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI '20, page 42–47, New York, NY, USA, 2020. Association for Computing Machinery.

[65] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 272–285, New York, NY, USA, 2016. Association for Computing Machinery.

[66] Liying Tang and Mark Crovella. Virtual landmarks for the internet. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, page 143–152, New York, NY, USA, 2003. Association for Computing Machinery.

[67] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering what-if deployment and configuration questions with wise. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 99–110, 2008.

[68] Eric Tzeng, Judy Hoffman, Kate Saenko, and Trevor Darrell. Adversarial discriminative domain adaptation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7167–7176, 2017.

[69] Madeleine Udell and Alex Townsend. Why are big data matrices approximately low rank? *SIAM Journal on Mathematics of Data Science*, 1(1):144–160, 2019.

[70] Zhiqiang Xu. The minimal measurement number for low-rank matrix recovery. *Applied and Computational Harmonic Analysis*, 44(2):497–508, 2018.

[71] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[72] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 731–743, 2018.

[73] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. Deepqueuenet: Towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 441–457, New York, NY, USA, 2022. Association for Computing Machinery.

[74] Yuzhe Yang, Guo Zhang, Dina Katabi, and Zhi Xu. Menet: Towards effective adversarial robustness with matrix estimation. *arXiv preprint arXiv:1905.11971*, 2019.

[75] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. *SIGCOMM Comput. Commun. Rev.*, 45(4):325–338, August 2015.

[76] Dong Zhang, Hanwang Zhang, Jinhui Tang, Xiansheng Hua, and Qianru Sun. Causal intervention for weakly-supervised semantic segmentation. *arXiv preprint arXiv:2009.12547*, 2020.

[77] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. Mimicnet: Fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 287–304, New York, NY, USA, 2021. Association for Computing Machinery.

[78] Zemin Zhang and Shuchin Aeron. Exact tensor completion using t-svd. *IEEE Transactions on Signal Processing*, 65(6):1511–1526, 2016.

Appendix A Tensor Completion with policy invariance

Here, we discuss a more generic version of the problem considered in §4.2 from the lens of tensor completion. Specifically, in §4 we considered the simplified setting where the trace was considered to be one-dimensional. Here, we shall consider higher dimensional traces. This, naturally suggests using the lens of Tensor instead of Matrix completion. We will also discuss how higher dimensional trace can enable recovery of more complex system dynamics or models compared to the simple solution we discussed in §4 for rank 1 setup.

Potential Outcomes Tensor. As considered in §4 let all possible actions be denoted as $[A] = \{1, \dots, A\}$ for some $A \geq 2$. Let the trace be of D dimension. As before, we have N trajectories of interest with trajectory $i \in [N]$ being of length $H_i \geq 1$ time steps. As before, let $U = \sum_{i=1}^{N} H_i$.

Consider an order-3 tensor M of dimension $A \times U \times D$, where $M = [m_{\alpha\beta\gamma} : \alpha \in [A], \beta \in [U], \gamma \in [D]]$ with $m_{\alpha\beta\gamma}$ corresponds to the γth co-ordinate of the D-dimensional trace corresponding to action $a_t = \alpha \in [A]$ when latent factor is $u_{i,t}$ with β corresponding to enumeration of (i,t) for some $i \in [N]$ and $t \leq H_i$. Recall that, as explained in Section 4, all possible $(i,t) : t \leq H_i, i \in [N]$ are mapped to an integer in $[U]$. We call this tensor M as the Potential Outcomes Tensor.

Indeed, if we know M completely, then we can answer the task of simulation or counterfactual estimation well since we will be able to estimate the mediator for each trajectory under a given possible sequence of counterfactual actions, and subsequently estimate the counterfactual observation (assuming we could learn the $\mathcal{F}_{\text{systems}}$).

We shall assume that there are $P \geq 1$ policies under which these traces where observed. In particular, each trajectory was observed under one of these P policies and the assignment of policy to the trajectory was done uniformly at random. Define $\Pi_p \subset [U]$ as collection of indices corresponding to trajectories $i \in [N]$ and their times $t \leq H_i$ where trajectory i was assigned policy p for $p \in [P]$. Let $U_p = |\Pi_p|$.

Tensor factorization, low CP-rank. The tensor M admits (not necessarily unique) factorization of the form: for any $\alpha \in [A], \beta \in [U], \gamma \in [D]$

$$m_{\alpha\beta\gamma} = \sum_{\ell=1}^{r} x_{\alpha\ell} y_{\beta\ell} z_{\gamma\ell}, \tag{8}$$

for some $r \geq 1$. For any tensor, such a factorization exits with r at most $\text{poly}(A,U,D)$.

*Assumption 1 (**low-rank factorization**).* We shall make an assumption that r is *small*, i.e. does not scale with A,U,D and specifically a small constant.

*Assumption 2 (**sufficient measurements**).* We shall assume that number of measurements per instance, D, is at least as large as the underlying rank r of the tensor M, i.e. $D \geq r$.

Distributional invariance and RCT. As before, we shall assume that the distribution of latent factors is the same across different policies due to random assignment of policies to trajectories in the setup of RCT. In the context of the tensor M, this corresponds to the distribution invariance of factors $y_{\beta\cdot} \in \mathbb{R}^r$ over $\beta \in \Pi_p$ for any $p \in [P]$. Concretely, for any $p \neq p' \in [P]$ and $\ell \in [r]$, we have

$$\frac{1}{U_p} \sum_{\beta \in \Pi_p} y_{\beta\ell} \approx \frac{1}{U_{p'}} \sum_{\beta' \in \Pi_{p'}} y_{\beta'\ell}. \tag{9}$$

More generally, any finite moment (not just first moment or average) of latent factors should be empirically invariant across policies. As in §4, we would like to utilize property (9) to estimate the tensor M.

A Simple Estimation Method and When It Works. We describe a simple method that can recover entire tensor as long as rank $r \leq D$. For simplicity, we shall assume $r = D$ (the largest possible rank for which method will work). By (8), for a given fixed $\alpha \in [A]$ and across $\beta \in [U], \gamma \in [D]$,

$$m_{\alpha\beta\gamma} = \sum_{\ell=1}^{r} y_{\beta\ell} \tilde{z}^{\alpha}_{\gamma\ell}, \tag{10}$$

where $\tilde{z}^{\alpha}_{\gamma\ell} = x_{\alpha\ell} z_{\gamma\ell}$. Since $D = r$, the matrix $\tilde{Z}^{\alpha} = [\tilde{z}^{\alpha}_{\gamma\ell} : \gamma \in [D], \ell \in [r]]$ is a square matrix. With this notation, we have that for any fixed $\alpha \in [A]$, the matrix $M^{\alpha} = [m_{\alpha\beta\gamma} : \beta \in [U], \gamma \in [D]] \in \mathbb{R}^{U \times D}$ (or $\mathbb{R}^{U \times r}$ since $r = D$) can be represented as

$$M^{\alpha} = Y \tilde{Z}^{\alpha,T}, \tag{11}$$

where $Y = [y_{\beta\ell} : \beta \in [U], \ell \in [r]] \in \mathbb{R}^{U \times r}$.

*Assumption 3 (**invertibility**).* We shall assume that the $D \times D$ (i.e. $r \times r$) square matrices $\tilde{Z}^{\alpha}$ for each $\alpha \in [A]$ are full rank and hence invertible.

The *Assumption 3* implies that $Y = M^{\alpha} \left(\tilde{Z}^{\alpha,T}\right)^{-1}$ for all $\alpha \in [A]$.

For policy $p \in [P]$, indices $\beta \in \Pi_p$ are relevant. For a given $\beta \in \Pi_p$, if the policy p utilized action $\alpha \in [A]$, $m_{\alpha\beta\cdot} \in \mathbb{R}^D$ is observed. To that end, let $\Pi_{p,\alpha} = \{\beta \in \Pi_p : \text{policy utilized action } \alpha\}$. Let $U_{p,\alpha} = |\Pi_{p,\alpha}|$ for any $\alpha \in [A]$. Then, define $Y^{p,\alpha} = [y_{\beta\ell} : \beta \in \Pi_{p,\alpha}, \ell \in [r]] \in \mathbb{R}^{U_{p,\alpha} \times r}$, $M^{\alpha,p} = [m_{\alpha\beta\gamma} : \beta \in \Pi_{p,\alpha}, \gamma \in [D]]$. Then we have $Y^{p,\alpha} = M^{\alpha,p} \left(\tilde{Z}^{\alpha,T}\right)^{-1}$.

Therefore, for any $\ell \in [r = D]$,

$$\begin{aligned} \sum_{\beta \in \Pi_{p,\alpha}} y_{\beta\ell} &= \mathbf{1}^{p,\alpha,T} Y^{p,\alpha} \mathbf{e}_{\ell} \\ &= \mathbf{e}_{\ell}^{T} Y^{p,\alpha,T} \mathbf{1}^{p,\alpha} \\ &= \mathbf{e}_{\ell}^{T} \left(\tilde{Z}^{\alpha}\right)^{-1} M^{\alpha,p,T} \mathbf{1}^{p,\alpha}, \end{aligned} \tag{12}$$

where $\mathbf{1}^{p,\alpha} \in \mathbb{R}^{U_{p,\alpha}}$ is vector of all 1s, and $\mathbf{e}_{\ell} \in \mathbb{R}^r$ be vector with all entries 0 but the $\ell \in [r]$th co-ordinate 1 .

Then, for any $\ell \in [r]$ and $p \in [P]$,

$$\begin{aligned}\frac{1}{U_p}\sum_{\beta\in\Pi_p} y_{\beta\ell} &= \frac{1}{U_p}\sum_{\alpha\in[A]}\sum_{\beta\in\Pi_{p,\alpha}} y_{\beta\ell} \\ &= \frac{1}{U_p}\sum_{\alpha\in[A]} \mathbf{e}_\ell^T \left(\tilde{Z}^\alpha\right)^{-1} M^{\alpha,p,T}\mathbf{1}^{p,\alpha} \\ &= \sum_{\alpha\in[A]} \mathbf{e}_\ell^T \left(\tilde{Z}^\alpha\right)^{-1}\left(\frac{1}{U_p}M^{\alpha,p,T}\mathbf{1}^{p,\alpha}\right) \\ &= \sum_{\alpha\in[A]} \mathbf{e}_\ell^T \left(\tilde{Z}^\alpha\right)^{-1}\mathcal{M}^{\alpha,p}, \end{aligned} \tag{13}$$

where $\mathcal{M}^{\alpha,p} = \frac{1}{U_p}M^{\alpha,p,T}\mathbf{1}^{p,\alpha} \in \mathbb{R}^{r,1}$ is an observed quantity, while $\tilde{Z}^{\alpha,T}$ is unknown. Using (13) and (9), we obtain that for any $\ell \in [r]$ and $p \neq p' \in [P]$,

$$\sum_{\alpha\in[A]} \mathbf{e}_\ell^T \left(\tilde{Z}^\alpha\right)^{-1}\mathcal{M}^{\alpha,p} \approx \sum_{\alpha\in[A]} \mathbf{e}_\ell^T \left(\tilde{Z}^\alpha\right)^{-1}\mathcal{M}^{\alpha,p'}. \tag{14}$$

Let $\tilde{z}^{\alpha,\ell} = \mathbf{e}_\ell^T \left(\tilde{Z}^\alpha\right)^{-1} \in \mathbb{R}^{1,r}$ be the ℓth row the of $r\times r$ matrix $\left(\tilde{Z}^\alpha\right)^{-1}$. Then (14) implies that for any $\ell \in [r]$ and $p \neq p' \in [P]$,

$$\sum_{\alpha\in[A]} \tilde{z}^{\alpha,\ell}(\mathcal{M}^{\alpha,p} - \mathcal{M}^{\alpha,p'}) \approx 0. \tag{15}$$

Which can be written in matrix form as

$$\begin{bmatrix}\tilde{z}^{1,\ell} & \tilde{z}^{2,\ell} & \dots & \tilde{z}^{A,\ell}\end{bmatrix}\begin{bmatrix}\mathcal{M}^{1,p}-\mathcal{M}^{1,p'} \\ \mathcal{M}^{2,p}-\mathcal{M}^{2,p'} \\ \vdots \\ \mathcal{M}^{A,p}-\mathcal{M}^{A,p'}\end{bmatrix} = 0 \tag{16}$$

By noting that that this hold for all $\ell \in [r]$, and recalling that $\tilde{z}^{\alpha,\ell}$ is the ℓ-th row the of the $r\times r$ matrix $\left(\tilde{Z}^\alpha\right)^{-1}$, we get,

$$\begin{bmatrix}\left(\tilde{Z}^1\right)^{-1} & \left(\tilde{Z}^2\right)^{-1} & \dots & \left(\tilde{Z}^A\right)^{-1}\end{bmatrix}\begin{bmatrix}\mathcal{M}^{1,p}-\mathcal{M}^{1,p'} \\ \mathcal{M}^{2,p}-\mathcal{M}^{2,p'} \\ \vdots \\ \mathcal{M}^{A,p}-\mathcal{M}^{A,p'}\end{bmatrix} = \mathbf{0}, \tag{17}$$

where $\mathbf{0}$ is a vector of zeros of size r. Note that the above is a system of r linear equations, with Ar^2 unknowns (recall that the $r\times r$ matrices $\left(\tilde{Z}^\alpha\right)^{-1}$ are unknown for $\alpha \in [A]$). Let $\mathbf{Z} \in \mathbb{R}^{r\times Ar}$ and $\mathbf{v}^{p,p'} \in \mathbb{R}^{Ar}$ denote the first and second matrix in the left hand side, respectively, then (17) can be re-written as,

$$\mathbf{Z}\mathbf{v}^{p,p'} \approx \mathbf{0}. \tag{18}$$

By definition, $\mathbf{v}^{p,p'}$ is observed quantity for each $p \neq p' \in [P]$. Now if we consider $P-1$ equations produced by considering pair of policies $(1,2),(1,3),\dots,(1,P)$ in (18), by design they are non-redundant linear equations. Let matrix $\mathbf{V} \in \mathbb{R}^{Ar\times P-1}$ be formed by stacking $\mathbf{v}^{1,2},\dots,\mathbf{v}^{1,P}$ column-wise.

Furthermore, let us define $\mathbf{s}^p \in \mathbb{R}^{Ar}$ as $[\mathcal{M}^{1,p},\cdots,\mathcal{M}^{A,p}]^\intercal$. Define $\mathbf{S} \in \mathbb{R}^{Ar\times P}$ by stacking $\mathbf{s}^1,\cdots,\mathbf{s}^P$ column-wise.

*Assumption 4 (**Sufficient, Diverse Policies**).* Let $P \geq Ar$ and the rank of $\mathbf{S} = Ar$.

Note that we can derive $\mathbf{V}$ from $\mathbf{S}$ by subtracting the first column from all other columns, and removing the first column. Thus, Under Assumption 4, the +rank of $\mathbf{V}$ is at least $Ar-1$. Further, given Assumption 3 which excludes the scenario $\mathbf{Z}=0$, it follows that the rank of $\mathbf{V}$ is $Ar-1$. As rank of $\mathbf{V}$ is $Ar-1$, we can uniquely (up to scaling) recover $\mathbf{Z}$ by solving for system of linear equation $\mathbf{ZV}=\mathbf{0}$ as the null space of $\mathbf{V}$ is of dimension 1.

Once we know $\mathbf{z}$, i.e. by undoing flattening, we obtain $\left(\tilde{Z}^{\alpha,T}\right)^{-1}$ for each $\alpha \in [A]$. Since for each policy $p \in [P]$ and $\alpha \in [A]$, $Y^{p,\alpha} = M^{\alpha,p}\left(\tilde{Z}^{\alpha,T}\right)^{-1}$ and we observe $M^{\alpha,p}$, we can recover $Y^{p,\alpha}$ and hence subsequently $Y \in \mathbb{R}^{U\times r}$.

By (11), we can now recover slice of tensor M, the M^α for each $\alpha \in [A]$, and hence we can recover entire tensor M as desired.

Interpretation of Assumption 4. Consider β^{th} Column of the matrix $\mathbf{S}$, i.e., $\left[\mathbb{E}[m^\intercal|i=1,\pi_\beta]\mathbb{P}(i=1|\pi_\beta),\cdots,\mathbb{E}[m^\intercal|i=A,\pi_\beta]\mathbb{P}(i=A|\pi_\beta)\right]^\intercal$ where i denotes the action index and β the policy index. This column is a vector of statistics associated with traces collected using policy β. Each element in this vector consists of two components: the first component is the conditional mean of the trace given a specific action, and the second element is the probability of taking this action. We interpret linear independence of each of these components for different policy vectors as policy diversity. For instance, think of the second component which captures probability vectors of different actions for each policy. Its linear independence across different policies roughly means that each policy should assign new probability vectors to different actions, and not a probability vector similar (linearly dependent) to that of previous policies. Also note that this assumption is not satisfied if an action is not taken by any of the policies which makes all elements of the corresponding row equal to zero.

Appendix B Real-world ABR

B.1 Comprehensive results

In Figure 7a, we presented a concise view of simulator fidelity, for an internal variable in ABR sessions called buffer occupancy level. Specifically, we considered the simulation of a target policy, given trajectories collected using a different source policy. We measured the error between buffer simulations and ground truth through EMD, a similarity index for distributions. For a complementary view, we provide the full distributions in Figure 9, for all simulators and ground truth for target and source policies. Below each plot, we also report the EMD of CausalSim predictions.

B.2 Policy Discriminator and Latent Invariance

The policy discriminator ($\mathcal{W}_\gamma$ in Figure 3) described in §5 has the goal of predicting the source policy, given a latent factor generated by the latent factor extractor ($\mathcal{E}_\theta$ in Figure 3). Since our data is collected with an RCT, the true latent factor distribution should be indifferent to the source policy. Therefore, if the latent factor extractor generates the ground truth latent factors, the policy discriminator should not be able to predict the source policy accurately. In fact, even the optimal policy discriminator outputs the population share of each source policy (e.g. what fraction of the data comes from BBA) in the training data [28]. To assess this statement, we present the confusion matrix and population share of source data, for three left-out policies in Table 1. Each row corresponds to one source policy, and each column corresponds to the policy discriminator's prediction of the source policy. We observe that predictions do not change noticeably with different source policies, and that they closely match the population share for each left-out policy. This demonstrates that the extracted latent features were indeed invariant to the source policy.

B.3 What makes a simulation scenario easy/hard?

In §6.3, we compared the accuracy of CausalSim, ExpertSim and SLSim, in a simulation task on real ABR data. We observed that in about 30% of scenarios, which we call *easy* scenarios, all simulators perform well. However, in about 70% of the source/target scenarios, which we call *hard* simulation scenarios, baseline predictions are highly biased towards the source distributions. In these hard scenarios, CausalSim is able to de-bias the trajectories and its predictions match the target distribution well, as observable in Figure 9.

So it is natural to wonder what makes a simulation scenario easy/hard? An easy simulation scenario happens when source and target policies take similar actions. Similar action means that the factual achieved throughput (of the source policy) is similar to the counterfactual achieved throughput (of the target policy). This is what both ExpertSim (explicitly) and SLSim (implicitly) assume for doing simulation. Making this assumption is the core reason their simulations are biased in hard cases, where source and target policies take different actions, as we discussed in detail in §2.2.3.

Figure 10 validates our reasoning for what makes a simulation scenario difficult. The X axis shows the Mean Absolute Difference (MAD) between source and simulation actions (bitrates) when simulating with SLSim in a specific

	Prediction			
Source Policy	BOLA2	BOLA1	Fugu-CL	Fugu-2019
BOLA2	22.44%	22.58%	26.99%	27.99%
BOLA1	22.43%	22.58%	26.99%	27.99%
Fugu-CL	22.44%	22.58%	26.99%	27.99%
Fugu-2019	22.44%	22.58%	26.99%	28.00%
	Source Policy			
	BOLA2	BOLA1	Fugu-CL	Fugu-2019
Population	22.45%	22.50%	27.11%	27.94%

(a) Left-out policy is BBA

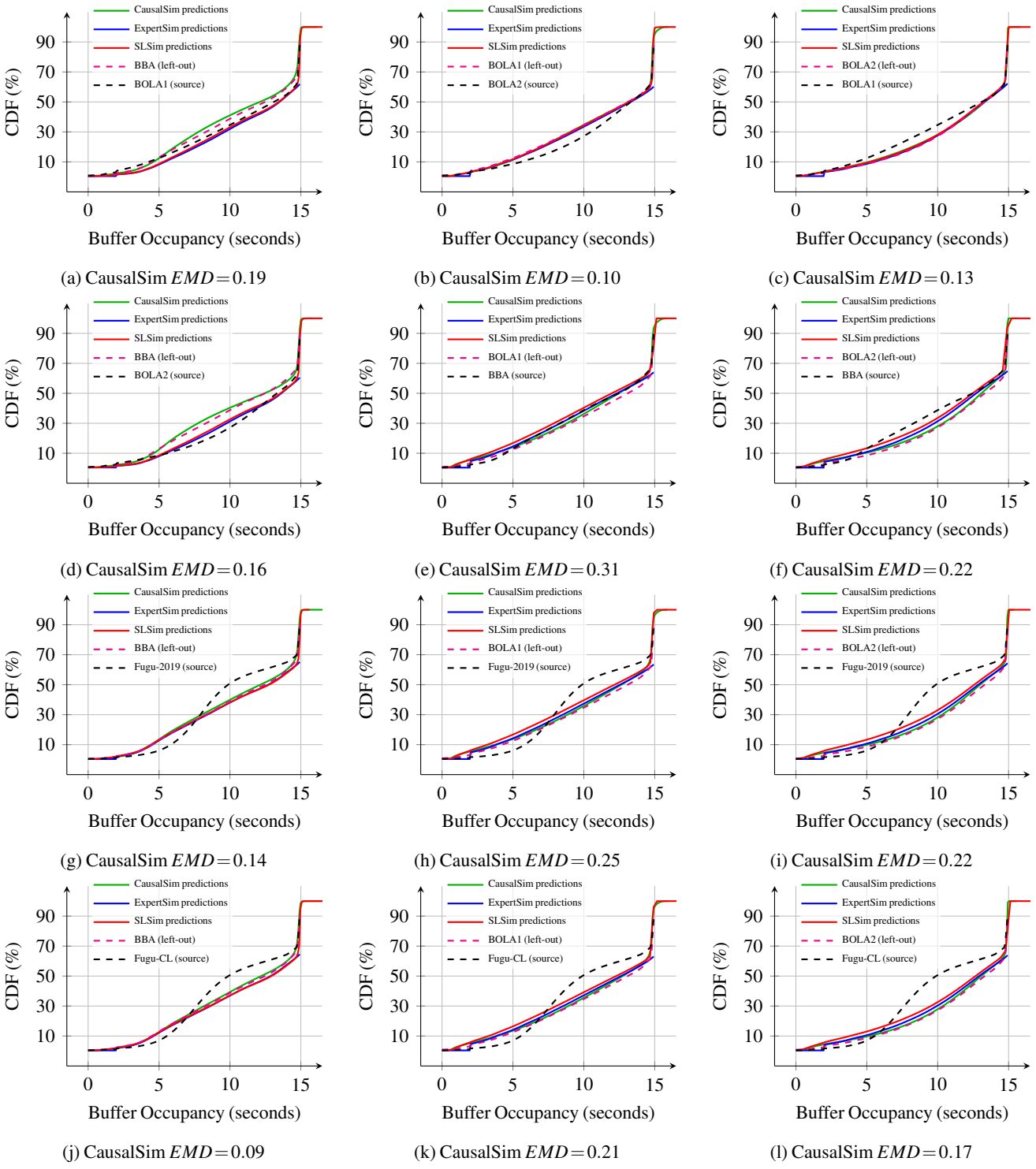
	Predictions			
Source Policy	BOLA2	Fugu-CL	Fugu-2019	BBA
BOLA2	21.34%	26.04%	26.75%	25.87%
Fugu-CL	21.33%	26.05%	26.75%	25.87%
Fugu-2019	21.33%	26.04%	26.77%	25.86%
BBA	21.33%	26.04%	26.76%	25.87%
	Source Policy			
	BOLA2	Fugu-CL	Fugu-2019	BBA
Population	21.48%	25.94%	26.74%	25.84%

(b) Left-out policy is BOLA1

	Predictions			
Source Policy	BOLA1	Fugu-CL	Fugu-2019	BBA
BOLA1	21.46%	26.00%	26.76%	25.78%
Fugu-CL	21.45%	26.01%	26.77%	25.76%
Fugu-2019	21.45%	26.00%	26.79%	25.76%
BBA	21.45%	25.99%	26.76%	25.80%
	Source Policy			
	BOLA1	Fugu-CL	Fugu-2019	BBA
Population	21.52%	25.93%	26.72%	25.83%

(c) Left-out policy is BOLA2

Table 1: Confusion matrix and population statistics for the policy discriminator with three left out policies.



(a) CausalSim $EMD = 0.19$ (b) CausalSim $EMD = 0.10$ (c) CausalSim $EMD = 0.13$

(d) CausalSim $EMD = 0.16$ (e) CausalSim $EMD = 0.31$ (f) CausalSim $EMD = 0.22$

(g) CausalSim $EMD = 0.14$ (h) CausalSim $EMD = 0.25$ (i) CausalSim $EMD = 0.22$

(j) CausalSim $EMD = 0.09$ (k) CausalSim $EMD = 0.21$ (l) CausalSim $EMD = 0.17$

Figure 9: Buffer level distribution of source, target, CausalSim predictions, and baseline predictions across all source/target scenarios.

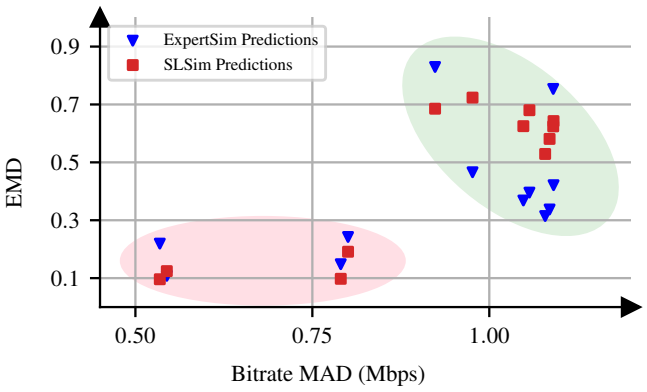


Figure 10: Simulation difficulty is related to how different counterfactual actions are from factual ones. This figure shows scatterplot of EMD versus mean absolute bitrate difference, for ExpertSim and SLSim, over all possible source left-out pairs. The pink cluster signifies the 'easy' scenarios and the green cluster signifies 'hard' ones.

source/target scenario. Y axis shows EMD (Our performance metric for simulation, smaller is better) of both baselines in that specific scenario.

Two main cluster of points are clearly visible in this figure. The pink cluster on the bottom left corresponds to easy simulations. It includes all source/target simulation scenarios where baselines perform well (bottom), and at the same time, source and target actions are quite similar (left).

The green cluster at the top right corresponds to the hard simulations. It includes all source/target simulation scenarios where baselines fail to perform an unbiased simulation (top), and at the same time, source and target actions are quite different (right).

B.4 A More Fine-grained Evaluation

Ideally, we would like to evaluate CausalSim's simulation to ground truth on a step-by-step basis for a given trajectory. But as discussed in §6.3, this is not possible in real-world data, as we only see the outcome of one ABR algorithm's chosen action for a single step. In other words, there is no way to get ground truth for individual steps in the observational data, which is referred to as the fundamental problem of Causal Inference [32]. This is the reason we evaluated predictions on a distributional level.

However, there is a way to evaluate CausalSim's predictions at a more fine-grained level. Instead of evaluating the predicted distribution of buffer occupancy across the whole population, we can evaluate on certain *sub-populations* of users. The only requirement is that the way we select these sub-populations should be statistically independent of the ABR algorithm. For example, we can partition users by a metric such as Min RTT, which is independent of the policy chosen for each user in the RCT. Min RTT is an inherent property of a network path[17], and we would expect Min RTT distribution to be the same for users assigned to different ABR policies.

We use the MinRTT to create the following four sub-populations:

1. Sub1: users with Min RTT $< 35^{ms}$
2. Sub2: users with $35^{ms} \leq$ Min RTT $< 70^{ms}$
3. Sub3: users with $70^{ms} \leq$ Min RTT $< 100^{ms}$
4. Sub4: users with $100^{ms} \leq$ Min RTT

Now, we can ask question of the following type: *had the users in sub-population two, who were assigned the source ABR algorithm, instead used the left-out ABR algorithm, what would the distribution of their buffer level look like?* As the ground truth answer to this question, we can use the buffer level distribution of users in sub-population two assigned to the left-out policy.

Figure 11a shows the CDF of CausalSim's EMD when simulating the left-out ABR algorithm over each of the above sub-populations. We can see that CausalSim maintains a superior EMD CDF compared to ExpertSim and SLSim, and remains accurate across different sub-populations. This further suggests that even at surgically small subpopulations, CausalSim maintains accuracy, and does not overfit to the whole distribution.

B.5 How to Tune CausalSim's Hyper-parameters?

Counterfactual prediction is not a standard supervised learning task that optimizes in-distribution generalization. Rather, it is always an OOD generalization problem, i.e., we collect data from a training policy (distribution 1), and want to accurately simulate data under a different policy (distribution 2). Since we do not use data from the test policy when we train CausalSim, we use the following natural proxy for tuning hyper-parameters: *Simulating ABR algorithms in the training data using trajectories of other ABR algorithms in the training data.* This of course can be viewed as an OOD problem as well. We claim that if a choice of hyper-parameters results in a robust model that performs well OOD across all validation ABR algorithms in the training data, it should work well for the actual left-out test policy as well.

We verify this hyper-parameter tuning procedure empirically. For each choice of the three left-out ABR algorithms (hence training dataset), we train eleven different CausalSim models with different choices of κ (defined in Equation (7)). We consider two metrics: *(i) Test EMD*, defined as the average EMD when simulating the left-out ABR algorithm with trajectories in the training dataset. This is our main performance objective. *(ii) Validation EMD*, defined as the average EMD when

[17] This is true to a first order approximation, if we ignore the possibility that a video streaming session drives up queueing delays throughout the course of a video, thereby inflating the observed Min RTT.

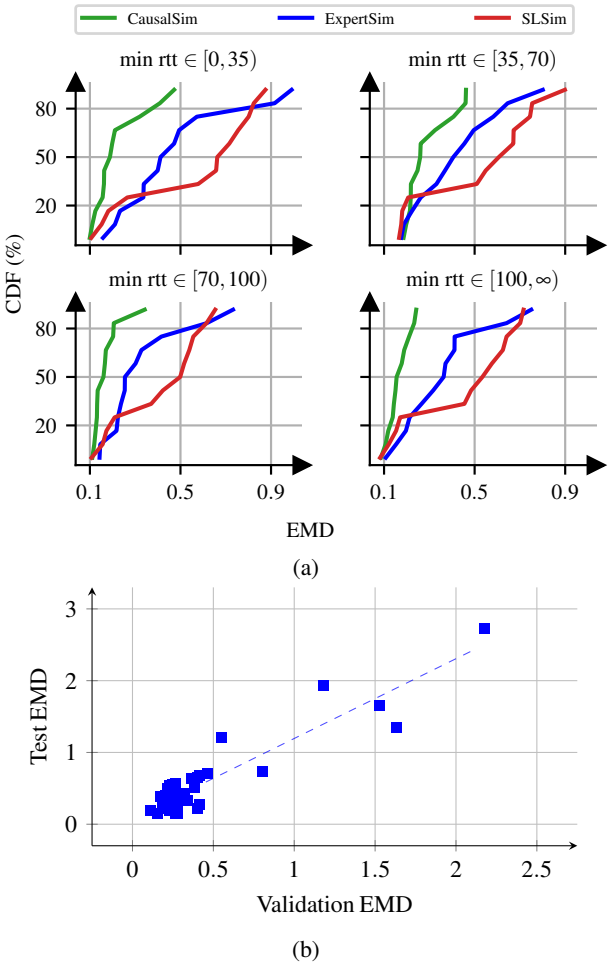


Figure 11: **(a)** Comparing the distribution of CausalSim EMDs with ExpertSim and SLSim over different sub-populations. **(b)** Validation EMD and test EMD are highly correlated. This justifies our hyper-parameter tuning strategy.

simulating ABR algorithms in the training dataset with trajectories in the training data that were collected with other ABR algorithms. This is our proxy objective for hyper-parameter tuning.

For each model (33 in all: 3 datasets, 11 example hyper-parameters), we calculate both Test EMD and Validation EMD, which results in one (Validation EMD, Test EMD) point in Figure 11b. The Pearson Correlation Coefficient (PCC) between Valid EMD and Test EMD is 0.92, which shows high linear correlation. Hence, though CausalSim might not always perform well (i.e., Test EMD is not low for some combinations of training dataset and hyper-parameters), we can have a very good idea of how well it works by measuring Validation EMD.

B.6 How to Tune SLSim's Hyper-parameters?

SLSim takes as input the current buffer value, selected chunk size and observed throughput, and similar to CausalSim, predicts the next buffer $\hat{b}_{t+1}$ and download time $\hat{d}_t$. We add two knobs to tune while training SLSim: **(1)** The loss function $\mathcal{L}_\xi(\cdot,\cdot)$ used to steer the NN output to the ground truth output, and **(2)** The relative weighting of the loss function for download time with respect to that of the buffer occupancy, η. Concretely, we use the following total loss:

$$\mathcal{L}_{\text{slsim}} = \mathbb{E}_B\left[\frac{1}{\eta+1}.\mathcal{L}_\xi(\hat{b}_{t+1},b_{t+1}) + \frac{\eta}{\eta+1}.\mathcal{L}_\xi(\hat{d}_t,d_t)\right] \quad (19)$$

where the expectation is over the a sampled minibatch B from dataset D, and b_{t+1} and d_t denote the ground truth values for next buffer level and chunk download time. Table 3 lists the loss functions and η values considered.

To tune these values, we use ground truth data from all policies except a left out policy. We then proceed with the proxy tuning objective used in §B.5, i.e. we look for the configuration with the highest accuracy at simulating algorithms in the training data using trajectories of other algorithms in the training data. We then use the resulting configuration (and model) to simulate the left-out policy on the training data.

From the perspective of tuning, this methodology puts SLSim on equal ground with respect to CausalSim, and makes for a fair comparison. Note that we do not tune loss function type or η with CausalSim due to limited computational resources, but tuning those as well could potentially improve CausalSim's accuracy.

B.7 Simulation Accuracy: Continued

In §6.1.1, we stated that ExpertSim and SLSim predictions are significantly affected by the source data they are simulating on, and demonstrated the effect of source policies on BOLA1 predictions in Figure 4b. Here, we demonstrate the same figure for BBA in Figure 12a and BOLA2 in Figure 12b. CausalSim is designed to remove the bias of the algorithm used for collecting source data when simulating a target policy and its predictions remains unaffected by the performance of that source policy. ExpertSim and SLSim however, due to the violation of the exogenous trace assumption, will predict different metrics when using different source traces.

B.8 Dataset & Algorithms

Our trajectories in the real-world (Puffer) data come from 'slow streams' in the time span of July 27, 2020 until June 2, 2021. In this period of time, 5 ABR algorithms appear consistently and are listed in Table 2. Each trajectory is an active client session streaming a live TV channel. We follow Puffer's definition of

Policies	Hyperparameter	Value	Used as source	Used as left out
BBA	Cushion	3 (as used in puffer)	✓	✓
	Reservoir	10.5 (as used in puffer)		
BOLA-BASIC v1	V	0.67 (As computed in puffer)	✓	✓
	γ	-0.43 (As computed in puffer)		
	Utility function	$\log_{10}(1-ssim)$ (As used in puffer)		
	Minimum utility	0 dB (As used in puffer)		
	Maximum utility	60 dB (As used in puffer)		
BOLA-BASIC v2	V	51.4 (As computed in puffer)	✓	✓
	γ	-0.43 (As computed in puffer)		
	Utility function	*ssim* (As used in puffer)		
	Minimum utility	0 (As used in puffer)		
	Maximum utility	1 (As used in puffer)		
Fugu-CL	-	-	✓	×
Fugu-2019	-	-	✓	×

Table 2: ABR algorithms used in the real-world dataset and experiments

'slow streams'; streams with TCP delivery rates below 6 Mbps. We use 'slow streams' data, since the highest quality chunks rarely surpass $6-7$ Mbps, and paths with higher bandwidth will always stream the highest quality chunks under all policies. Puffer uses the same reasoning and evaluates algorithms at two population levels; 'slow streams' and 'all streams'.

In aggregating 'slow stream' logs, we met several difficulties that we outline here for reproducibility. Data without these difficulties would potentially improve CausalSim's accuracy. Note that this does not affect Figure 5, as the data for that figure is reported directly on Puffer [2, 3].

Puffer logs are reported as three separate event groups; 1) 'video_sent': the first packet of a chunk is sent, 2) 'video_acked': The last packet of a chunk is acknowledged, 3) 'client': The client sent a message. Stall rate is computed using the 'client' logs and quality is computed using the 'video_sent' logs.

1. To compute download time, we have to merge 'video_sent' and 'video_acked', and ensure that merged logs are consecutive in timestamps, i.e. no chunk is missing in between two other chunks. However, in the current data this removes all chunks that have been sent but not acknowledged, usually the last chunk. Puffer uses these chunks in measuring quality level, but we can't. This did not have any measurable impact, however.

2. To compute stall rate, both total stall time and total watch time are computed with 'client' logs. For this, the latest report that obeys a set of rules is used. We, however, have to compute stall time and watch time using our merged logs (merged logs are also what we get out of simulation). This would be easy on the original data, if 'client' logs and 'video_sent' were in sync, but they are not; whenever a rebuffering is reported by the client, 'client' log is updated but 'video_sent' is updated in the next few chunks. To circumvent this, we recompute rebuffering as $t_r = max(0, t_d - b)$, where t_r is rebuffering, b is buffer occupancy and t_d is download time. This formula is off by half of an RTT, and empirically inflates stall rates by $1.26-1.31x$, for all policies. In the absence of synchronized data, this is the best we can recover, but it does not affect the comparison among policies. Hence, we believe simulating with this data should lead to similar trends as with clean unperturbed data.

3. We cannot calculate watch time as Puffer does, since we have to use the merged log. We tried several simple formulas that should calculate watch time, but oddly most turn out to be inaccurate. One reason is that in some streams, buffer playback rate is not 1, i.e. one second of buffer is not depleted per second. These streams are likely due to browser tabs put in background, and throttled by the browser threading system. As a workaround, we use the original watch time minus the original stall time that Puffer computed for a stream, and offset it by the total stall time in the simulation.

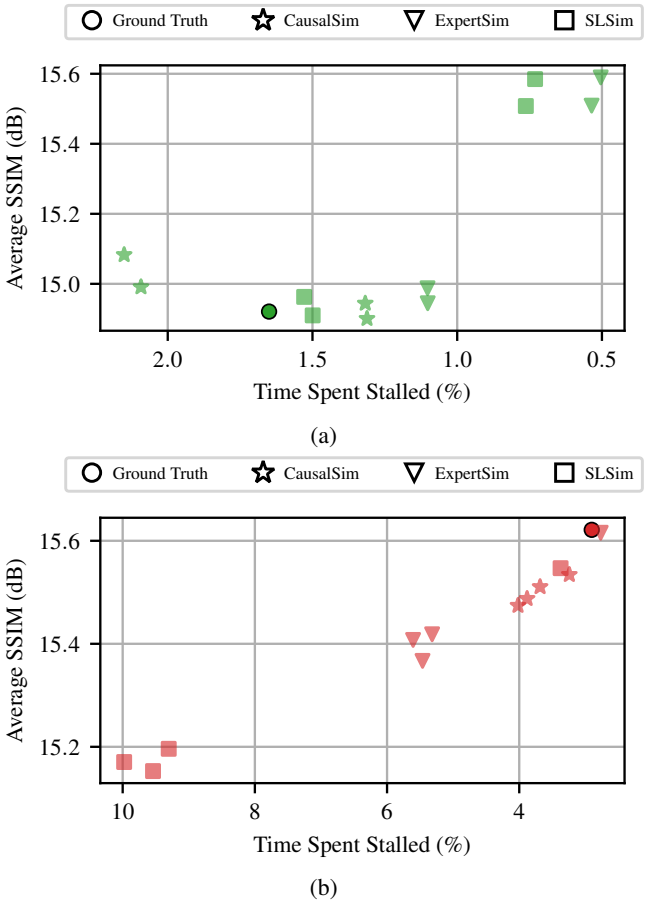


Figure 12: Predictions for **(a)** BBA and **(b)** BOLA2, separated by the ABR algorithm source data was collected with. Each point indicates a specific **source** ABR algorithm.

4. At each step, the buffer should not increase by more than a single chunk, 2.002 seconds, but it does (sometimes by as much as 14 seconds). We filter such data out.

5. When we are about to send a chunk, our last reported buffer value must never dip below 2.002 (except in the beginning). When buffer is below 15 seconds, the next chunk must be sent immediately after the last one. If rebuffering occurs, the next buffer value will be exactly 2.002 and if it doesn't, it will be larger than 2.002. We frequently (more than one million instances) observe buffer values below 2.002. We do not filter them out, as this would invalidate most logs.

To test out CausalSim, we need to simulate the streaming session using a different algorithm than the one that was actually used in that session. This requires implementation of the ABR algorithms. To ensure our implementations are correct, we attempt to reconstruct the choices made at runtime by each policy, and compare them to the logged choices. We expect our reproduction to match 100% when our implementation is faithful and logs match runtime inputs. For the logs in July 27th, 2020, we observe 100% matching for BOLA1 and BOLA2 and 99.993% for BBA. For the latter, there are rare cases where two encodings are seemingly equal in SSIM up to the 6 logged decimal places, but were likely slightly different in double precision format at runtime. These instances are rare enough that we can ignore them.

For Fugu-2019 or Fugu-CL however, our reproductions did not match in 6% and 19% of cases, whether we used the original C implementation or our own Python port. The Puffer team informed us of a use-after-free issue regarding the Transmission Control Protocol (TCP) info struct that was fixed in March 7th, 2022. Hence we retried this process for the logs pertaining to July 27th, 2022 and the error rate shrank to 0.53% and 0.64%. Unfortunately, a 0.5% error rate is still too high and even if we ignore that, limits us to RCT logs after March 7th. Therefore, we do not consider Fugu-2019 or Fugu-CL as candidates for left-out algorithms.

B.9 Training setup

We use Multi Layer Perceptrons (MLPs) as the NN structures for CausalSim models and the SLSim model. All implementations use the Pytorch [56] library. Table 3 is a comprehensive list of all hyperparameters used in training.

Appendix C Synthetic ABR

As explained in §6.3.1, we also evaluate CausalSim in a synthetic ABR environment, in which we can obtain ground truth for individual counterfactual predictions on a step-by-step basis for a trajectory. In these experiments, we also use a larger set of policies than available in the real data.

C.1 Simulation Dynamics

In each simulated training session, we start with an empty playback buffer and a latent network path characterized by an RTT and a capacity trace. In each step, an ABR algorithm chooses a chunk size, which is transported over this network path to the client as the buffer is depleting. Once the user receives the chunk, the buffer level increases by the chunk duration. This simple system can be modeled as follows:

$$b_{t+1} = min(b_t - d_t, 0) + c \tag{20}$$

where b_t, d_t and c refer to the buffer level at time step t, the download time of the chunk at time step t, and the chunk video length in seconds, respectively. Streaming the next chunk is started immediately following receiving the previous one, except when the buffer level surpasses a certain value (in our case, 10 seconds to mimic a live-stream ABR setting). To compute d_t, we model the transport as a TCP session with an Additive Increase - Multiplicative Decrease (AIMD)

Model	Hyperparameter	Value
SLSim (1 network), **CausalSim** (3 networks)	Hidden layers	(128, 128)
	Hidden layer Activation function	Rectified Linear Unit (ReLU)
	Output layer Activation function	Identity mapping
	Optimizer	Adam [40]
	Learning rate	0.001
	β_1	0.9
	β_2	0.999
	ε	10^{-8}
	Batch size	2^{17}
CausalSim	κ	{0.05, 0.1, 0.5, 1, 5, 10, 15, 20 ,25, 30, 40}
	Training iterations (num_train_it)	5000
	num_disc_it	10
	Loss function	Huber($\delta=0.2$)
	η (download time weight wrt buffer)	1
SLSim	Training iterations	10000
	Loss function	{Huber($\delta=0.2$), L1, MSE}
	η (download time weight wrt buffer)	{0.5, 1, 10}

Table 3: Training setup and hyperparameters for the real-world ABR experiment

congestion control mechanism with slow start. For every chunk, the TCP connection starts from the minimum window size of 2 packets and increases the window according to slow start. Therefore, it takes the transport some time to begin fully utilizing the available network capacity. The overhead incurred by slow start depends on the RTT and bandwidth-delay product of the path. When downloading chunks with large sizes, the probing overhead is minimal but it can be significant for small chunks. Therefore, as we observed in the Puffer data, the throughput achieved for a given chunk in this synthetic simulation depends on the size of the chunk.

Performance Metric: We compare CausalSim predictions with ground truth counterfactual trajectories, via the Mean Squared Error (MSE) distance between the two time series:

$$MSE(\mathbf{p},\mathbf{q}) = ||\mathbf{p}-\mathbf{q}||_2^2 \tag{21}$$

Here, $\mathbf{p} = \{p_t\}_{t=1}^N$ and $\mathbf{q} = \{q_t\}_{t=1}^N$ are time series vectors. Better predictions yield smaller MSE values, where an ideal MSE is 0.

C.1.1 Data & Algorithms

Simulating a trajectory in our synthetic ABR environment needs three components:

- A video, with several bit-rates available. We use "Envivio-Dash3" from the DASH-246 JavaScript reference client [22].
- An ABR algorithm. We have a set of 9 policies to choose from, presented in Table 4.
- A network path, which is characterized by the latent network capacity and the path RTT.

We use random generative processes to generate 5000 network traces and RTTs. The RTT for a streaming session is sampled randomly, according to a uniform distribution:

$$rtt \sim Unif(10\,ms, 500\,ms)$$

Our trace generator is a bounded Gaussian distribution, whose mean comes from a Markov chain. Prior work shows Markov chains are appropriate models for TCP throughput [65], and Gaussian distributions can model throughputs in stationary segments of TCP flows [49].

Concretely, at the start of the trace, the following parameters

Policies	Hyperparameter	Value	Used as source	Used as left out
BBA	Cushion	5	✓	✓
	Reservoir	10		
BOLA-BASIC	V	0.71 (Computed using puffer formula)	✓	✓
	γ	0.22 (Computed using puffer formula)		
	Utility function	ln(chunk sizes) (As used in BOLA paper [63])		
Random	-	-	✓	✓
BBA-Random mixture 1	Cushion	5	✓	✓
	Reservoir	10		
	Random choices	50%		
BBA-Random mixture 2	Cushion	10	✓	✓
	Reservoir	20		
	Random choices	50%		
MPC	Lookback length	5	✓	✓
	Lookahead length	5		
	Rebuffer penalty	4.3		
	Throughput estimate	Harmonic mean		
Rate-based	Lookback length	5	✓	✓
	Throughput estimate	Harmonic mean		
Optimistic Rate-based	Lookback length	5	✓	✓
	Throughput estimate	Max		
Pessimistic Rate-based	Lookback length	5	✓	✓
	Throughput estimate	Min		

Table 4: ABR algorithms used in the synthetic ABR experiments.

are randomly sampled:

$$\begin{aligned} v &\sim Unif(30, 100) \\ p &= 1/v \\ l, h &\sim Unif(0.5, 4.5) \\ s.t.\ &\frac{h-l}{h+l} > 0.3 \\ s_0 &\sim Unif(l, h) \\ c_\sigma &\sim Unif(0.05, 0.3) \end{aligned}$$

At each time step, the state remains unchanged with probability $1-p$ and changes otherwise. When changing, the next state is sampled from a double exponential distribution centered around the previous state:

$$\begin{aligned} \lambda &= solve_{x\in\mathbb{R}^+}(1 - e^{x(h-s_{t-1})} - e^{x(s_{t-1}-l)} = 0) \\ s_t &= DoubleExp(s_{t-1}, \lambda) \end{aligned}$$

The point for this specific transition kernel is that small changes in network capacity should be more likely than drastic changes.

Finally, the network capacity c_t in each step is sampled from a Gaussian distribution, defined by these parameters:

$$c_t \sim Normal(s_t, s_t \cdot c_\sigma)$$

C.1.2 Training setup

Similar to the real-world ABR experiment, we use MLPs as the NN structures for CausalSim models and the SLSim model. We tune all the hyperparameters of both baselines as is done in the real-world ABR experiment (see §B.5 and §B.6). Table 5 comprehensively lists all hyperparameters used in training.

C.2 Can CausalSim Faithfully Simulate New Policies?

Similar to our real-data evaluations, we train models based on training data generated using all policies except a left-out policy, for which the model does not observe any data. Although

Model	Hyperparameter	Value
CausalSim (4 networks)	Hidden layers (SLSim)	(128, 128)
	Hidden layers (CausalSim: Extractor, Discriminator and $\mathcal{F}_{system}$)	(128, 128)
	Hidden layers (CausalSim: Action encoder)	(64, 64)
	Rank r	2
	Hidden layer Activation function	ReLU
	Output layer Activation function	Identity mapping
SLSim (1 network)	Optimizer	Adam [40]
	Learning rate	0.0001
	β_1	0.9
	β_2	0.999
	ε	10^{-8}
	Batch size	2^{13}
CausalSim	κ	{0.01, 0.1, 1, 10, 100}
	Training iterations (num_train_it)	20000
	num_disc_it	10
	Loss function	{MSE}
SLSim	Training iterations	20000
	Loss function	{Huber($\delta = 1.0$), L1, MSE}

Table 5: Training setup and hyperparameters for the synthetic ABR experiments.

traces come from the same generative process, no two trajectories in the dataset collected with different policies share the exact same trace, as this would be an unrealistic data collection scenario. Given that we have 9 possible policies to leave out, we have 9 possible datasets and models. There are 8 possible groups of trajectories to choose as sources, based on the policy that generated them. In total this leaves 72 different combinations and scenarios. We use the same hyper-parameter tuning approach examined in §B.5. Figure 13a compares the CDF of MSE values resulting from CausalSim and the two baselines. As evident, both baselines suffer from inaccurate predictions and in some cases are catastrophically inaccurate. On the contrary, CausalSim maintains favorable performance, even in the tail of its MSE distribution. Figure 13b gives a closer look at the CDF curves. We see CausalSim dominates at every scale.

Figure 13c is a heatmap of the two dimensional histogram of CausalSim predictions and ground truths. A fully accurate prediction scheme would perfectly match the ground truth and only the diagonal of this histogram would be populated. CausalSim almost achieves that, indicating it produces accurate trajectories on a step-by-step basis.

Further, in Figure 14, we compare the the Mean Absolute Percentage Error (MAPE) of CausalSim, ExpertSim and SLSim predictions across all trajectories at each time step for the first 35 steps. Note that the error naturally accumulates for all three methods as we move froward in time. However, CausalSim maintains a MAPE of ($\sim$5.1%) which significantly lower than both ExpertSim's and SLSim's ($\sim$10%).

C.3 Learning ABR policies with CausalSim

We observed how CausalSim can be used to design an improved policy in §6.2, and verified this through deployment in the wild. We would like to take these experiments one step further and ask *can CausalSim be used to design learning-based policies, such as with Reinforcement Learning (RL)?*

Recent work has shown that RL algorithms can learn strong ABR policies by learning through interactions with the environment [50]. Could we use a CausalSim model to train high-performance ABR policies without direct environment interaction? As a first step, we decided to carry out an initial experiment in the synthetic ABR environment. We build a CausalSim model using traces from a "simulated RCT" on the synthetic environment.

Performance Metric. ABR algorithms are typically evaluated through QoE metrics [75]. Assuming the chosen bitrate at step t was q_t, the download time was d_t and the buffer was b_t, we use the following QoE definition:

$$QoE_t = q_t - |q_t - q_{t-1}| - \mu \cdot max(0, d_t - b_{t-1})$$

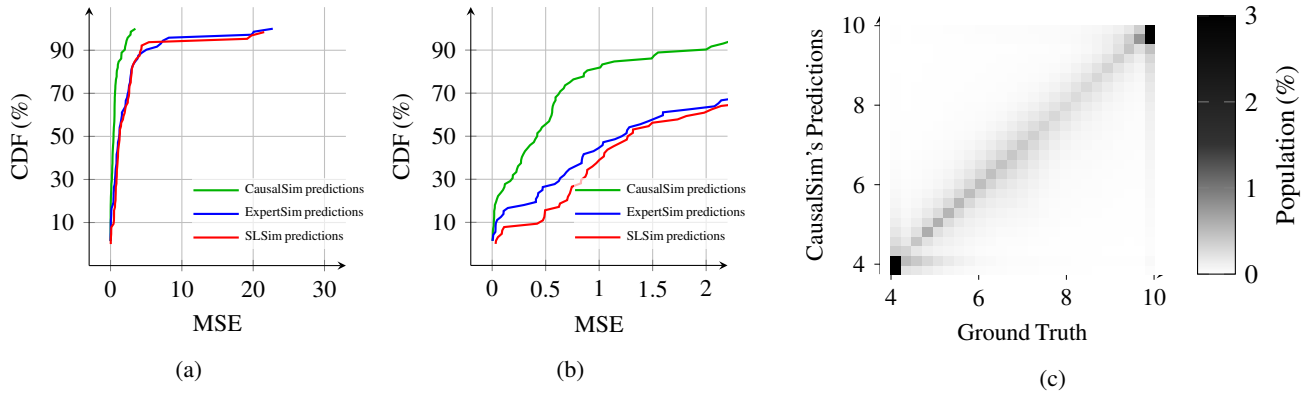


Figure 13: **(a)** Distribution of CausalSim, ExpertSim, and SLSim MSEs over all possible source left-out pairs. **(b)** The same figure with a smaller MSE range. In this magnified view, CausalSim clearly outperforms the baselines. **(c)** Two-dimensional histogram heatmap of CausalSim predictions vs. ground truth.

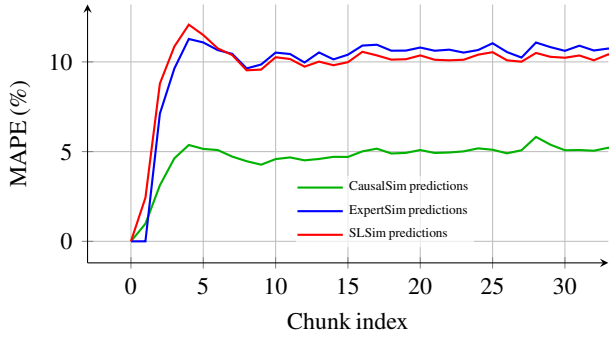


Figure 14: A time series plot of the Mean Absolute Percentage Error (MAPE) across all trajectories, for CausalSim, ExpertSim and SLSim predictions. Notice how errors accumulate in trajectory simulation.

This QoE metric captures three goals (in succession): 1) Stream in high quality, 2) Maintain a stable quality, 3) Avoid rebuffering. Better policies yield higher QoE values, where an ideal QoE is equal to the max bitrate.

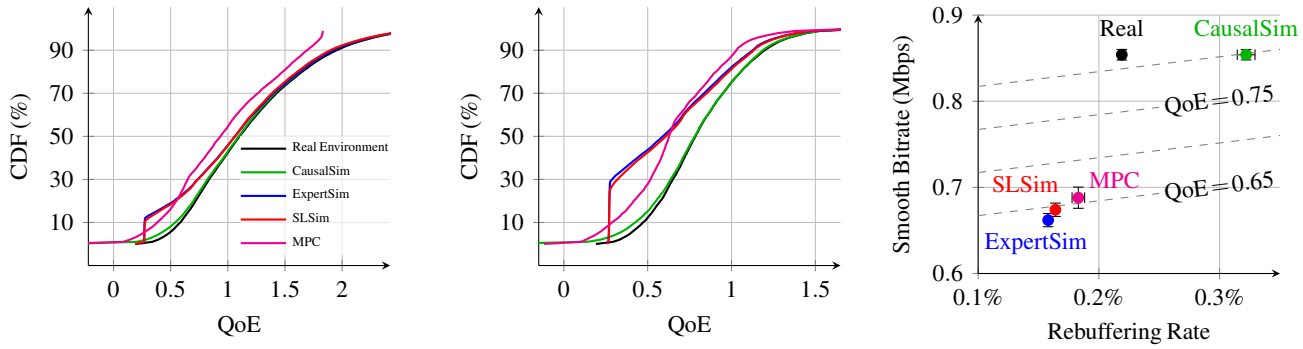
C.3.1 How to train policies via simulators?

To train the RL agent, we take a set of logged trajectories where the source policy was MPC and feed them to CausalSim. In each step, CausalSim will predict the next counterfactual observation and reward, and the RL agent will choose the next counterfactual action based on that observation. This process repeats until this simulated session is over, after which the counterfactual trajectory is used to train the RL agent. For the RL algorithm, we utilize the Advantage Actor Critic (A2C) method, a prominent on-policy algorithm, along with Generalized Advantage Estimation (GAE). Table 6 lists all hyperparameters for the RL training.

C.3.2 Does CausalSim train better policies?

Figure 15a plots the CDF of average session QoE that each policy attains. Here, *Real Environment* refers to training directly with the synthetic ABR environment, and CausalSim, ExpertSim and SLSim refer to policies trained by using each of these simulators. CausalSim trains policies nearly as well as training directly on the environment, while ExpertSim and SLSim fail to provide robust policies across all sessions. Figure 15b plots the CDFs for the high RTT (above 300 ms) clients, where the gap between CausalSim and the baseline simulators is even larger.

In this environment, chunk are downloaded according to the slow start model, where congestion control must ramp up its window size over several RTTs before the download rate can reach the available bandwidth. As a result, downloads of smaller chunks (with lower bitrates) incur a noticeable overhead, particularly on high-RTT paths. This overhead becomes less apparent as chosen bitrates become larger. Biased simulators such as SLSim and ExpertSim, which assume all actions lead to the same observed bandwidth, overestimate the achieved rate when counterfactual bitrates are smaller than factual ones (chosen by the source policy) and underestimate it when the counterfactual bitrates are larger. Since the source policy is conservative and tends to choose low bitrates, ExpertSim and SLSim find larger bitrates to be undesirable in the QoE trade-off. This can be seen in Figure 15c, which visualizes the 3 aspects of QoE in terms of the rebuffering rate and the smoothed birate, i.e the chosen bitrates with the smoothnes penalty. Notice how policies trained on the real environment and CausalSim utilize the network by 200 kbps more than other policies. The extra rebuffering that CausalSim incurs is negligible compared to the extra bitrate: 5.9 seconds every hour.



(a) Full population

(b) High RTT clients

(c) QoE breakdown in High RTT clients

Figure 15: CausalSim trained policies perform well, only marginally behind training on the real environment. Distribution of Quality of Experience (QoE) in policies trained with the real environment, CausalSim, ExpertSim, and the MPC policy. CausalSim does not underestimate bandwidth in high RTT clients and trains policies that strike the best balance in QoE goals.

Group	Hyperparameter	Value
Neural Network	Hidden layers	(32, 32)
	Hidden layer activation function	ReLU
	Output layer activation function	A2C actor: Softmax
		A2C critic: Identity mapping
	Optimizer	Adam [40]
	Learning rate	0.001
	β_1	0.9
	β_2	0.999
	ϵ	10^{-8}
	Weight decay	10^{-4}
A2C training	Episode lengths	490
	Epochs to convergence (T_c)	8000 (3920000 samples)
	Random seeds	4
	γ	0.96
	Entropy schedule	0.1 to 0 in 5000 epochs
	λ (for GAE)	0.95
Environment	Chunk length c	4
	Number of actions (bitrates)	6

Table 6: Training setup and hyperparameters for learning RL policies in the synthetic ABR environment.

C.4 Low-rank structure

As discussed in §4.1, we can formulate the counterfactual estimation problem in the context of matrix completion. For each time step, we know the chosen bitrate (action) and the achieved throughput (trace). We also know the trace is computed using a latent factor and the action. Suppose the latent factor is the network bottleneck capacity c_t[18]. $\mathcal{F}_{\text{trace}}$ describes how the achieved throughput (the trace) relates to this latent factor. Intuitively, this should be a close-to-linear function, $m_t \approx c_t$. But it's not exactly linear; for example, congestion control may under-utilize the network capacity for

[18]There may be other latent factors but bottleneck capacity is likely to have the strongest influence on the achieved throughput.

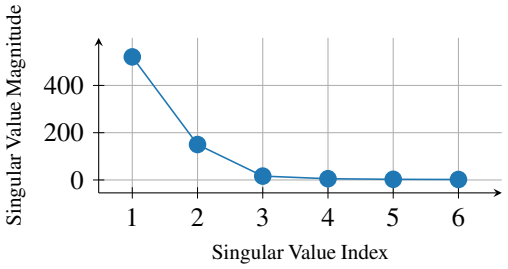


Figure 16: Singular values of matrix M in synthetic ABR suggest that M is approximately rank 2.

small transfers on high-RTT paths.

We form a matrix M, where the rows denote actions $a_t \in [A]$ and the columns denote the latent factors u_t^i for each trajectory. The 'factual' data we have are single observed trace values in each column, i.e for each step and each latent, we have observed the trace from a single action. To estimate counterfactuals, we must complete the matrix. We have no way of knowing the true $\mathcal{F}_{\text{trace}}$ in the Puffer dataset. But to get a sense for what it might look like and whether it's plausible that M is low rank, we can investigate this in the synthetic ABR environment instead.

For the TCP slow start model this environment uses, $\mathcal{F}_{\text{trace}}$ takes the following form:

$$\text{Let } \hat{RTT} := \frac{RTT}{\ln(2)} \tag{22}$$

$$m_t = \begin{cases} \dfrac{c_t}{1+\frac{\hat{RTT}\cdot(\ln(c_t/\dot{c})-c_t+\dot{c})}{s_t}} & \text{if } s_t \geq \hat{RTT}.(c_t-\dot{c}) \\ \dfrac{s_t}{\hat{RTT}\cdot\ln(\frac{s_t}{\hat{RTT}\cdot\dot{c}}+1)} & \text{otherwise} \end{cases} \tag{23}$$

where s_t is the chunk size (which itself is determined by the bitrate chosen by ABR) and $\dot{c}$ is the starting download rate in the slow start algorithm (in our case, equal to 2 MTUs). We use this model to generate a version of M with $A = 6$ actions and $U = 49000$ latent network conditions. We compute the singular value decomposition with the 6 singular values represented in non-increasing order ($\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_6$). The total "energy" of matrix is given by sum of squares of these singular values. It turns out that $\frac{\sigma_1^2+\sigma_2^2}{\text{total energy}}$ is more than 0.999. This suggests that most of the matrix is captured by its rank-2 approximation, as depicted in Figure 16. In other words, M is approximately low (=2) rank.

Appendix D Load Balancing

D.1 Does CausalSim Faithfully Infer Latent States?

We test the claim that estimating the exogenous latent state and using it to predict the next state was indeed the key to producing accurate counterfactual predictions, as the architecture of CausalSim suggests. To do so, we compare CausalSim's estimated latent state with the underlying job sizes—the job size is indeed the latent state that dictates the dynamics in the load balancing environment. We find that the estimated latent states and the job sizes are highly correlated, as illustrated in Figure 17, with a PCC of 0.994. This demonstrates that CausalSim can learn faithful representations of true latent states.

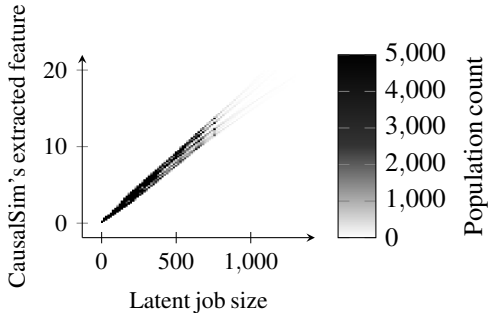


Figure 17: Two-dimensional histogram heatmap of CausalSim extracted latent state vs. latent job sizes.

D.2 Data & Algorithms

To simulate the load balancing problem described in §6.4.1, we need to set the server processing rates $\{r_i\}_{i=1}^N$, and arriving job sizes S_k. Server rates are generated randomly, as follows:

$$r_i = e^{u_i} \tag{24}$$

$$\text{where } u_i \sim Unif(-\ln(5), \ln(5)) \tag{25}$$

We generate job sizes using a time-varying Gaussian distribution. At step k of the trajectory, job size S_k is sampled as follows:

$$S_k \sim Normal(\mu_k, \sigma_k)$$

where μ_k and σ_k signify the mean and variance of the generative distribution at time step k. At each time step, with a probability of $p = 1/12000$, the mean and variance change and with a probability of $1-p$, they remain the same. The mean and variance values are drawn from random distributions, both at the start of a trajectory and when a change occurs, in the following manner:

If $k=0$ (start of trace) or, mean and variance must change:

$$\mu_k \sim Pareto(\alpha=1,\ L=10^1,\ H=10^{2.5}) \tag{26}$$

$$\sigma_k \sim Unif(0,\ 0.5\mu_k) \tag{27}$$

Else:

$$\mu_k = \mu_{k-1} \tag{28}$$

$$\sigma_k = \sigma_{k-1} \tag{29}$$

Jobs generated according to this process are temporally correlated, and therefore not independent and identically distributed. Training data consists of 5000 trajectories of length 1000, each of which was randomly assigned a policy from a set of 16 policies, described in Table 7.

Policies	Description	Used as source	Used as left out
Server limited policy (8 variations)	Randomly assign to only two servers	✓	×
Shortest queue	Assign to server with smallest queue	✓	✓
Power of k ($k \in \{2,3,4,5\}$)	Poll queue lengths of k server and assign to shortest queue	✓	✓
Oracle optimal	Normalize queue sizes with server rates and assign to shortest normalized queue	✓	✓
Tracker optimal	Similar to oracle, but estimates server rates with historical observations of processing times	✓	✓

Table 7: Scheduling policies used in the load balancing experiment.

D.3 Training setup

As before, we use MLPs as the NN structures for CausalSim models and the SLSim model and Table 8 is a comprehensive list of all hyperparameters used in training. We tune the parameter κ for CausalSim and the loss function in SLSim in a similar fashion to what is described in §B.5 and §B.6. Note that, as mentioned in §6.4.1, we assume access to $\mathcal{F}_{\text{system}}$ and focus on the more challenging task of estimating the trace quantities, for both CausalSim and SLSim. Therefore, in training, there are no observations and hence $\mathcal{L}_{\text{total}}$ consist of two terms: the squared loss of the trace quantities and the discriminator loss.

Appendix E Causal Inference Related Work

Identifying causal relationships from observational data is a critical problem in many domains [30], including medicine [55], epidemiology [59], economics [36], and education [23]. Indeed, identifying causal structure and answering causal inference queries is an emerging theme in different machine learning tasks recently, including computer vision [74, 76], reinforcement learning [6, 24], fairness [27], and time-series analysis [7] to name a few. One important aspect about causal inference is its ability to answer counterfactual queries. For such queries, many methods were developed; where some approaches are motivated by Pearl's structural causal model [57], and by Rubin's potential outcome framework [61]. We refer the interested reader to recent surveys such as [30] and references there in for an overview of recent advances in our ability to infer causal relationships from observational data.

Another related line of work within this literature is synthetic controls and its extension synthetic interventions, which aims to build synthetic trajectories of different units (e.g. individuals, geographic locations) under unseen interventions by appropriately learning across observed trajectories [4, 5, 9–12]. However, these approaches assume a static set of intervention and do not apply to our setting.

Model	Hyperparameter	Value
CausalSim (3 networks)	Hidden layers (SLSim)	(128, 128)
	Hidden layers (CausalSim: Extractor, Discriminator)	(128, 128)
	Hidden layers (CausalSim: Action encoder)	No hidden layers
	Rank r	1
	Hidden layer Activation function	ReLU
	Output layer Activation function	Identity mapping
	Optimizer	Adam [40]
SLSim (1 network)	Learning rate	0.0001
	β_1	0.9
	β_2	0.999
	ε	10^{-8}
	Batch size	2^{13}
CausalSim	κ	$\{0.01, 0.1, 1, 10, 100\}$
	Training iterations (num_train_it)	10000
	num_disc_it	10
SLSim	Training iterations	10000
	Loss function	Huber, L1, MSE

Table 8: Training setup and hyperparameters for the load balancing experiment.

HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network

Zhenyu Song*†, Kevin Chen*, Nikhil Sarda*, Deniz Altınbüken, Eugene Brevdo,
Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, Ramki Gummadi*

Google

Abstract

Video streaming services are among the largest web applications in production, and a large source of downstream internet traffic. A large-scale video streaming service at Google, YouTube, leverages a Content Delivery Network (CDN) to serve its users. A key consideration in providing a seamless service is cache efficiency. In this work, we demonstrate machine learning techniques to improve the efficiency of YouTube's CDN DRAM cache. While many recently proposed learning-based caching algorithms show promising results, we identify and address three challenges blocking deployment of such techniques in a large-scale production environment: computation overhead for learning, robust byte miss ratio improvement, and measuring impact under production noise. We propose a novel caching algorithm, HALP, which achieves low CPU overhead and robust byte miss ratio improvement by augmenting a heuristic policy with machine learning. We also propose a production measurement method, impact distribution analysis, that can accurately measure the impact distribution of a new caching algorithm deployment in a noisy production environment.

HALP has been running in YouTube CDN production as a DRAM level eviction algorithm since early 2022 and has reliably reduced the byte miss during peak by an average of 9.1% while expending a modest CPU overhead of 1.8%.

1 Introduction

YouTube is one of the largest sources of downstream internet traffic, accounting for 15% of global application traffic in 2021 [27]. It leverages a Content Delivery Network with a presence in more than 200 countries and territories to serve videos to over 2 billion users [30]. Caching in CDNs is done by storing content, such as videos, in proxy servers that are distributed closer to end users instead of delivering content from the origin servers. A CDN uses multiple levels of caches to reduce the cost of content distribution and access latency for end users. A key metric to optimize in CDN caches is byte miss ratio, i.e., the portion of user-requested bytes missed in the CDN cache.

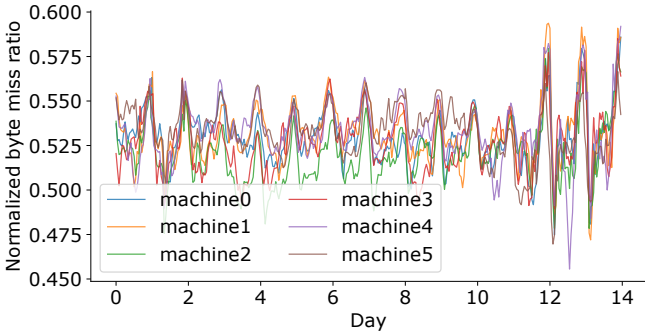


Figure 1: Byte miss ratios over time for six machines on a rack, normalized by dividing with a reference constant to hide the proprietary absolute values. The substantial differences across machines and over time make it hard to measure a new cache algorithm's production impact accurately.

Recently, machine learning techniques have been used to improve cache eviction and admission policies (e.g., [32, 34]). Caching algorithms can benefit from learning patterns from existing workloads, predicting which byte is more likely to be accessed in the future, and using this information to improve caching decisions.

In this paper, we present a new cache eviction algorithm called Heuristic-Augmented Learned Preferences (HALP), and share our experience in deploying HALP at a large scale. From our experience, while reducing the byte miss ratio is important to improve cache efficiency, it is not the sole criterion for deployment. For a solution that uses machine learning to be deployed in a large-scale production environment, there are three main challenges that need to be tackled:

- **Computation overhead for learning.** Learning-based cache algorithms can be more computationally expensive to run compared to heuristic-based algorithms. The model training and prediction cost is high compared to

*Equal technical contributions. The corresponding author is Pawel Jurczyk (pawelj@google.com).

†Zhenyu is affiliated with Princeton University, but this work was done during his internship in Google.

normal cache operation such as LRU eviction. Using LRB [32] as an example, for each eviction it needs to run predictions for 64 objects, which makes deployment cost-prohibitive ($\approx$ 19.2% additional CPU overhead)

- **Robust byte miss ratio improvement.** Learning-based cache algorithms can introduce regressions if their design does not include a regression prevention mechanism. For large-scale systems, bounding regression of byte miss ratio is crucial. YouTube CDN contains a large number of locations, and byte miss ratio regressions in even a few locations could result in degraded user experience. In addition, having a robust algorithm also increases our confidence in the design.
- **Measuring impact under production noise.** It is challenging to accurately measure the impact of a new eviction algorithm in a large-scale deployment. We cannot solely rely on simulations as they are imperfect proxies for production behavior. It is also impractical to replicate user requests and test different algorithms at every location. Therefore, current production practice uses A/B testing. An example is to compare different machines on a rack because machines on a rack share the same hardware/software configurations, and the request mix they receive should be similar. However, in practice, the behaviors of machines are never identical. Figure 1 shows byte miss ratios over time for six machines on a rack. The substantial differences across machines and over time make it hard to measure the production impact of a new algorithm accurately.

To tackle the first two challenges, we develop a novel approach, the HALP policy, to perform eviction decisions with low-overhead and to generalize over the whole production system with limited regressions. It achieves this by augmenting a heuristic policy with machine learning instead of learning a policy end-to-end. The HALP eviction policy uses the heuristic policy to select eviction candidates and the learning policy to pick the final object to evict from those candidates.

To address the third challenge, we developed an impact distribution analysis that evaluates the impact of a new caching algorithm deployment in a noisy production environment.

HALP has been deployed in YouTube's CDN as a DRAM level eviction algorithm since early 2022. It has robustly reduced the byte miss by an average of 9.1%. In addition, these improvements were achieved with a modest 1.8% CPU overhead.

In this paper we make the following three contributions:

- We present Heuristic Augmented Learned Preferences (HALP), a learned cache eviction algorithm with low computation overhead and robust byte miss ratio improvement by augmenting a heuristic policy with a learnable scoring function.
- We propose an impact distribution analysis to measure the impact of a caching algorithm in the presence of production noise.
- We evaluate HALP in YouTube's large-scale production environment and provide a detailed analysis on how it improves the cache efficiency of YouTube CDNs. [1]

The paper is structured as follows: §2 describes the background of the problem. §3 covers the design of HALP. §4 introduces our impact distribution analysis design, and §5 shows the evaluation results.

2 Background

In this section, we give an overview of the YouTube CDN architecture. We then describe heuristic-based caching algorithms and learning-based caching algorithms. Lastly, we describe the key ideas for deploying a learned cache algorithm in a large-scale production environment.

2.1 YouTube CDN Edge Cluster Architecture

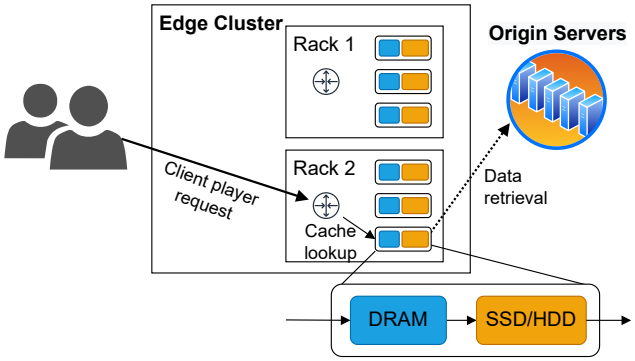


Figure 2: A YouTube CDN edge cluster contains multiple racks of servers. Machines in a rack are of the same type.

YouTube CDN [4, 11] contains edge clusters spreading more than 200 countries and territories globally. As shown in Figure 2, an edge cluster consists of multiple racks. Each rack contains multiple cache servers configured homogeneously using the same type of machines. Servers from different racks may have hardware from different generations. Each cache server is equipped with DRAM, SSDs and HDDs used for caching data chunks. A video is stored in these data chunks on the cache server.

Client player requests are sharded amongst machines in an edge cluster. A request includes a key and a byte range of a data chunk. Because a video is played sequentially, video range requests are issued sequentially as well. On the arrival of a request, the server checks if the requested data chunk is

[1] Two traces from a developed market region and an emerging market region (§5.2) can be shared with interested parties, but a signed data sharing agreement between Google and the outside institutions is required.

in its DRAM. If it is not present (a.k.a. is missed), the data will be fetched from other cache layers such as local SSDs and HDDs, with the remote origin server being the last resort. When the DRAM cache is full and a miss occurs, an eviction algorithm is used to remove data chunks from the cache to insert new data chunks.

As the first caching tier, the DRAM cache serves an important role in reducing traffic for subsequent tiers. It also contributes to the overall storage costs of the YouTube CDN. A better DRAM eviction algorithm with a lower byte miss ratio would require less DRAM to be provisioned while keeping a similar traffic reduction on subsequent tiers. This saves the overall resource cost as long as the computation overhead is modest (which requires additional CPU resources). We focus on the byte miss ratio during the peak hours. This is because during peak hours, large numbers of videos are concurrently accessed, causing the byte miss ratio peaks, which could degrade Quality of Experience (QoE). We therefore focus on reducing the 95th percentile byte miss ratio: **P95 byte miss ratio**. We choose to not directly optimize QoE because it is too noisy as feedback for each cache eviction. §5.3 and §5.4 list other metrics related to cache performance.

The previous eviction algorithm used in production [28] uses heuristic to rank chunks. A score is computed for each chunk by summing its request rate score and end of chunk score, and the chunk with lowest score would be evicted. The request rate score is calculated based on the chunk's past request rate, which captures the temporal locality. The end of chunk score is a binary score indicating whether the previous range request hits the chunk end. Since range requests for a chunk are issued sequentially, after a client requests the last byte of a video chunk, the same chunk is less likely to be fetched again. This score captures the spatial locality.

2.2 Heuristic and Learned Cache Algorithms

Many heuristic cache algorithms maintain a priority queue for objects in the cache and select the lowest priority object to evict when a miss occurs. For example, A Least Recently Used (LRU) policy uses the latest time of access for an object to determine evictions. This ordering is good for workloads where objects that have been accessed recently are more likely to be accessed repeatedly. A First In First Out (FIFO) policy uses the order in which items were inserted into the cache for determining evictions. This ordering performs well for workloads where objects are accessed sequentially. Managing priority queues is efficient, which makes these algorithms efficient as well. However, these algorithms work well in some workloads, but not in others. Caching policies that can self-tune and balance between different features, recency and frequency in Adaptive Replacement Cache (ARC) [25] or object size and frequency in Greedy-Dual-Size-Frequency (GDSF) [12, 13], can cover a wider range of workloads but only adapt to specific features [21], limiting their performance for changing workloads.

Learning-based algorithms like LRB [32] achieve better performance than heuristic algorithms, because they train a model to learn the cache access pattern directly from the trace instead of assuming a static workload behavior. As an example, LRB maintains features for objects that are both currently, and historically present in the cache, and trains a regression model to predict an object's time to next access. When an eviction is required, it randomly samples 64 objects and runs this predictive model on them and evicts the object which is predicted to be accessed furthest in the future.

When optimizing the byte miss ratio for variable-size objects, the eviction methodology is similar to optimizing the miss ratio for uni-size objects. This is because in the variable-size object scenario, we can treat each eviction decision as a group of decisions, which evicts each byte of an object individually.

3 HALP Eviction Policy Design

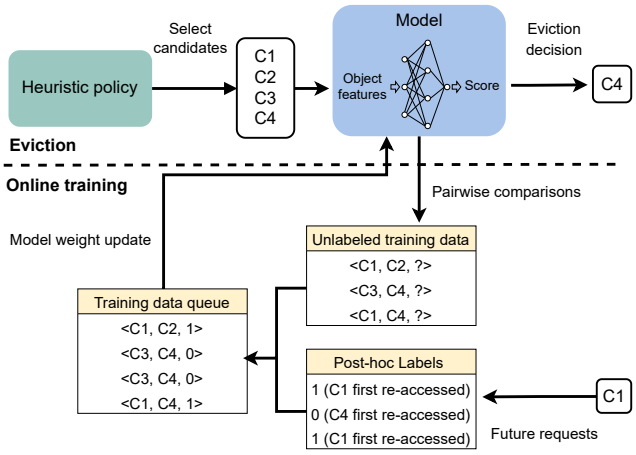


Figure 3: The architecture of HALP. A key component of HALP is a neural network based *score function*, whose inputs correspond to the features for a single eviction candidate, and whose output is a real valued score which tracks the likelihood of a quick re-access to the same object. When an eviction is required, a heuristic policy (e.g., LRU) is used to propose a small set of eviction candidates. Then a neural network-based model ranks the eviction candidates and selects the final eviction decision by pairwise comparisons. The same pairwise comparisons are also used to generate training data for online training.

This section describes the design of HALP, which is illustrated in Figure 3. A key component of HALP is a neural network based *score function*, whose inputs correspond to the features for a single eviction candidate, and whose output is a real valued score which tracks the likelihood of a quick re-access to the same object. When an eviction is required, a

heuristic policy is first used to propose a small set of eviction candidates. Then, a neural network score function is used to re-rank this small set of candidates, to identify the final eviction decision. A key design challenge involves how to learn the scoring function, which involves both generating the training data and adjusting the weights of the neural network. As part of its design, HALP also includes the steps required to efficiently update this score function, starting with randomly initialized weights.

Because of these design choices, HALP can be deployed without the operational overhead of having to separately manage the labeled examples, training procedure and the model versions in separate offline pipelines. Therefore, HALP has minimal extra overhead for operation similar to other heuristic policies, but has the added benefit of being able to take advantage of additional features to make its eviction decisions and continuously adapt to a changing access patterns.

To learn the score function efficiently, we convert the ranking problem into a small set of pairwise preference queries, which is a general and robust framework for learning to rank [9, 29] multiple items from a list. As a result, HALP makes repeated use of pairwise comparisons during decision making to simultaneously generate training data for online training. One challenge in efficiently managing the training data is that the time required to identify the labels is non-deterministic and depends on the future re-accesses to items. HALP snapshots the features generated for pairwise comparisons used at eviction decision time saved as unlabeled training data tuples (see Figure 3). In parallel, HALP continuously observes incoming requests to resolve any pending labels for prior comparisons and generate training data that continuously updates the model.

3.1 Heuristic-based Candidate Selection

A key insight for ensuring a low overhead is that many objects can be easily excluded from eviction consideration without the need to use expensive computations, ML or otherwise. For example, objects near the head of an LRU priority queue are less likely to be good eviction candidates as opposed to objects near the tail. Therefore, we can appropriately bias our learned eviction towards only the tail instead of considering the entire cache, saving overhead on training and inference.

The goal of using a heuristic policy for candidate selection is to reduce the ML computational overhead. It also provides a lower limit on decision quality. This heuristic algorithm can be selected as LRU, LFU, or other heuristic policies. We find in practice LRU policy is sufficient to achieve good performance.

The number of eviction candidates is a hyperparameter. If too many candidates are selected, the ML pipeline overhead will be too high. But too few candidates may lead to not a single good candidate to evict. We find empirically selecting four candidates achieves a good balance between the recall of good candidates and the incurred CPU overhead.

3.2 Ranking-based Learned Eviction

HALP is designed to provide better eviction decisions than the heuristic algorithm in the general case. To achieve this, the pairwise comparisons should pick the eviction decision that is the best for improving cache efficiency. Since the goal of cache eviction is to use the limited size of the cache to receive as many hits as possible, finding the best eviction decision is equivalent to ranking the candidate that will be accessed furthest in the future (or not at all) highest, in effect evicting it before other candidates.

After four eviction candidates are selected from the heuristic, the best candidate is selected based on three pairwise comparisons done in tournament style. The deselected candidates are re-inserted into the heuristic policy. In the case for LRU, those deselected candidates are re-inserted into the head of LRU queue.

To have a theoretical intuition that the combination of a heuristic and a learning policy can increase the robustness of eviction candidate selection, we analyze a simple Gaussian model for the benefits of re-ranking in Appendix B. This analysis underlines the conditions under which such re-ranking might generate more utility than the baseline heuristic.

Online Training. As shown in Figure 3, when a pairwise comparison is done, the same pair of candidates is selected to generate training data. However, at the time of prediction, the required label (i.e., which of these two candidates will be accessed further in the future) is not available. Therefore, an initial feature snapshot is taken at the pairwise comparison during eviction and is buffered in an unlabeled state with a label placeholder until one of the candidates is accessed again, making the label available. Accordingly, HALP maintains a collection of pending comparisons. This collection of pending comparisons continuously observes all incoming requests, and upon the first access to either candidate, a binary label is assigned to construct the training example.

HALP keeps the feature metadata of objects in a "*ghost cache*". For our application of video caching, this metadata is lightweight relative to the sizes of the objects being cached, therefore the information continues to be stored for keys that are evicted from the cache up to a limit. This limit is set to be a multiple of the number of elements in the actual cache to track enough history. Evictions from the ghost cache are performed using LRU when the size exceeds this set limit. When a key is removed from the ghost cache, pending comparisons associated with the key are also deleted.

The training data generated from the above procedure is stored in an in-memory replay buffer. When the replay buffer accumulates 1024 training entries, it creates a mini-batch to update the ML model. The retrain batch size is a hyperparameter of HALP and was chosen empirically. Theoretically, a

pathological workload could have access pattern shifts aligning with the retraining period, causing HALP performance degradation. However, we didn't observe this in our production workloads.

The online training framework and the model itself are written using XLA [2] and carefully crafted C++ code to ensure low overhead. We also leverage uncommon synchronization primitives such as user space per CPU spinlocks and RCUs to ensure maximum performance without sacrificing scalability and thread safety in highly concurrent environments.

ML Model. The model is trained as a binary classification task (which of a pair gets re-accessed first) with cross entropy loss. It is a simple neural network model with one hidden layer. We found that increasing the number of layers did not help improve the model further. With this simple model, we are able to run a pairwise prediction in 720 ns, and each training in several ms. And HALP implementation is based on Google's SmartChoices service [10]. Details about the loss function and the model weight updates are provided in Appendix A.

Feature name	Dimension
Access-based	
Time between accesses	32
Exponential decay counters	10
Number of accesses	1
Average time between accesses	1
Time since last access	1
Video-specific	
End of chunk	1

Table 1: Features used by HALP.

Features. Table 1 shows the features HALP uses. Of these features, time since last access, time between accesses, and exponential decay counters are the same as the features used in [32]. Time since last access and time between accesses capture short-term access patterns while they retain information about at most 32 accesses. Exponential decay counters (EDCs) capture longer term trends. The end of chunk score is identical to the previous production algorithm (§2.1).

4 Impact Distribution Analysis

Comparing cache algorithms may seem like a straightforward hypothesis test (e.g., t-test or z-test) over an A/B testing experiment. A new algorithm with lower byte miss ratio that passes the hypothesis test would generally be considered as an improvement. However, the operating conditions in a large-scale system could be very diverse, and understanding the robustness of an improvement is critical to decision making in practice.

To illustrate the risk of solely relying on mean-shift estimates, consider a scenario where a new algorithm is beneficial for most machines but performs extremely poorly for some small set of machines. In that case, applying the new algorithm everywhere could be sub-optimal. Any algorithms without a theoretical performance lower-bound (relative to an optimal solution) have these risks, but the concern is exacerbated for learning algorithms that are prone to over-fitting.

A naive approach to the diversity problem is to enumerate all configurations and perform separate A/B tests (e.g., one test for each rack where workload and hardware is assumed to be similar). However, this severely limits the number of data points, and the signal to noise ratio for each configuration could be very poor in a production setting. Figure 1 is a typical example of byte miss ratio variation for production machines on the same rack with identical setting.

We propose a novel impact distribution analysis to get a more holistic picture of how a new algorithm is affecting the fleet. Specifically, instead of estimating the average performance change, we try to estimate the *distribution* of performance changes across different conditions.

4.1 Model of Measurements

We model the measured relative improvement as

$$M = I + N \tag{1}$$

where I represents actual impact and N represents the noise. In other words, we model the PDF of M as a convolution between I and N.

The core idea is that we could directly sample M by A/B tests, and sample N by A/A tests (performance difference measured in no-op experiment), and once we have those two distributions, we can get to I by deconvolution.

4.2 Measurement Setup

The environment we want to measure the effect of using HALP comprises of racks from hundreds of locations. In our experiment setup, we randomly split machines in a rack into three different configurations:

- **Experiment Machines:** Experiment machines use the HALP algorithm.
- **No-op Machines:** No-op machines use the baseline caching algorithm. They are used to measure the production environment noise.
- **Control Machines:** Control machines also use the baseline caching algorithm. They are selected as the baseline to compare with the experiment group and no-op group.

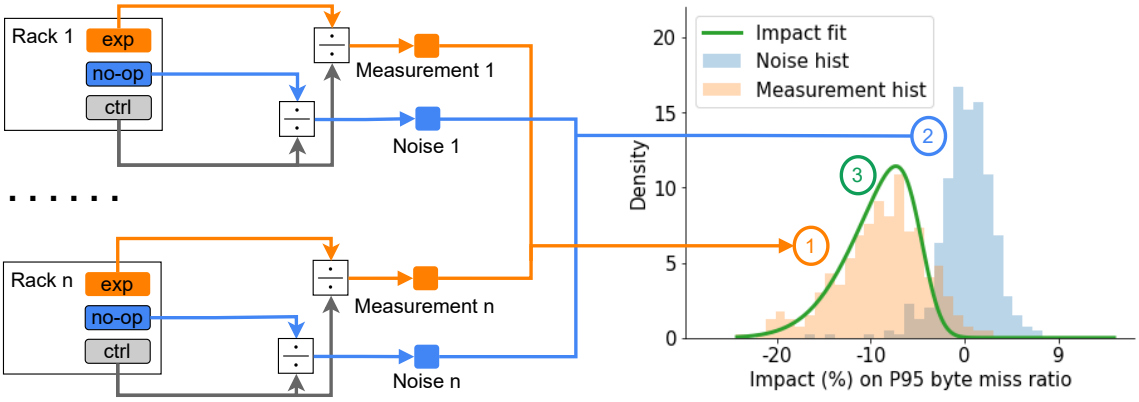


Figure 4: Impact distribution analysis procedure: 1. Estimate measurement distribution. 2. Estimate noise distribution. 3. Fit impact distribution.

Figure 4 shows how the measurements from these three groups are used to calculate the impact distribution. We first collect the relative values (e.g., relative byte miss ratios) of experiment machines over control machines as measurement samples (Measurement M). Then we collect the relative values of no-op machines over control machines as an estimation of the environment noise (Noise N). Finally, we fit an impact distribution from the measurement distribution and noise distribution using each comparison as a data point, as we describe in the next subsection.

4.3 Fitting Impact Distribution

Algorithm 1 Algorithm for fitting impact distribution

Input: measurement samples M, noise samples N, and distribution candidates.

Output: measurement distribution P_M, impact distribution P_I, noise distribution P_N.

1: $P_N = FitByMLE(dist=\text{“t-dist”}, N)$
2: **for** candidate_dist in candidate_distributions **do**
3: // Approximate P_M by discretized grid G.
4: $P_I = FitByMLE(dist=\text{candidate_dist}, M, P_N)$, with $P_M(m) \approx \sum_{v \in G} P_N(v) P_I(m-v)$
5: **end for**
6: **return** P_I with the highest likelihood

Algorithm 1 describes our algorithm to fit the impact distribution given sample M and N. We first fit the noise distribution P_N using noise samples with maximum likelihood estimation (MLE) (Line 1). We assume the noise has zero mean and follows a t-distribution. We choose a t-distribution because we expect noise to exhibit a symmetric and bell-shaped behavior like the normal distribution but allow fitting to have more degrees of freedom.

Next, we fit the impact distribution (Lines 2-5). This is done in two steps. First, a distribution type for impact needs to be chosen (Line 2). Since this depends on the setting, several well-known distributions could be reasonable candidates. Therefore, we iterate over a list of common distributions (beta, non-central student, and skewed normal) and pick the one that best fits our data. Second, we estimate the measurement distribution by discretizing the distribution into a fine-grained grid G. Then we use the maximum likelihood estimation to fit the chosen distribution candidate to find the measurement distribution that is the best fit (Line 4).

Note that this method is only feasible because we have machine data from more than 200 countries and territories. Without enough samples the fitting will not be able to recover impact accurately from the noise.

5 Evaluation

In this section, our goal is to answer the following questions for HALP and our impact distribution analysis:

- Can HALP improve cache performance without causing regression in production? (§5.3)?
- What is the computation overhead of HALP compared to the previous production algorithm (§5.4)?
- How does HALP compare to other learned and heuristic algorithms (§5.5)?
- What are the effects when changing HALP's hyperparameters (§5.6)?

5.1 Deployment Setup

Deployment HALP was rolled out in production in early 2022. The rollout was done in stages, and the impact of the new algorithm was monitored using the impact distribution

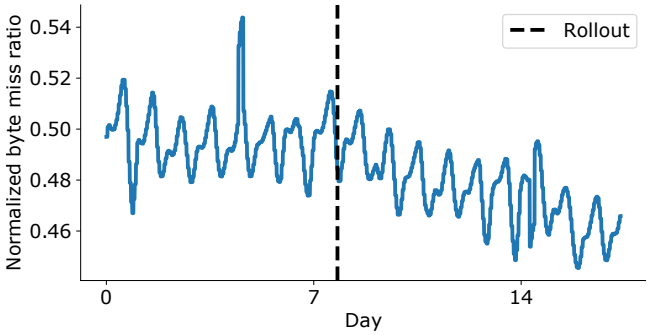


Figure 5: YouTube fleet normalized byte miss ratio for the DRAM level, before and after rollout.

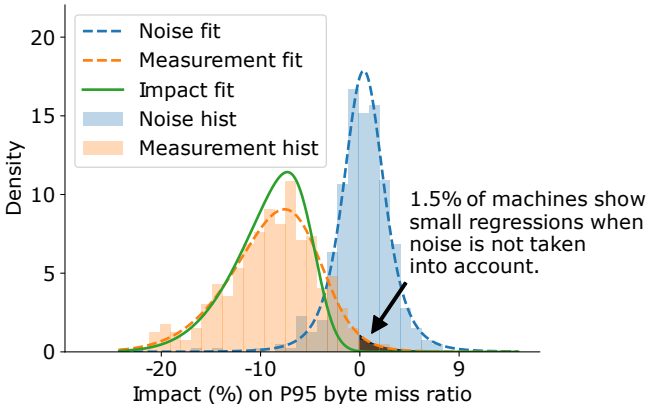


Figure 6: Our impact distribution analysis can "denoise" the experiment measurements from the no-op measurements and reveals HALP's clear average 9.1% byte miss ratio reduction with negligible regression. When noise is not taken into account, the measurements show 1.5% of racks that were impacted negatively, where some machines showed up to 4% byte miss ratio increase.

analysis on the changes in DRAM byte miss ratio. Figure 5 shows the DRAM byte miss ratio changes in the fleet before and after the rollout, which shows a fleet-wide drop.

Release Process HALP has an online design to make sure that the model does not degrade over time. As a result, the online model does not require new releases. While the model does not need to go through production releases, code changes and any improvements in HALP design go into production through releases that happen periodically. HALP is integrated into the existing YouTube release process, which guarantees that during rollout, HALP goes through the release tests and any code changes are released in a safe manner.

Monitoring As part of maintaining stable performance, HALP is integrated into the monitoring setup used for monitoring YouTube deployment. In addition to existing metrics that monitor cache efficiency, two new metrics were added to monitor HALP performance: 1) model accuracy, and 2) the byte miss ratio difference between HALP and a holdout previous production algorithm. For model accuracy, we monitor model loss which is an indicator of how good the model decisions are. For the byte miss ratio difference, we keep 1% of the machines running with previous production algorithm and alert if the byte miss ratio for machines using HALP become worse than the heuristic algorithm. We do not use the impact distribution analysis here as our goal is to detect abnormal behaviors with a low false positive ratio.

5.2 Experimental Methodology

Production experiments. To measure reduction in the byte miss ratio and overhead, we used production experiments and our impact distribution analysis. To use the impact distribution analysis, we randomly selected a small percentage of racks from all locations. For each rack, we selected one experiment machine, one no-op machine, and the rest as control machines (§4). We use one day of data for our measurement after observing HALP training is stable.

Simulation experiments. We use simulation to measure how HALP compares to other cache algorithms and how the changes in hyperparameters effect the performance of HALP. Because simulation does not have production noise and is deterministic, and direct comparisons can be set up reliably, we compare the byte miss ratio from simulations directly instead of using our impact distribution analysis.

For simulation experiments (except two experiments in §5.5), we use traces from a small percentage of randomly selected locations. For each location selected, we choose four traces, each three days long, with each trace coming from different quarters in the calendar year of 2021 (except the retrain interval experiment uses six days long trace). Using a diverse set of traces helps reduce seasonal/weekly noise. For each simulation, the first day of the trace is used as a warm-up, and we measure the P95 byte miss ratio of the next two days.

5.3 HALP Improvements in Production

P95 byte miss ratio. This experiment measures the impact of HALP on the byte miss ratio, disk latency, and joint latency during production. To measure the improvement, we collected byte miss ratios from machines on randomly selected racks and applied our impact distribution analysis. Figure 6 shows the relative change in the byte miss ratio distribution. The regions and lines are the P95 byte miss ratio distribution relative to the control groups.

The blue region is the no-op group relative change distribution. Although the no-op group uses the same configuration as the control group, the noise can cause up to 10% difference in P95 byte miss ratio. The blue dash line shows the fitted t-distribution of the noise. The orange region is the mea-

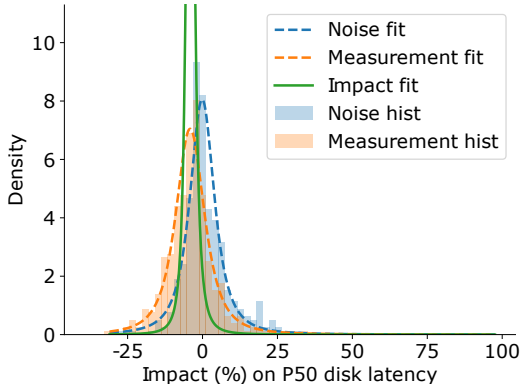


Figure 7: Compared with the previous production algorithm, HALP reduces disk first byte latency by an average of 3.8%.

surement distribution. The orange dash line shows the measurement fitted gamma distribution, which was picked as the best distribution for this specific measurement. When noise is not taken into account, as shown by the orange dash line there is 1.5% of racks that were impacted negatively, where some machines showed up to 4% byte miss ratio increase. However, just looking at the measurement fit, we do not know whether the negative impact is because of the algorithm or the production noise.

Our impact distribution analysis (Section 4) can "denoise" the experiment measurements from the no-op measurements, and the result is the green curve showing a clean byte miss ratio reduction up to 24% with negligible regression, with an average reduction of 9.1%. This shows HALP can not only improve the average byte miss ratio, but also has negligible regression. In addition, the variance of improvements also shows different locations have different access patterns which have different difficulty for learning.

Disk first byte latency. Disk first byte latency is the time between a disk cache receives a request and returns the first byte. It is a good indicator of DRAM cache efficiency because a better DRAM eviction algorithm reduces the number of requests to the disk layer, thus reducing the disk request queue length. Figure 7 shows the disk first byte latency impact of HALP. Compared with the previous production algorithm, the change in latency ranges from a 13% reduction to a 5% increase, with an average reduction of 3.8%.

The tail increase (the part of the impact fit that is above 0) is likely to come from the object miss ratio increase, which is more correlated with disk first byte latency than the byte miss ratio. Different from the byte miss ratio, the object miss ratio is the fraction of user requests missed in the cache. These two metrics may conflict with each other. Since HALP's primary goal is to reduce the byte miss ratio, it may slightly increase the object miss ratio in certain cases.

Join latency. Join latency is the time taken to start video playback after the user hits "play", and one of the most important metrics of streaming. Since join latency is a playback metric that involves both clients and servers instead of servers only, we are unable to use our impact distribution analysis to measure it. We set up an experiment that distributes playbacks from clients to server machines with and without HALP and compares latency on clients. HALP reduces join latency from 1.03% to 1.41%, with an average reduction of 1.22%. This shows that the improvement of the memory cache has a strong impact on the end-to-end user experience.

5.4 HALP Computation Overhead

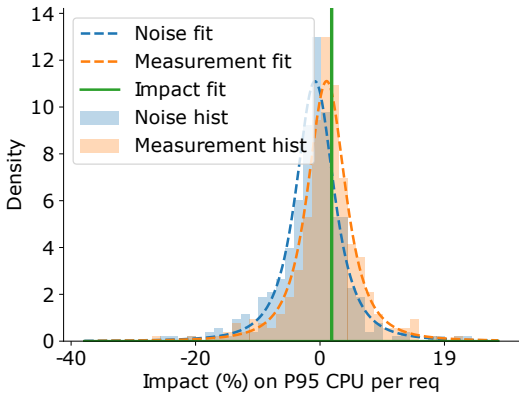


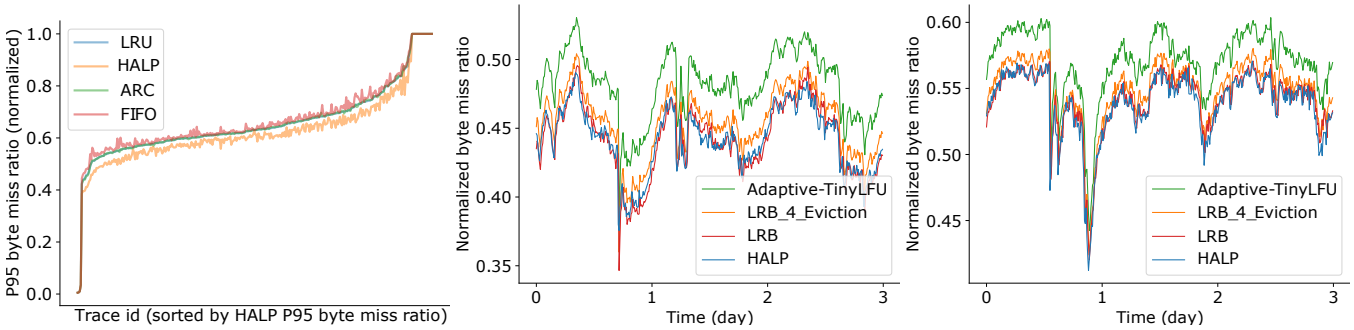
Figure 8: The CPU overhead of HALP is 1.8% per request with low variance. This implies the additional CPU cost is roughly linear to the number of requests.

Learning comes with overheads, so it is important to measure these overheads and underline the trade-offs. In this section, we show the computation overhead associated with using HALP for cache eviction decisions, including prediction and online training.

We have analyzed the extra CPU overhead that is incurred while using HALP using our impact distribution analysis, and Figure 8 shows the impact of P95 CPU normalized per request. The CPU impact is consistently at 1.8% with low variance, meaning the additional CPU cost is roughly linear to the number of requests. This is because both training and prediction costs are roughly linear to the number of requests. For training, the cost is roughly linear to the amount of training data, and on average each prediction generates a single pair of training data. In addition, each eviction requires three pairwise comparisons. Finally, since all locations have similar byte miss ratios, the number of evictions (misses) is roughly linear in the number of requests. To conclude, the computation overhead is small compared to the byte miss ratio improvement.

5.5 HALP vs. Other Cache Algorithms

To further evaluate HALP, we ran simulation experiments to compare it with other cache eviction algorithms.



(a) (b) Developed market (c) Emerging market

Figure 9: (Fig. 9a) P95 byte miss ratios for different cache algorithms over a variety of traces in simulation. HALP achieves a strictly better performance than all other algorithms on 92.6% of traces, and achieves the same performance as the best of the other algorithms on 7% of traces, in effect performing worse than the best algorithm on only 0.4% of traces. (Figs. 9b and 9c) Normalized byte miss ratio over time for HALP and LRB [32] on production traces from a developed market region, and an emerging market region. HALP achieves a similar byte miss ratio, but only needs 4 eviction candidates compared to 64 for LRB. Tuning LRB's eviction candidates from 64 to 4 would increase P95 byte miss by about 2%.

Comparison with classic cache algorithms

We compared HALP with three heuristic cache algorithms: LRU, FIFO, and ARC [25]. Figure 9a shows the normalized P95 byte miss ratios for different traces, sorted by HALP P95 byte miss ratio. HALP achieves a strictly better performance than all other algorithms on 92.6% of traces, and achieves the same performance as the best of the other algorithms on 7% of traces, in effect performing worse than the best algorithm on only 0.4% of traces.

Comparison with advanced cache algorithms

We compared HALP with a state-of-the-art learned cache algorithm LRB [32] and heuristic cache algorithm Adaptive-TinyLFU [15]. We use two YouTube production traces from a developed market region and an emerging market region in 2020 to get robust results. These traces are four days long. Our implementation of LRB and Adaptive-TinyLFU is based on LRB's open-source simulator [1]. For a fair comparison, we extended LRB's features to be identical as HALP. We tuned LRB's major hyperparameter (memory window) using its public implementation. In additional to using LRB's original 64 eviction candidates, we also tested using 4 eviction candidates identical as HALP. Figure 9b and 9c show normalized byte miss ratios over time for Adaptive-TinyLFU, LRB with 4/64 eviction candidates, and LRB. We use the first day of each trace as a warm-up, and exclude it from the figures.

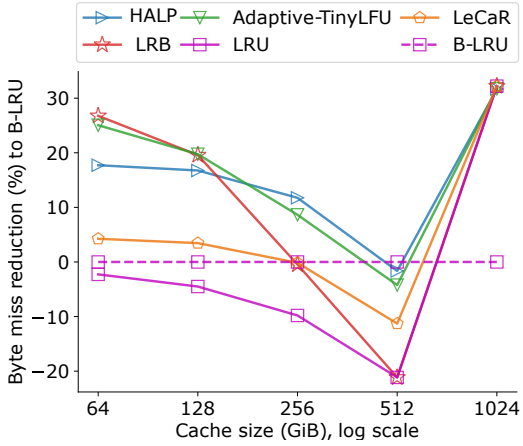


Figure 10: The byte miss reduction of different algorithms compared to B-LRU on a public trace from a Wikipedia CDN node. LRB, Adaptive-TinyLFU, HALP, and B-LRU achieve the best performance individually at cache size 64 GiB, 128 GiB, 256 GiB, 512 GiB. At 1024 GiB, the cache is big enough that all cache algorithms converge.

Adaptive-TinyLFU achieves P95 byte miss ratios of 0.515 and 0.598 on two traces. Compared to it, LRB and HALP achieve smaller ratios (0.479/0.565 for LRB, 0.475/0.564 for HALP). HALP achieves similar P95 byte miss ratios compared to LRB (0.17%/0.83% less byte misses) with less than an order of magnitude computation overhead. For each eviction it only compares 4 eviction candidates instead of 64 for LRB. As a result, HALP computes a prediction for each eviction in 2.1 μs in comparison to 60 μs that is required by LRB [32]. Tuning LRB's eviction candidates from 64 to 4 would increase P95 byte misses by about 2% (to 0.489/0.575). And this increase would be higher with larger cache sizes given there are more number of objects in cache. In addition, LRB's performance is sensitive to the selection of its memory window, which requires extensive tuning.

Comparison on a public general CDN trace

To test HALP's performance on a general CDN workload, we evaluated HALP on a trace from a Wikipedia CDN node [32]. We mimic LRB evaluation settings in cache sizes, the warmup length, and the byte miss reduction metric (Figure 9(a) in LRB), but we converted the object sizes into uni-size. We

select the uni-size to be 32 KiB to match the average request size of the original trace. We compare HALP with the best-performing cache algorithms in LRB evaluations, i.e., LRB, Adaptive-TinyLFU, LeCaR, B-LRU, and LRU. We ran HALP by our simulator, and baseline algorithms by LRB public simulator. For LRB, we use the hyperparameter values described in the paper and its website. Compared to LRB, HALP does not use the additional categorical feature in the trace.

Figure 10 shows the byte miss reduction of different algorithms compared to an industry standard algorithm B-LRU (LRU-eviction policy using a Bloom filter as admission control [24]). None of the algorithms achieves the best performance across all cache sizes. LRB, Adaptive-TinyLFU, HALP, and B-LRU achieve the best performance individually at cache size 64 GiB, 128 GiB, 256 GiB, 512 GiB. At 1024 GiB, the cache is big enough that all cache algorithms converge. At such cache size B-LRU suffers from its admission control. Our observation for this trace is the frequency of objects remains stable over time, making past frequency a reliable indicator of future frequency and allowing the frequency-based heuristic algorithms such as Adaptive-TinyLFU to perform well. In contrast, the workload on YouTube exhibits strong spatial locality, which means that past frequency is less indicative of future frequency, resulting in a lower performance of the frequency-based heuristic algorithms. Note the differences between these results and Figure 9(a) in LRB are likely due to the uni-size object transformation.

5.6 Hyperparameter Selection

We validate the effect of different hyperparameters. This includes different numbers of eviction candidates, different neural network architectures, and different retrain intervals.

Neural network architecture

HALP uses a simple neural network with one hidden layer. Here we vary the number of hidden neurons in the hidden layers and measure the byte miss ratio.

Fig. 11a shows the relationship between the geometric mean of P95 normalized byte miss ratio of all traces as the number of neurons in the hidden layers increase logarithmic from 1 to 256. We see a marginal benefit by increasing the number of neurons up to 8. Beyond this point, more hidden representations do not help. To keep a safe margin, we select the number of hidden neurons in our deployment to be 20.

Number of eviction candidates

HALP uses this parameter in training and prediction. After candidates are selected by the heuristic policy, it iteratively does pairwise ranking to select the final chunk to evict, and later uses these comparisons to generate training data. We vary the number of eviction candidates in the simulation, and measure the byte miss ratio. Note that this changes training and prediction distributions in lock-step.

Fig. 11b shows the relationship between the geometric mean of P95 normalized byte miss ratio of all traces and the number of candidates selected by the heuristic algorithm varying from 2 to 16. As the number of eviction candidates increases from 2 to 4, the byte miss ratio reduces from 60.4% to 59.3%. Further increasing the number of eviction candidates has a marginal effect. Large numbers of eviction candidates have a marginal benefit of the byte miss ratio, but too large a number may harm the byte miss ratio if the other training hyperparameters are not adjusted accordingly.

The number of pairwise comparisons per eviction increases from 3 to 7 when the number of eviction candidates increases from 4 to 8. This increase in CPU does not justify the less than 1% relative byte miss ratio reduction, as a result HALP uses four eviction candidates and does three pairwise comparisons.

Retrain interval

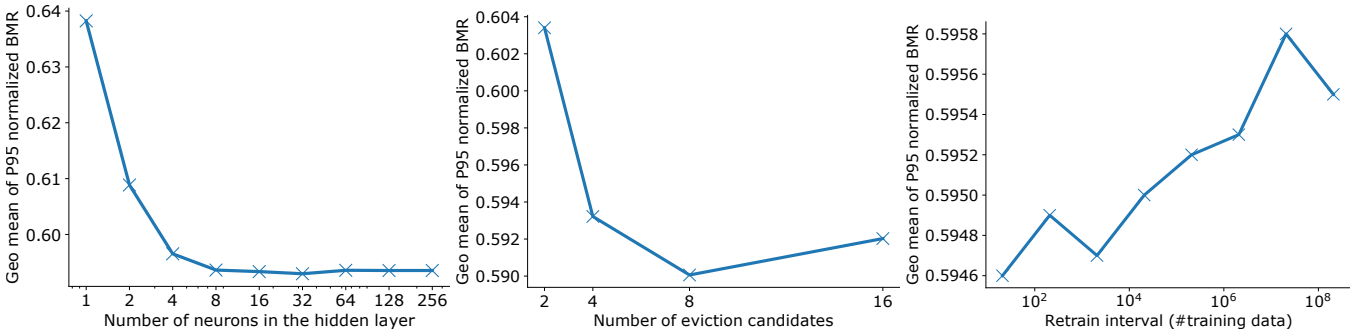
HALP trains online as new requests are processed. We conduct simulation experiments to test different retrain intervals. In order to only test the difference in updating the model online, we increase the trace length to be 6 days from 3 days. We use the first 3 days to train the model in the same retrain interval, and validate that the training loss has been stable. After that, we vary the retrain intervals in the next 3 days, and only measure the byte miss ratio during the latter 3 days.

Fig. 11c shows the relationship between the geometric mean of P95 normalized byte miss ratio of all traces and the retrain intervals. As the retrain intervals increases from processing every 1 new training data input to every 10^8 new training data input, the byte miss ratio slightly increases by less than 0.2%. Our hypothesis is that the traffic pattern change is slow in most traces. But to increase the algorithm robustness, we keep the retrain interval to be every 1024 training data input. This is acceptable in production given the CPU increase is only 1.8% and enables the model to adjust to unpredictable quick workload changes.

6 Related Work

Heuristic-based cache algorithm. Many heuristic-based cache algorithms have been proposed in the past six decades, and from them the most impactful ones include LRU, FIFO, CLOCK [3], SLRU [19], 2Q [18], ARC [25], and TinyLFU [15,16]. Many heuristic algorithms have low computation overhead and provable competitive ratios. But because they are not adaptive enough, they work well in some traces but not in others.

Learned cache eviction. Recently, many learning-based cache algorithms have been proposed to make cache eviction decisions. Table 2 summarizes the state-of-the-art learned cache eviction and admission algorithms. We list four properties of learning-based cache algorithms: target application, whether they are used to make admission or eviction decisions, if they employ online learning, and which underlying machine learning algorithm they employ.



(a) (b) (c)

Figure 11: Geometric mean of P95 normalized byte miss ratio of all traces (a) as the number of neurons in the hidden layers increases logarithmically from 1 to 256. We see a marginal benefit by increasing the number of neurons up to 8. (b) as the number of candidates selected by the heuristic algorithm varies from 2 to 16. As the number of eviction candidates increases from 2 to 4, the byte miss ratio reduces from 60.4% to 59.3%. Further increasing the number of eviction candidates has a marginal effect. (c) as the retrain interval increases by how much training data is processed. As the retrain interval increases from processing every 1 new training data input to every 10^8 new training data input, the byte miss ratio slightly increases by less than 0.2%.

CACHEUS [26] proposes two new heuristic algorithms: SR-LRU, a scan-resistant version of LRU, and CR-LFU, a churn-resistant version of LFU. Then it proposes a regret minimization algorithm to switch between these two experts. As a limitation, the overall algorithm cannot adapt to a new workload if neither of the two experts can adapt to it. In addition, the metadata overhead scales linearly with the number of experts because each one needs to maintain its priority queue. [36] learns next request distribution from tags collected by a distributed tracing framework. It combines a lookup table, a K-Nearest Neighbor approach, and a Transformer model to achieve low overhead and high accuracy. But it has a high learning overhead. LRB [32] uses a regression model to approximate Relaxed Belady, a relaxed oracle algorithm. It uses random sampling to generate eviction candidates and training data. Because of a large number (64) of candidates are needed, the eviction has a high computation overhead. In addition, generating training data with enough critical objects is costly due to the uniform sampling process. And LRB's performance is sensitive to the selection of the memory window (its major hyperparameter).

Parrot [22] and LFO [8] use imitation learning to mimic the oracle algorithm. The objective is to achieve an end-to-end design, but they suffer from a distribution shift. This is because they train their models in an offline fashion, and in practice learning-based cache algorithms have a substantial gap from an oracle, and the objects in the cache as a result differ from a cache that would use an oracle algorithm.

AViC [5] is designed for a video streaming CDN, leveraging the constant speed sequential access patterns, and predicts the time to the next access for the following chunks. However, it has a high implementation overhead because of the complex synchronization between video sessions. Glider [31] targets an eviction policy for CPU caches, and uses an LSTM model for offline analysis. It uses a fast SVM model for an online policy heavily leveraging the program counter (PC) address feature, which is unavailable in the CDN domain.

LHD [6] estimates the hit density of an object between admission and eviction using Bayesian approaches. But it cannot scale with increasing number of features since it does not have a general model for prediction. Predictive Marker [23] is a theoretical work using learning to augment a cache using the Marker algorithm. This idea inspires the design of HALP.

Another line of works [14,20] use reinforcement learning to directly optimize an eviction policy with the target objective. But because cache feedback (hit) can take tens of millions of steps, reinforcement learning approaches suffer from such long feedback and currently have lower performance than supervised learning approaches in practice.

Learned cache admission. In addition to learned eviction policies, many recent research proposed to use learning in cache admission. Cache admission is helpful when a cache has a bottleneck in write constraints (e.g. SSD write amplification and endurance), or a large portion of objects are never reaccessed. The prominent papers include Flashield [17], CacheLib [7], and CacheSack [35]. Because their decision space is more limited than that of eviction algorithms, they have worse performance. HALP's eviction policy can be used jointly with a learned admission policy.

Statistical hypothesis test. Many statistical hypothesis tests have been proposed. But they often focus on using small data, and not on measuring the distribution change. For example, a standard t-test [33] measures the change of mean value.

Algorithm	Year	Target application	Learned admission or eviction	Online learning?	Algorithm
HALP (ours)	2022	CDN	Eviction	Yes	1-hidden-layer MLP, Heuristic + pairwise preference ranking from re-accesses.
CacheSack [35]	2022	Flash cache	Admission	Yes	Greedy optimization
CACHEUS [26]	2021	Storage	Eviction	Yes	Heuristic algorithms w. regret minimization
Learning on distributed traces [36]	2021	Storage	Eviction	No	Lookup Table, K-Nearest Neighbors, Transformers
LRB [32]	2020	CDN	Eviction	Yes	Decision trees
Parrot [22]	2020	CPU	Eviction	No	Transformers
CacheLib [7]	2020	Multipurpose	Admission	No	*Private*
AViC [5]	2019	CDN	Both	No	Decision trees
Glider [31]	2019	CPU	Eviction	Yes	SVM
Flashield [17]	2019	Flash cache	Admission	No	SVM
LHD [6]	2018	KV store	Eviction	Yes	Probability model
LFO [8]	2018	CDN	Eviction	No	Decision trees
Predictive Marker [23]	2018	/	Eviction	/	Learning + Marker algorithm
Harvesting randomness [20]	2017	KV store	Eviction	Yes	Reinforcement learning

Table 2: A summary of state-of-the-art learned cache eviction and admission algorithms

7 Future Work

For future work, we aim to expand HALP to the SSD and HDD caching tiers of the YouTube CDN. We also seek to jointly optimize eviction and admission policies. Another line of future work we plan to explore is to redesign the features and model architecture leveraging existing hardware accelerators. Right now, HALP uses only CPUs and the pairwise comparisons that use the model to pick candidates are subject to the CPU overhead limits acceptable in production. With accelerators like GPUs or TPUs we will be able to explore a larger design space of features and model architectures. One challenge here is how to design an asynchronous batched eviction algorithm to achieve a high utilization of accelerators while preventing it on path of cache operations that require a low latency.

8 Conclusion

This work describes the design, implementation, and evaluation of HALP, a learned caching algorithm that has been deployed to a large-scale production CDN. We also describe an impact distribution analysis method that allows us to measure the impact of deploying a new cache algorithm in a production setting with significant measurement noise. The key insight of HALP is to augment a preexisting heuristic caching policy with machine learning, using the heuristic policy to pick candidates for eviction and the ML model to decide which candidate to evict. The key insight of our impact distribution analysis is modeling machine level measurement noise by comparing machines with HALP deployed against no-op machines.

These design decisions enable HALP's robust byte miss reduction by an average of 9.1%. In addition, these improvements were achieved with a modest CPU overhead of 1.8%.

Acknowledgements

We are grateful to our anonymous reviewers, our shepherd Francis Yan, Ken Barr, Nils Krahnstoever, Jeff Dean, Martin Maas whose extensive comments substantially improved this work. We also thank Yundi Qian and Richard McDougall who contributed to the early stages of the project.

References

[1] LRB open-source simulator. https://github.com/sunnyszy/lrb.

[2] XLA. https://www.tensorflow.org/xla.

[3] *A paging experiment with the multics system*. MIT Press, 1969.

[4] Vijay Kumar Adhikari, Sourabh Jain, and Zhi-Li Zhang. Youtube traffic dynamics and its interplay with a tier-1 isp: An isp perspective. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 431–443, New York, NY, USA, 2010. Association for Computing Machinery.

[5] Zahaib Akhtar, Yaguang Li, Ramesh Govindan, Emir Halepovic, Shuai Hao, Yan Liu, and Subhabrata Sen. Avic: a cache for adaptive bitrate video. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 305–317, 2019.

[6] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *USENIX NSDI*, pages 389–403, 2018.

[7] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[8] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *ACM HotNets*, pages 134–140, 2018.

[9] Christopher J. C. Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. *Proceedings of the 22nd international conference on Machine learning*, 2005.

[10] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. Smartchoices: hybridizing programming and machine learning. *arXiv preprint arXiv:1810.00619*, 2018.

[11] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, page 1–14, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Ludmila Cherkasova. Improving WWW proxies performance with greedy-dual-size-frequency caching policy. Technical report, Hewlett-Packard Laboratories, 1998.

[13] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In Bob Hertzberger, Alfons Hoekstra, and Roy Williams, editors, *High-Performance Computing and Networking*, pages 114–123, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[14] Renato Costa and Jose Pazos. Mlcache: A multi-armed bandit policy for an operating system page cache. Technical report, University of British Columbia, 2017.

[15] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *ACM Middleware*, pages 94–106, 2018.

[16] Gil Einziger and Roy Friedman. TinyLFU: A highly efficient cache admission policy. In *IEEE Euromicro PDP*, pages 146–153, 2014.

[17] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *USENIX NSDI*, pages 65–78, 2019.

[18] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[19] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46, 1994.

[20] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. Harvesting randomness to optimize distributed systems. In *ACM HotNets*, pages 178–184, 2017.

[21] Jie Li, Jinlong Wu, György Dán, Åke Arvidsson, and Maria Kihl. Performance analysis of local caching replacement policies for internet video streaming services. In *2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 341–348, 2014.

[22] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020.

[23] Thodoris Lykouris and Sergei Vassilvtiskii. Competitive caching with machine learned advice. In *International Conference on Machine Learning*, pages 3296–3305. PMLR, 2018.

[24] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR*, 45:52–66, 2015.

[25] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[26] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354, 2021.

[27] Sandvine. The global internet phenomena report january 2022, January 2022. Available at https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2022/Phenomena%20Reports/GIPR%202022/Sandvine%20GIPR%20January%202022.pdf?utm_referrer=https%3A%2F%2Fwww.sandvine.com%2Fphenomena, accessed 06/17/22.

[28] Richard Schooler and Pawel Jurczyk. Streaming media cache for media streaming service, August 27 2019. US Patent 10,397,359.

[29] Nihar Shah and Martin Wainwright. Simple, robust and optimal ranking from pairwise comparisons. *Journal of Machine Learning Research*, 18:1–38, 2018.

[30] Shailesh Shukla. Introducing media cdn—the modern extensible platform for delivering immersive experiences, April 2022. Available at https://cloud.google.com/blog/products/networking/introducing-media-cdn, accessed 09/07/22.

[31] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019.

[32] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[33] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[34] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *USENIX HotStorage*, 2018.

[35] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, 2022.

[36] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. *Proceedings of Machine Learning and Systems*, 3, 2021.

A Details about the loss function and model weight updates

To explain why we use a cross entropy loss: let w denote the neural network weight parameters and $s_w(f(k,t))$ denote the score output of the neural network for features corresponding to cache key k at time t. Assume that the feedback generation process for the pairwise comparison orders the cache key k_1 ahead of k_2 while querying for the first access to either after time t (i.e. k_1 arrives before k_2 for the first access to either of them after time t). In this case, the cross entropy loss penalizes the loss according to how much the predicted score for k_2 exceeds that of k_1. Specifically, with $\Delta = s_w(f(k_2,t)) - s_w(f(k_1,t))$, as the difference in scores, the neural network weight parameters w are adjusted based on the gradient of the loss function $log(1+e^{\Delta})$. When $\Delta \ll 0$, the loss is close to 0, but when $\Delta \gg 0$, the loss is linear in Δ and varies smoothly around 0.

B Analysis of a simple model for reranking

Let (U,H,L) be a triple of jointly distributed random variables. Let

$$(U_1,H_1,L_1),\ldots,(U_n,H_n,L_n)$$

be id samples $\sim (U,H,L)$. The value H_i corresponds to the score for item i predicted by some (heuristic) ranking policy and similarly, L_i corresponds to the score predicted by, (say a learned) ranking policy L. U_i denotes the true utility from object i, but this is a latent variable. The problem is to choose an index i such that U_i is as large as possible. Define the expected utility of a policy $X \in \{U,H,L\}$ as follows:

$$\mathcal{U}(X) = \mathbb{E}[U_{\arg\max_i(X_i)}]$$

We'd like to pick $i^* \triangleq \arg\max_i(U_i)$, which achieves the optimal utility $\mathcal{U}(U)$, but we only observe (H,L) with U being a latent variable. Given two ranking policies H and L, define an aggregate selection policy on them, $\pi(H,L)$ as a map [2], $\pi : \mathbb{R}^{2n} \mapsto [n]$, and the resulting expected utility as $\mathcal{U}(\pi) \triangleq \mathbb{E}[U_{\pi(H,L)}]$. Next, we describe and analyze a simple aggregation strategy that we've discovered to be useful in learned caching. Let $\lambda_X(i)$ be defined as the item with ranked

[2] $[n]$ denotes the set $\{1,\ldots,n\}$

order[3] i when using the ranking policy X for $X \in \{U, H, L\}$. For each $k \in [n]$, define $\pi_k(H,L)$ as the item chosen when re-ranking the top k items in the heuristic H according to L.

$$\pi_k(H,L) \triangleq \lambda_H(j), \texttt{ where } j = \arg\max_{i \in [k]} L_{\lambda_H(i)}$$

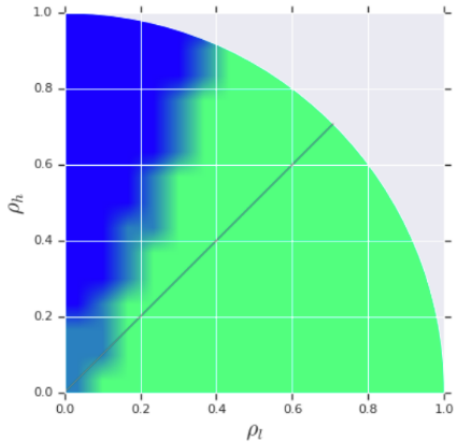


Figure 12: The benefit of rank aggregation evaluated over various configurations for the correlation coefficients based on a sample of $N = 20$ items. The diagonal line indicates points at which $\rho_H = \rho_L$. The green region indicates configurations where $\mathcal{U}(\pi_2(H,L)) > \mathcal{U}(H)$ based on a 95% confidence interval generated from bootstrap estimates of the sample mean. the blue region indicates areas where $\mathcal{U}(\pi_2(H,L)) > \mathcal{U}(H)$ with at least a 95% CI. The grey areas are where the confidence interval overlaps with 0. Interestingly, even when $\rho_H > \rho_L$, it could be advantageous to switch to the lightweight reranking of just the top two items despite having a poor (e.g., learned) policy L. Our experiments indicate that as we increase n, it is better to uniformly switch from H to $\pi_2(H,L)$ for all configurations.

The notation implies $\pi_1(H,L) = \lambda_H(1)$ and $\pi_n(H,L) = \lambda_L(1)$. In other words, $\mathcal{U}(\pi_1(H,L)) = \mathcal{U}(H)$ and $\mathcal{U}(\pi_n(H,L)) = \mathcal{U}(L)$. To understand precisely when the proposed aggregation might help, we now make some assumptions to help with mathematical tractability, analysis and visualization. Let $\rho_H \geq 0, \rho_L \geq 0$ be such that $\rho_H^2 + \rho_L^2 \leq 1$, and consider the jointly Gaussian distribution,

$$(U,H,L) \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} 1 & \rho_H & \rho_L \\ \rho_H & 1 & 0 \\ \rho_L & 0 & 1 \end{bmatrix}\right)$$

It is clear that $\mathcal{U}(H)$ and $\mathcal{U}(L)$ are monotone in ρ_H, ρ_L respectively under the above assumptions. To map the above model to a motivating practical scenario, think of H as an efficient heuristic strategy (e.g. LRU). L could be imagined to be a learned policy that is (1) expensive to evaluate (2) not always safe, i.e. we can end up with $\rho_L < \rho_H$. The proposed mechanism addresses both of these issues simultaneously. For (1) we only need to invoke L on at most e.g. $k = 2$ items, and for (2) the below claim argues that we get an improved outcome, $\mathcal{U}(\pi_2(H,L))$ compared to the baseline strategy $\mathcal{U}(H)$ (which is also naturally greater than $\mathcal{U}(L)$, when $\rho_H > \rho_L$). Based on numerical analysis, we observe, and hypothesize more generally, that as $n \to \infty$, the re-ranking strategy improves over the pure heuristic policy, i.e. $\mathcal{U}(\pi_2(H,L)) > \mathcal{U}(H)$. The above hypothesis applies to arbitrary positive values of ρ_H and ρ_L. It says that we can improve on H even with a worse alternate policy L. This helps give some evidence for why such a strategy appears to be "safe" in terms of improvement over the baseline. In Figure 12, we plot the situation numerically for $N = 20$, for all possible problem configurations of ρ_H, ρ_L.

[3] order 1 is the largest item.

OpenLoRa: Validating LoRa Implementations through an Extensible and Open-sourced Framework

Manan Mishra
University of Wisconsin-Madison

Daniel Koch
University of Wisconsin-Madison

Muhammad Osama Shahid
University of Wisconsin-Madison

Bhuvana Krishnaswamy
University of Wisconsin-Madison

Krishna Chintalapudi
Microsoft Research

Suman Banerjee
University of Wisconsin-Madison

Abstract

LoRa is one of the most widely used LPWAN communication techniques operating in the unlicensed sub-GHz ISM bands. Its long range however also results in increased interference from other LoRa and non-LoRa networks, undermining network throughput due to packet collisions. This has motivated extensive research in the area of collision resolution techniques for concurrent LoRa transmissions and continues to be a topic of interest. In this paper, we verify the implementation and efficacy of four of the most recent works on LoRa packet collisions, in addition to standard LoRa. We implement *OpenLoRa*, an open-source, unified platform to evaluate these works and extensible for future researchers to compare against existing works. We implement each of the four techniques in Python as well as separate the demodulator and decoder to provide benchmarks for future demodulators that can be plugged into the framework for fair and easy comparison against existing works. Our evaluation indicates that existing contention resolution techniques fall short in their throughput performance in practical deployments, especially due to poor packet detection in low and ultra-low SNR regimes.

1 Introduction

LoRa is one of the most widely used Low Power Wide Area Network (LPWAN) technologies for IoT applications such as smart cities [1, 2], smart agriculture [3, 4], and industrial IoT [5, 6]. LoRa's popularity stems from its long operating range, low power consumption, low-cost, and ease of deployment [7–10]. Its long range however, is a double-edged sword as it also results in increased interference from other independently deployed LoRa networks, leading to poor network throughput due to packet collisions [11, 12]. Ghena et.al [13] show that LoRa falls short of meeting the requirements for a large variety of IoT applications due to two key reasons : (a) under-utilization of the network capacity and (b) lack of co-existence between networks.

Recently, a large number of *LoRa collision resolution* techniques have been proposed to address the above challenges: Choir [14], FTrack [15], NScale [16], CoLoRa [17], mLoRa [18], CIC [19], Pyramid [20], and AlignTrack [21]. These techniques develop novel LoRa de-modulation algorithms that can simultaneously decode multiple colliding LoRa packets to improve network throughput and address scalability challenge faced by LoRa networks.

In this paper, we seek to compare and verify the efficacy of state-of-the-art in LoRa collision resolution algorithms and ask the question, *"How effective are state-of-the-art collision resolution techniques in improving network throughput?"* . Our goal is to evaluate and analyze various existing techniques in a variety of important scenarios such as indoor/outdoor and low/high-SNR networks. Towards this evaluation, we have developed *OpenLoRa*, an extensible, open-source framework using Python to implement each of the demodulators and design an evaluation pipeline, along with extensive datasets that can be used for benchmarking future works in LoRa receiver designs. To this end, we pick four recent techniques, (i) FTrack [15], (ii) NScale [16], (iii) CoLoRa [17], and (iv) CIC [19] that provide public implementations.

The motivation for our paper stems from several key gaps in the available implementations and evaluations.

Lack of throughput evaluations. While increasing network throughput (in kilo bits per second) is their key goal, most LoRa packet collision resolution literature evaluates the performance of the demodulator only, which outputs data symbols rather than bits; this can perhaps be attributed to the difficult task of recreating LoRa's encoder and decoder. In the absence of the decoder, one can only evaluate the average symbol error rate, but not bit or packet error rates and hence network throughput in kbps. As we demonstrate in Section §5.1, lower symbol error rates do not necessarily translate to lower packet error rates and corresponding higher network throughput. In fact, seemingly lower symbol error rates compared to standard LoRa might still result in lower throughput!

Lack of co-existence evaluation. Although co-existence of LoRa networks has been identified as a key challenge [11–13], to the best of our knowledge existing literature has not studied the impact of interference due to other LoRa and non-LoRa

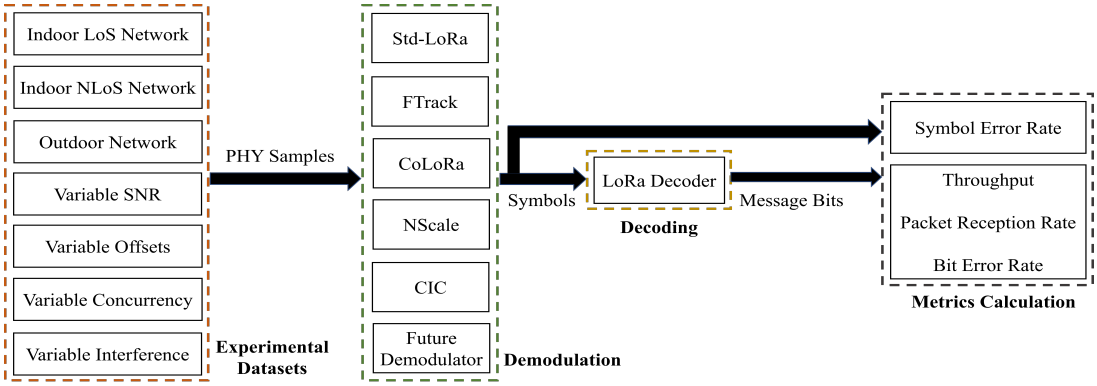


Figure 1: Overview of *OpenLoRa*, the proposed open-source framework

networks on the efficiency of collision resolution. In this paper, we design and perform a new set of experiments to measure the impact of LoRa interference from networks with different SF and non-LoRa interference such as FSK on the network throughput performance.

Lack of a uniform evaluation and a benchmark datasets. Understanding and analyzing the performance of various techniques in different settings is crucial to future research. Existing evaluations differ in terms of evaluation metrics, methodology, scenarios that impact performance such as signal-to-noise-ratio (SNR), indoor/outdoor settings, nature of traffic, effect of bursts etc. In some cases, these have only been studied using simulations. In this paper, we create a set of benchmark datasets spanning various important LoRa scenarios that can be used to evaluate uniformly. In our experiments we find that existing techniques under-perform compared to standard LoRa in very low SNR scenarios.

Implementation variability. Current implementations do not use a common tool: Python, GNURadio, and MATLAB are some of the tools used. Additionally, each of the implementations have a different data preprocessing methodology, making it a challenging task to input a sample file and determine the metrics of interest. Therefore, despite the availability of public code repository, it is a challenging task to input a new file and obtain the performance of the demodulator.

In order to address the above gaps, we have implemented an evaluation framework (depicted in Figure 1) with the goal of providing a common framework to benchmark existing methods. We believe that our extensible framework will help future researchers evaluate LoRa collision resolution techniques uniformly against prior works with common datasets, and analyze them. *OpenLoRa* includes a pipeline of four key stages. First, a suite of experimental datasets comprising received raw samples of LoRa transmissions obtained from various experimental deployments, specifically designed to evaluate various important aspects of collision resolution algorithms. Second, a uniform Python based interface to interact with each demodulator. A future LoRa demodulator algorithm can be simply plugged into this framework and compared against other implementations on the metrics of interest for real-world deployments. Third, a standard LoRa decoder that can convert symbols generated by any demodulator into packets, so that we can measure bit error rates, packet error rates, and throughput. Last but not the least we provide a suite of important metrics such as bit error rate, packet reception rate, and network throughput.

In summary, we make the following contributions towards our verification of state-of-the-art LoRa demodulators:

- We present *OpenLoRa*, an extensible, open-source framework to evaluate and compare different techniques that we hope future researchers will be able to use (provided in GitHub repository[1] as well as in a Docker container[2] enabling environment-independence to run the demodulators locally). Our framework comprises benchmark datasets, a standard interface to plug in various demodulators, a LoRa decoder that outputs bits and a suite of relevant metrics.
- We implement and verify the performance of four state-of-the-art LoRa collision resolution demodulators and standard LoRa, comparing their throughput (in kbps) improvements. We find a surprising fact that even though many techniques decode more symbols on average than standard LoRa, this does not necessarily translate to throughput improvements. As the network traffic increases in long-range outdoor scenarios, standard LoRa outperforms all existing techniques.
- In order to cross-validate the results reported in original works, we recreate the key experimental scenarios presented by each and compare the results. Our results are in line with the results in the respective papers evaluated. This extra effort validates the fidelity of our framework and implementation of various demodulators.
- We develop a web interface[3] for users to easily add new custom demodulation techniques for benchmarking and analyze the performance of implemented techniques.

[1] https://github.com/UW-CONNECT/OpenLora
[2] Linked in OpenLoRa Github page.
[3] https://openlora.wisc.edu

- We implement and validate the standard LoRa decoder in Python, allowing the comparison of different demodulation techniques based on end-to-end metrics such as throughput (in kbps) and the number of successfully received packets after error correction. We hope this openly accessible implementation enables future researchers to evaluate their works based on similar end-user metrics.
- We design new experiments to study the effects of collisions under extremely-low SNR, interference from LoRa networks with different spreading factor and non-LoRa interference such as Frequency-Shift-Keying from other networks on the throughput performance.

2 LoRa Demodulators Validated

In LoRa, data is modulated using a Chirp Spread Spectrum (CSS) scheme which confers it long range and sub-noise decoding ability. We present details on LoRa's modulation/demodulation and effects of collision on network throughput in Appendix A. Choir [14] is a pioneering work in LoRa collision resolution with the goal of improving network throughput. It leverages the inherent hardware imperfections in the radio of LoRa transmitters and distinguishes colliding packets by uniquely mapping their imperfections to the transmitters. mLoRa [18] leverages Successive Interference Cancellation to iteratively decode the symbols with the highest power and remove them from consideration. In this paper, we implement four demodulator algorithms that improve upon Choir and mLoRa to decode multi-packet collisions. We discuss these demodulators in the rest of the section.

2.1 FTrack [15]

FTrack is one of the first approaches to use time and frequency domain features to resolve LoRa collisions. FTrack relies on Short Time Fourier Transform (STFT) to obtain time and frequency features. FTrack proposes to apply STFT on the dechirped LoRa symbol to leverage the spread spectrum gain as well as to remove the linear change in frequency with time. Appendix B.1 explains how FTrack chooses an appropriately sized window and leverages time-frequency resolution to resolve collisions.

2.2 CoLoRa [17]

CoLoRa, similar to FTrack, leverages time offsets and frequency features to resolve collisions. CoLoRa observes that collided packets are misaligned in time and therefore have different lengths of symbol segments appearing in the demodulation window. This results in FFT peaks with heights proportional to the length of the symbol in the current demodulation window. CoLoRa also observes that the ratio of FFT peak between two consecutive windows remains the same throughout a packet. It uses these key insights to translate time offsets to frequency features and differentiate colliding packets. Details on CoLoRa's demodulation window choice and the use of peak ratios can be found in Appendix B.2.

2.3 NScale [16]

As the range increases, the SNR of LoRa packets decreases and the relative performance improvement of FTrack and CoLoRa degrades. NScale [16] focuses on decoding packet collisions at SNRs as low as -10dB. Similar to CoLoRa, it translates the timing offsets to frequency features and further amplifies the time offsets by non-stationary signal scaling. NScale's strength lies in its ability to decode and resolve LoRa packet collisions at SNRs below -10 dB. In Appendix B.3, we explain how NScale achieves sub-noise decodability.

2.4 Concurrent Interference Cancellation [19]

Concurrent Interference Cancellation (CIC) also leverages time and frequency domain analysis to decode multi-packet collisions. CIC identifies that due to Heisenberg's Time Frequency Uncertainty Principle, one can achieve either the best frequency resolution or best time resolution, not both. CIC attempts at getting the best of both resolutions by accumulating multiple windows of varying lengths, resulting in varying time and frequency resolutions. Appendix B.4 explains CIC's technique to resolve packet collisions.

3 Framework Implementation

We implement *OpenLoRa* and evaluate standard LoRa, FTrack, CoLoRa, NScale, and CIC in a uniform framework using Python as illustrated in Fig. 1. The extensible framework also provides the ability to add a new demodulator and evaluate its performance against the implemented techniques over the datasets collected over a range of scenarios.

We have built an easy-to-use web interface to help users analyze the implemented demodulators' performance in more detail. We also provide the option to plug-in one's own new demodulator for benchmarking, with a few simple steps. Some screenshots to walk through the web page are provided in Fig. 2. The homepage asks for user selection to either add a new demodulator or run existing techniques as shown in Fig. 2(a). Fig 2(b-c) show the flow to add a new custom demodulator with the user downloading the existing framework, adding their files, testing on a sample dataset and uploading to run and compare against already implemented techniques. Similarly, Fig 2(d-f) show the flow to analyze the existing demodulators in detail by choosing a scenario and configuration among those presented in Section §5 and presenting detailed data points as well as plots.

OpenLoRa has a pipeline of four major blocks: datasets, interface to the demodulators, decoder, and metrics. We describe each one of these blocks in detail below.

3.1 Datasets : Experimental setup for data collection

For a thorough and fair evaluation of the demodulation algorithms, a uniform set of data sample files is necessary. We deployed practical networks of LoRa nodes in varying configurations (SF and BW) and scenarios (Line-of-Sight, SNR) as shown in Fig 3. We believe the presented evaluation accounts for a comprehensive (but not exhaustive) set of conditions to assess the feasibility of real-world usage, and serve as a benchmark to gauge the performance of future work in this domain. We will make the datasets, along with the ground truth, publicly available for reproducibility. We believe this will help other researchers to evaluate their work on a variety of scenarios as well.

We used 20 battery-powered Adafruit Feather M0 with RFM95 [22] as LoRa transmitters deployed in the following settings, with a USRP B200 as the receiver. Unless otherwise mentioned, by default, each transmitter sends a known message, while the duty cycle and the load follows a Poisson distribution. The arduino code flashed onto Adafruit Feather M0 boards and the circuit diagram to setup the boards can be found in the github repository [4] under the folder Experiment_ Setup. At each of the locations (red dots in Fig. 3), the individual transmitters were verified to be reachable from the receiver i.e.,in the absence of collisions, LoRa packets from each of the transmitters were successfully received using the standard LoRa demodulator. Each data point in the evaluation results was averaged over multiple iterations of data transmissions (ranging from a minimum of 200 packets to over 6000 packets depending on the scenario). The details on each experimental setup and the methodology specific to each experiment can be found in the Appendix E. The three settings used in our experiments are:

1. **Indoor Line-of-Sight (LoS) :** This setting serves as a high-SNR scenario in our experiments. Twenty LoRa nodes were deployed in a 15m x 10m room, distributed uniformly in LoS with the receiver as shown in Fig. 3 (a). This setting emulates a deployment similar to a smart home, with IoT nodes distributed in a small space, many of which have LoS to the receiver. In this setup, we performed experiments which required precise control over collision parameters, parametric analysis with controlled time offset between colliding packets, concurrent collisions, and high SNR collisions.

2. **Indoor Non-Line-of-Sight (NLoS) :** This setting serves as a low-SNR setting inside a building spanning an area of 150 m x 75 m (per floor) over two floors. The transmitters were deployed as per Fig. 3 (c) and (d), showing the distribution of nodes on first and second floor respectively. The nodes were distributed in NLoS setting, separated from the receiver by multiple concrete walls, elevator shafts, and metal obstructions. This setting emulates a typical deployment of IoT nodes in an indoor office or factory building, with human movement as well as wireless traffic interfering with active transmissions. This setup was used to collect datasets for collisions with increasing aggregate transmission rate. We also performed controlled interference experiments here.

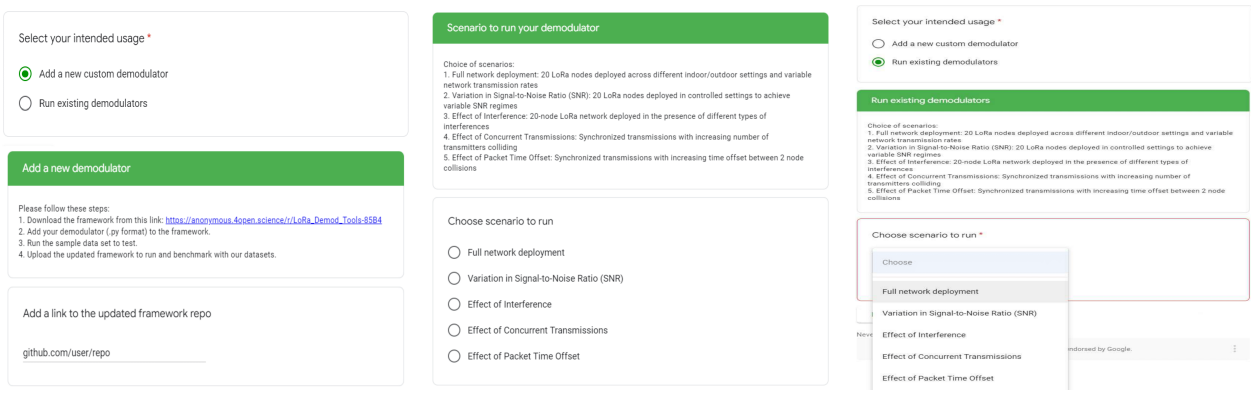
3. **Long-range outdoor :** This setting serves as an extremely low-SNR setting with nodes at distances of 1 to 8.25 km from the receiver. The nodes were distributed across urban areas and along a lake shore as seen in Fig. 3 (b). This setting emulates applications which particularly befit the use of LoRa modulation where communication over long distances is necessary, such as city monitoring applications or sensor deployment across huge agricultural fields. We collected datasets for various transmission rates in extremely low-SNR conditions.

[4] https://github.com/UW-CONNECT/OpenLora/tree/main/Experiment_Setup

3.2 Demodulators Block

Fig. 13 in Appendix C.1 shows an overview of the Python implementation and how the demodulators were integrated into the overall flow of the framework. Appendix C.2 also describes the organization of the implementation code into Python modules. We thank the authors of each of the works presented here for sharing their implementations with us. In addition to re-implementing the code in Python, the following refinements were made to accept and pre-process a variety of datasets, as well as to reduce the computation time of demodulation using parallel processes:

FTrack : FTrack processes the entire input data file at once. While it is feasible for slices with a few packets, it is computationally intensive for real-life data capture with tens of million of samples. This also posed a challenge in the memory constraints for practical datasets such as those from our experiments. To mitigate this, we built a pre-processing block that separates out the active data transmission from silence periods based on energy thresholds of the signal and passes only the valid transmission data to the demodulator. FTrack's demodulator algorithm uses a number of threshold parameters for separating collisions such as peak power ratio, noise floor power, and ratio of stronger to weaker peaks, among others. These parameters need to be modified based on the input dataset for accurate functioning of the algorithm. We set these parameters based on our datasets empirically, however a formal procedure for the derivation of these thresholds would be highly beneficial for any user.



(a) (b) (c)

Figure 2: Screenshots from the system interface web page : (a) Flow to add a new demodulator (b) choose a scenario to benchmark (c) Flow to run an existing demodulator for analysis with network setting and SF configuration

NScale : In addition to re-implementing in Python, we parameterized NScale implementation to accept datasets with different LoRa configurations. Various parameters and thresholds for recognizing sync words, clustering packets from the same transmitters together, and choosing the correct demodulation window were generalized to get the optimal demodulation results. The original implementation resulted in missed or repeated symbols when any of the collided packets was split approximately in half by the demodulation window due to ambiguity in length and peak ratios. We implemented an up-chirp correlation-based packet identification, as described in the paper, and aligned the demodulation window appropriately to avoid this corner case.

CoLoRa :We implemented an up-chirp correlation based strategy to process the input file parallelly using Python multiprocessing. This enabled us to identify the start of packets and choose a reception window, ensuring any symbol's split ratio to be between 1/3 and 3 as derived in the paper. We implemented the Akaike Information Criterion based algorithm to detect the onset of the received packet.

CIC: CIC iteratively decodes packets and hence stores the entire data transmission session. At higher data rates, frequent transmissions can lead to very long packet transmissions that can overflow the memory. We overcome this challenge by splitting active data transmissions longer than a threshold into multiple sessions and process them separately.

Std-LoRa Demodulator: For the implementation of the standard LoRa demodulator, we used *rpp0/gr-LoRa* [23], an open source GNURadio block for decoding LoRa packets. *gr-LoRa* has demodulator and decoder integrated as a single block. We split the two into separate blocks and used the demodulator as part of our std-LoRa demodulator implementation. It looks for the strongest peak in each demodulation window and tries to find consecutive occurrences of the same symbol to detect preambles. Once all the preambles are detected and the preamble indices saved, it continues finding the strongest peak in every demodulation window of the detected packet and outputs the corresponding symbols.

3.3 Standard LoRa Decoder Block

Majority of the existing works focus on demodulator performance as a function of the symbols received. A LoRa receiver consists of a demodulator followed by a decoder. The decoder maps received symbols to message. LoRa decoder (and encoder) is responsible for performing forward error correction using Hamming codes, interleaving, whitening, and gray coding to decode symbols to bits and then message. LoRa supports four coding rates ranging from 4/5 having the least redundancy to 4/8 having the highest. This redundancy allows the LoRa signal to endure interferences and correct small errors. While the demodulated symbols provide some understanding of the receiver, the output of the decoder is required to obtain metrics such as throughput, bit error rate, and number of packets successfully received. Additionally, the number of symbol errors that can be corrected by the receiver depends on the coding rate used by the LoRa transmitter.

In order to evaluate the throughput performances of the demodulators, we implemented LoRa decoder in Python. We used an open-source LoRa receiver framework [23] that jointly demodulates and decodes LoRa samples and separated the decoder and demodulator modules. We then implemented the decoder module separately in Python such that the output of any demodulator can be decoded. This allows for a modular implementation of a LoRa receiver pipeline. Our decoder implementation first extracts the symbols corresponding to LoRa header from the demodulator output. It infers the payload CRC and payload length information by reversing the process of Gray encoding, interleaving, shuffling, and Hamming encoding. Finally, these operations are performed on the symbols corresponding to the payload and the final message is displayed as the output. This process is repeated for

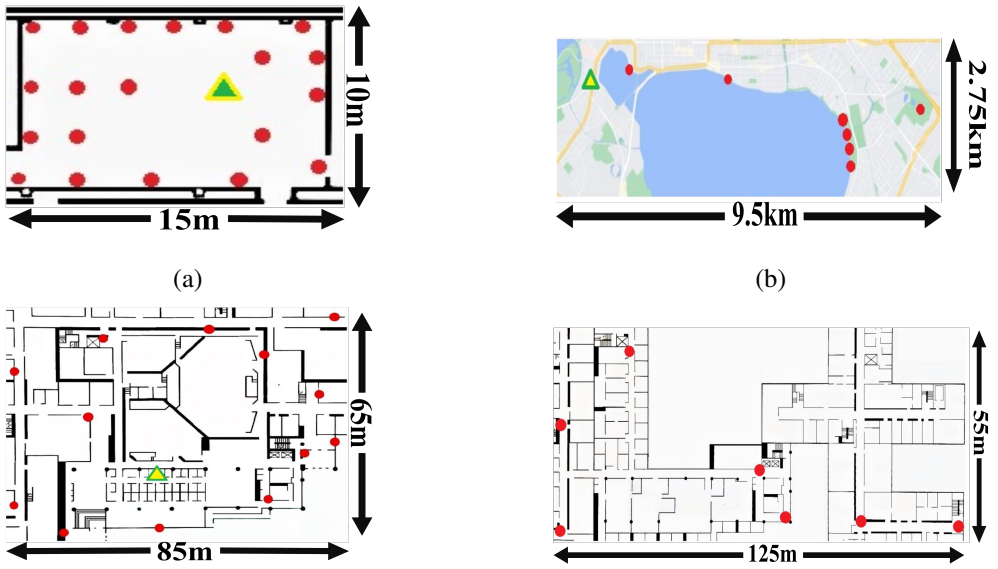


Figure 3: Experimental Setup for LoRa deployments. (a) Indoor LoS, (b) Outdoor, (c) 1st and (d) 2nd Floor Indoor NLoS. Triangle:Base Station, Circles:Transmitters

each demodulator's symbols and the final output is used to calculate the metrics. We validate this Python implementation of the standard LoRa decoder in Section §4. As shown in Fig. 13 in Appendix C.1, the implemented decoder is independent of the demodulators and thus can be integrated with any other demodulator in the pipeline. This openly accessible implementation we have made available, can be used by other researchers to evaluate their demodulator on end-user metrics.

3.4 Metrics

The final module of *OpenLoRa* is the set of metrics that evaluate the end-to-end-performance. This module takes in the demodulated symbols and decoded bits and outputs the calculated metrics. We use the following definitions for the metrics exposed by our framework:

1. Symbol Error Rate (SER): SER is calculated as the ratio of number of incorrect symbols to the total number of transmitted symbols. This metric evaluates the efficiency of demodulator algorithms. With prior knowledge of the transmitted symbols, a one-to-one comparison is used to determine the number of incorrect symbols per packet.

2. Packet Reception Rate (PRR): PRR is calculated as the ratio of number of correct packets received to the total number of transmitted packets. A packet is considered correct if and only if the received message (after error correction) is equal to the transmitted message. Thus, PRR is determined from the output of the decoder.

3. Throughput: This is the one of the most important metrics from an end-to-end workflow perspective. Throughput is defined as the number of correct bits received per second. Towards calculating throughput, we only consider correct packets i.e.,packets where all the received bits are correct.

Calculating metrics from the decoded bits provides a holistic evaluation of the receiver performance of the demodulator algorithms, which we believe is critical in practical LoRa deployments. We will use the metrics used in the papers being validated first, in order to cross-validate our implementation in Section §4. Then, we will present our evaluations using the end-to-end metrics of Packet Reception Rate and Throughput obtained from the output of decoder in Section §5 under varying settings, configurations, and scenarios.

4 Cross Validation of the Demodulators

In order to validate the fidelity of our framework and implementation of the various demodulators, we recreate a representative result from each of the four papers considered. As mentioned in Section §2, each existing work proposes unique techniques to resolve multi-packet collisions. Existing works compare their performance against standard LoRa and a few other existing state-of-the-art. However, each one of them uses a dataset that has been captured in different experimental scenarios, using different configurations for Spreading Factor, Bandwidth, duty cycle and concurrency.

In this section, we recreate the experimental setup as discussed in the original papers to the best of our knowledge and

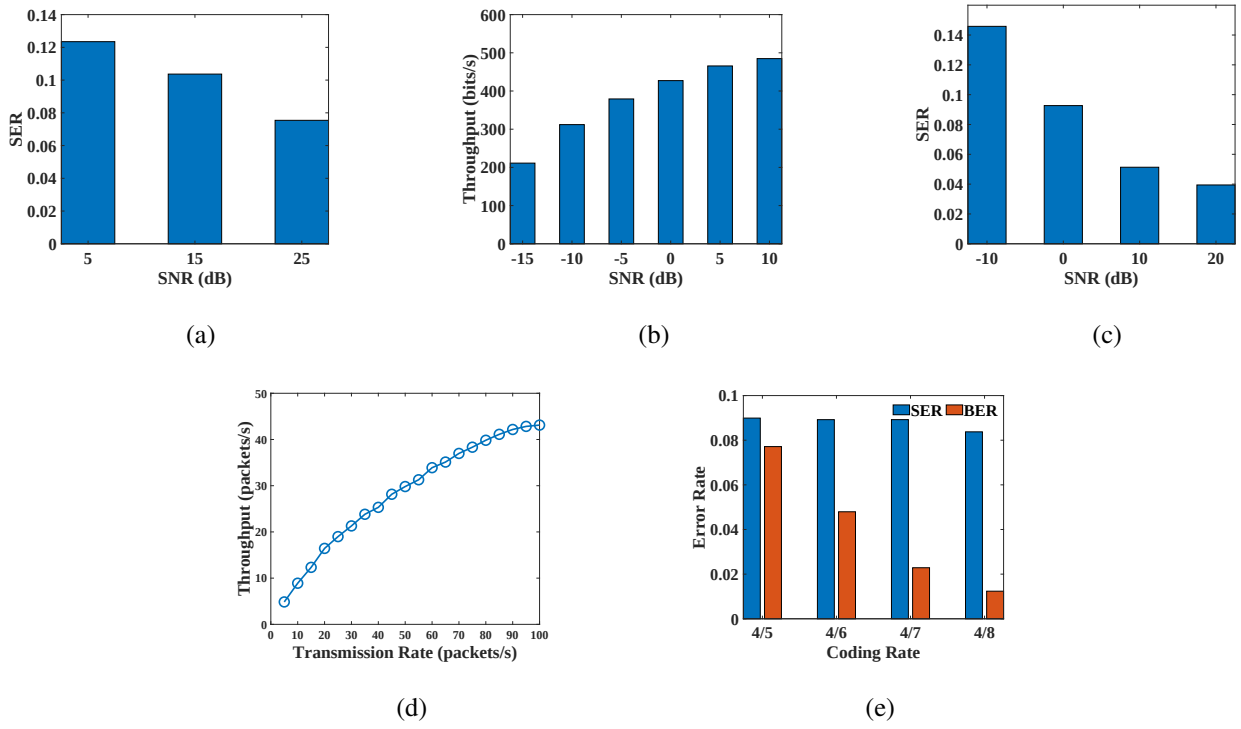


Figure 4: Cross-validation of results from original papers : (a) FTrack: SER vs SNR (b) CoLoRa: Network Throughput vs SNR (c) NScale: SER vs SNR (d) CIC: Throughput vs Aggregate rate (e) Cross-validation of Std-LoRa Decoder : SER and BER for varying coding rates of a single transmitter

report the same result metrics. The goal of this exercise is two-fold : 1) To validate our implementation by recreating the original reported results in each of the papers considered 2) To provide a module in our framework for future works to recreate the existing works and their results.

We have selected the following results to recreate :

1. FTrack's Fig. 14 : SER vs SNR
2. CoLoRa's Fig. 12 : Throughput vs SNR
3. NScale's Fig. 11a : SER vs SER
4. CIC's Fig. 28 : Throughput vs transmission rate

FTrack: Following FTrack's setup, we created the two-node collisions initiated by beacon packets. Upon listening to the beacon packets, the two LoRa nodes send packets with a random delay within a packet duration ensuring collisions. Each node transmitted packets of fixed length with SF8 and BW 250kHz. To achieve the SNR ranges of low (<5dB), medium (5 – 20 dB) and high (>20dB) as mentioned in the paper, we installed nodes at appropriate distances to achieve SNRs of 5, 15 and 25dB respectively. Fig. 4 (a) shows that the SER decreases with increasing SNR, implying better collision resolution at high SNRs by FTrack. SER of ≈ 0.1 is in complete agreement with the results presented in the original work.

CoLoRa: We design the experiment with a 20-node architecture that closely follows the description in the original paper. Each node transmitted SF10, BW 250kHz packets at a fixed rate of 1 packet per second. As mentioned in the paper, we captured data for high SNR packets and then added Additive White Gaussian Noise (AWGN) on the captured data to vary its SNR in a controlled manner. Fig. 4 (b) shows the throughput that we obtain for varying levels of emulated SNR. As the SNR increases from -15dB to 10dB, the network throughput increases from 200 bits/s to over 400 bits/s. This result is in complete agreement with the original work.

NScale: For NScale, we generated beacon-initiated collisions with the responding nodes transmitting packets with SF10 and BW 125kHz, following the experimental setup of NScale. We installed nodes at distances that ensured SNRs of -10, 0, 10, and 20 dB. As shown in Fig. 4 (c), lower SER of 0.04 at 20dB SNR indicates strong collision resolution capability of NScale. SER increases slightly for -10dB SNR to approximately 0.1, close to the results presented in NScale. This trend is similar to the original results presented and verifies that NScale is able to achieve low SER even at -10dB SNR.

CIC: In order to recreate the result in CIC, we used the open source data-set provided by CIC in the GitHub repo instead of designing a new experiment. The results, as shown in Fig. 4

(d) are in complete agreement with actual results presented in the paper. CIC's throughput improves from 5 packets per second to over 40 packets per second as the aggregate rate increases, indicating its ability to resolve collisions.

Std-LoRa Decoder : We also validate the correctness of our decoder implementation, since it forms the basis of throughput in Section §5. To validate the decoder, we designed an experiment to study the impact of Forward Error Correction (FEC) over raw symbols. LoRa offers 4 coding rates : 4/5, 4/6, 4/7, 4/8 where 4/8 implies twice as many redundant bits as data bits. Therefore, higher coding rates lead to better reliability and resilience to bit errors, but lowers effective data-rate. We placed a LoRa transmitter in a NLoS setting from the receiver to ensure low SNR of approximately -15dB. We transmit SF8 packets with 250 kHz BW and vary the coding rate of the transmitter for the same message. Fig 4 (e) shows SER and BER for varying coding rates. We can observe that the SER remains almost constant, since SER is unaffected by the coding rate and only depends on the SNR. However as the coding rate varies from 4/5 to 4/8, redundancy increases and as expected, the decoder is able to correct more errors such that the Bit Error Rate decreases from 7.7% to 1.2%. This result establishes the expected decoder operation and validates our implementation of the standard LoRa decoder. This reliability comes at the cost of longer packet time with the coding rate of 4/8 needing 64 ms to transmit the same data as compared to 49 ms for 4/5 coding rate.

To summarize, we recreated one representative result from each of the papers being validated to verify the correctness of our implementation as well as validate the results with the exact same experimental setup described in the respective papers. We show that the results recreated are in complete agreement with that in the original papers. We also validated the decoder block implemented by comparing the SER and BER performance for varying coding rates. In the next section, we design new experiments to further test the throughput performance of each of these demodulators integrated with our LoRa decoder block.

5 Experimental Evaluation

We evaluate the performance of the five demodulators on several metrics and configurations. We aim to answer the following questions in this section through our experiments:

- Do collision resolution techniques improve the overall network throughput in the presence of collisions?
- What is the impact of variations in the SNR for different transmitters on decoding multi-packet collisions?
- What is the impact of LoRa and other non-LoRa narrowband interferers in decoding concurrent transmissions?
- How many concurrent transmissions can be successfully decoded from the cumulative signal?
- How does the time offset between two colliding packets affect the demodulation and throughput performance?

In the rest of this section, we describe the experiments performed and the results observed to answer these questions. Most of the experiments were repeated for two different configurations: SF8, BW 125kHz and SF10, BW 250kHz, to represent low and high air-times respectively. Unless otherwise mentioned, the default configuration is the larger packet airtime with SF10, BW 250kHz, and a coding rate of 4/5.

5.1 Impact of transmission rate on Network Throughput

In this section we evaluate the overall network throughput achieved by various techniques for *Indoor LoS*, *Indoor NLoS* and *Outdoor* data sets for a 20-node network. The transmission rate of each node is varied from 1 packet/s to 5 packets/s, resulting in an aggregate network rate of 20 packets/s to 100 packets/s as the x-axis. Packets were generated with inter-arrival times following an exponential distribution. We collected upto 6000 packets for each iteration of this experiment for different transmission rates, repeated for both SF, BW configurations. Each data point in the figures presented is averaged over all these packets.

Indoor LoS: In this scenario, packets from almost all the nodes are captured at high SNR therefore, all techniques perform at their best. We present the results and analysis of this scenario in Appendix D.1.

Indoor NLoS: At low-SNR indoor NLoS scenario, where 20 nodes are deployed across two floors in an office building, a trend similar to indoor-LoS can be observed with increasing Transmission Rate. As shown in Fig. 5 (a) and (b), at low SNR, the average network throughput is lower than that of higher SNR for most demodulators. All the demodulators achieve their peak throughput at 40 packets/s in case of SF8 transmissions, beyond which it decreases with an increase in the aggregate transmission rate. In case of SF10 transmissions, a much sharper descent in the throughput curve can be seen due to the compounding of the lower SNR with more severe collisions. As noted by the authors themselves in [15], FTrack fails to detect packets at SNRs lower than 10dB. Since majority of the nodes operated near the noise floor (0dB SNR) in this setting, FTrack failed to detect any packets and is not shown in the corresponding Fig. 5.

Long Range Outdoor: To test the long-range capability of LoRa and the demodulators, we deployed LoRa nodes in outdoor environments at distances as far as 8 km from the receiver. Transmissions from these devices were received at SNRs as low as -15dBm. At this low SNR, even schemes that are specifically designed for low-SNR such as NScale were unable to detect any packets. While only CIC was able to

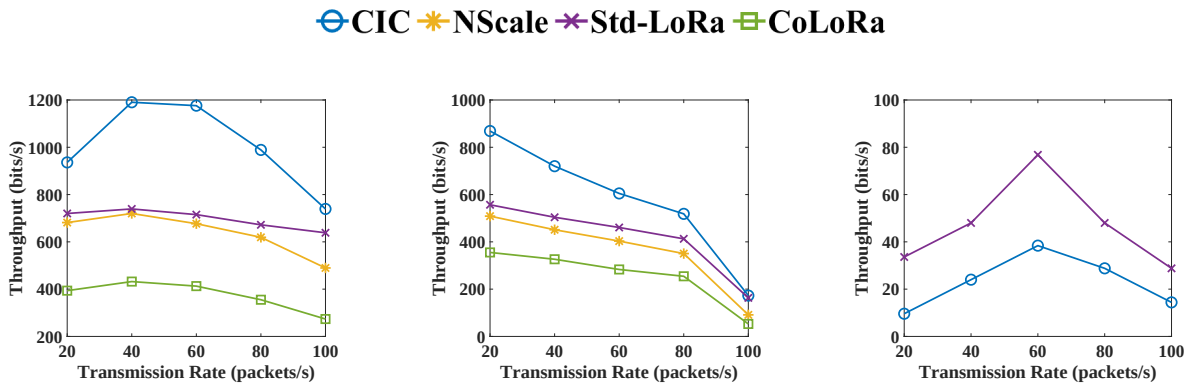


Figure 5: Throughput of a 20-node indoor NLoS network with increasing aggregate transmission rates
(a) SF8, 125kHz bandwidth (b) SF10, 250 kHz bandwidth
(c) Throughput of a 20-node outdoor network with SF10, 250 kHz bandwidth

receive packets, its network throughput is worse than that of standard LoRa as seen in Fig. 5 (c).

Summary: In summary we arrive at the following conclusions : on an average, the network throughput is higher for SF8/125kHz than that of SF10/250kHz for every demodulator. This is because, for the same message, SF10/250kHz packets have twice the duration of SF8/125kHz, which leads to a higher probability of collisions. As the aggregate rate increases, the network throughput peaks due to higher traffic. Further increments lead to an increased number of collisions, thus reducing network throughput. Extreme combination of configuration parameters: NLoS, SF10, 100 packets/s (Fig. 5 (b)), leads to comparable throughput for all demodulators, with excessive collisions nullifying any gains from the elaborate demodulation techniques as compared to the simplistic Std-LoRa. CIC demonstrates significant throughput gains over Std-LoRa in both high (>10dB) and low SNR (around 0dB) settings, followed by FTrack (in high SNR scenario) and NScale. CoLoRa despite performing better on the metrics of SER and BER, falls short of matching Std-LoRa's throughput. FTrack fails to demodulate any packets in the low SNR settings. Most techniques perform poorly at extremely low SNRs (around -15dB) as their preamble detection fails and is not as robust as Std-LoRa. Based on our experiments we believe that further study and *novel techniques for packet detection and collision resolution, especially in low-SNR regimes is needed.*

5.2 Impact of Signal to Noise Ratio (SNR) on Network Throughput

We note from the network throughput in the long-range outdoor settings above that none of the existing techniques perform better than Std-LoRa at extremely low SNR scenarios. To study the impact of SNR on the throughput performance of each demodulator, we design a new controlled experiment. In the Indoor setup, we deploy 20 LoRa nodes and accurately control their transmit power and physical placement such that all the nodes have comparable SNR that falls under one of the following categories. We repeat the experiment such that all the nodes are in High, Medium, Low, and Extremely-low SNR categories, as defined below :

- **High** SNR : >10 dB
- **Medium** SNR : 5 to 10 dB
- **Low** SNR : -5 to 5 dB
- **Extremely Low** SNR : <-5 dB

Each transmitter was configured at SF10, BW 250kHz; we collected upto 3000 colliding packets for each combination of SNR and transmission rate. Fig. 6 shows the throughput of the network as a function of decreasing SNR regime defined above, repeated for aggregate transmission rates of 20, 60 and 100 packets/s respectively. Due to the controlled setting needed for this experiment, it was not performed in the outdoor setting.

Summary: The results corroborate the earlier observations in indoor and outdoor experiments. CIC, NScale, and FTrack outperform Std-LoRa in high SNR, low traffic scenarios (Fig. 6 (a)) because of lesser collisions. However, as the SNR drops below noise floor, the throughput gains delivered by these demodulators decline sharply. FTrack fails to detect packets in lower SNR settings, whereas the throughput for other demodulators drop as we move towards lower SNR and higher transmission rates (Figs. 6 (b) and (c)). As stated in NScale, its performance is comparable to FTrack at high and medium SNR, and outperforms FTrack at lower SNRs. CIC improves the most over Std-LoRa in most of the scenarios, NScale also performs well in medium and low SNR regimes, not

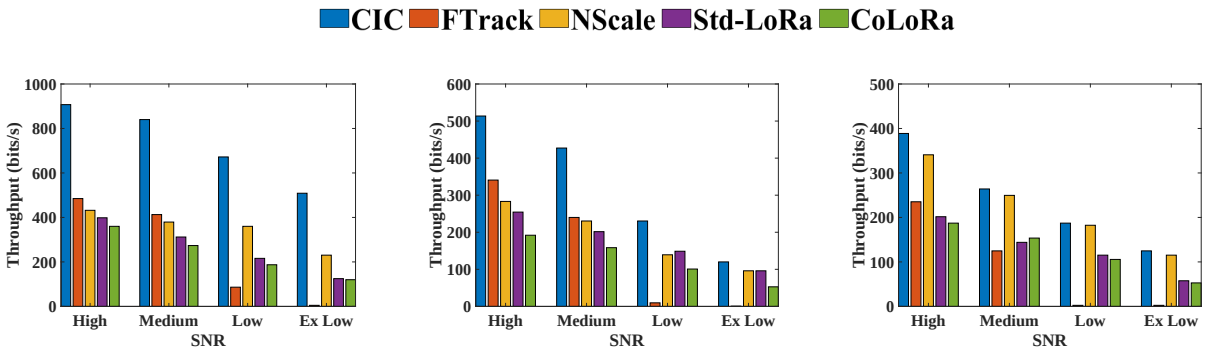


Figure 6: Throughput of a 20-node SF10 network with varying SNR and aggregate transmission rates (a) 20 packets/s (b) 60 packets/s (c) 100 packets/s

suffering excessive drop in throughput with lower SNR. At extremely-low SNR settings, all these techniques fail to demodulate most of the packets, which is further worsened at higher traffic rate. Similar to our earlier observations, the SER analysis of CoLoRa indicates that even though it has low SER, at low and extremely low SNRs, the overall throughput is worse than that of Std-LoRa. From these experiments, *we conclude that further research is required to achieve significant throughput gains over standard LoRa in extremely-low SNR and/or dense deployments, that are typical scenarios for LoRa applications.*

5.3 Impact of Interference on Network Throughput

In addition to underutilized network capacity, co-existence of multiple LoRa networks as well as across technologies was identified as a bottleneck for scalability [11, 13]. In this experiment, we evaluate the throughput performance of the five demodulators in the presence of LoRa transmissions from neighboring LoRa networks as well as non-LoRa, Gaussian Frequency Shift Keying (GFSK) narrow-band transmissions, representing other LPWANs operating in the same band. To the best of our knowledge, this is the first work to design an experiment and evaluate the impact of interference on the demodulator performance.

The primary LoRa network in this setup consists of 20 nodes, configured at SF10, BW 250kHz setting. We study the impact of the following three interfering nodes, each placed within few meters from the receiver; the interfering signal transmits at a duty cycle of 50% with an SNR of approximately 7 dB at the receiver.

i SF8, BW 125kHz LoRa node sending 93 ms packets

ii SF12, BW 500kHz LoRa node sending 290 ms packets

iii GFSK node sending 50 ms packets

Summary: Fig 7 shows the network throughput under these three types of interference. *Ambient* shows the results with no added interference. Due to the orthogonality of CSS, we expected no impact from SF8, BW 125kHz node. However, it does have a very minor impact on the performance of SF10 nodes for most demodulators. Although LoRa signals with different SFs are typically assumed to be perfectly orthogonal, this result attests to the Quasi-Orthogonality of different SFs (discussed in [24]). This quasi-orthogonality leads to residual interference even after dechirping which raises the noise floor and consequently reduces the SINR (Signal to Interference + Noise Ratio). GFSK interference has a minimal impact on the throughput performance, with a minor drop attributed to SINR. This showcases the inherent robustness of CSS to other narrow-band interference. SF12, BW 500kHz interference however notably reduces the throughput of all demodulator algorithms by almost 50%. This is primarily due to the longer packet duration of SF12 which has higher probability of interfering with complete SF10 packets whereas shorter duration of SF8 packets are less likely to interfere complete SF10 packets. *Thus, higher SFs have more impact on lower SF transmissions due to the increased probability of collision.* This is critical to notice in practical deployments where multiple LoRa networks, each operating at a different SF would co-exist. Additionally, SF-based MAC protocols have been proposed as a way to improve LoRa's scalability. *Thus, we believe that collision resolution techniques that consider the impact of interference from other LoRa networks with multiple SFs remains to be developed.*

5.4 Impact of concurrent transmissions on Packet Reception Rate

In the experiments so far, the nodes transmitted packets randomly at a predetermined rate. As the rate increased, packet collisions increased, affecting network throughput. To understand the efficacy and limitations of each algorithm in decod-

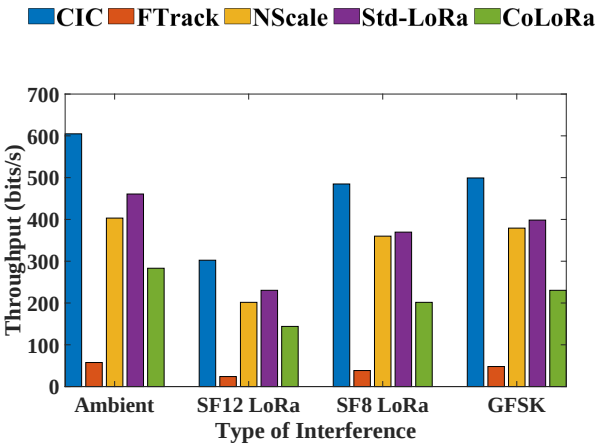


Figure 7: Throughput of a 20-node SF10 network in the presence of various interference signals

ing concurrent collisions, we performed controlled collision experiments. We synchronized all the nodes with a beacon packet: on receiving a beacon, each sender transmits a single packet with a random delay between 0 and packet duration. We define concurrency to be the number of colliding packets within one packet duration i.e.,each packet partially overlaps with every other packet.

We evaluate the number of colliding packets that can be resolved by each demodulator using the metric Packet Reception Rate (PRR). We measure PRR as we increase the number of concurrent packets from 2 to 12 in indoor LoS setup. We define PRR as the ratio of fully correct packets decoded at the receiver to the total number of packets transmitted from the transmitters. We collect upto 180 such colliding packets for each iteration of the experiment and present the metrics averaged over this dataset. Figs. 8 (a) and (b) show the PRR for SF8/125kHz and SF10/250kHz.

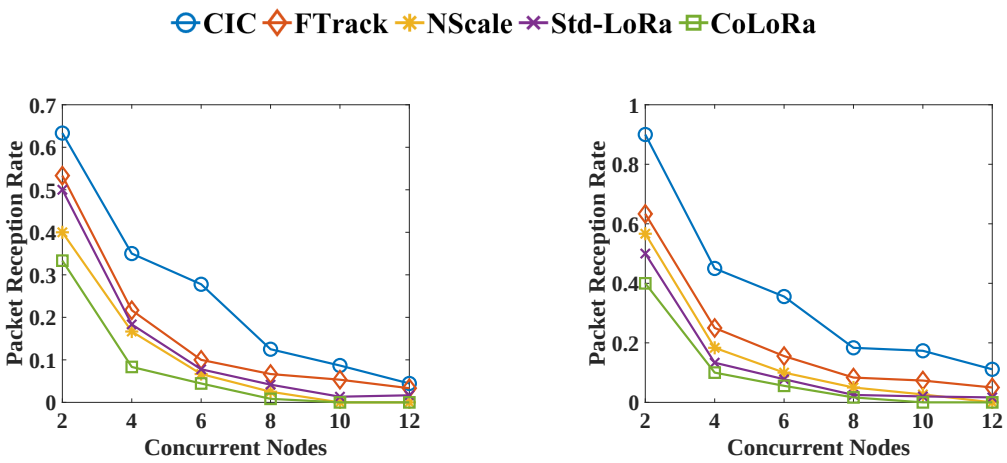
As the number of concurrent transmissions increases, PRR decreases for every demodulator. As expected, Std-LoRa has a sharp decrease in PRR since it demodulates at most one packet at a given time. We notice that irrespective of the number of nodes, Std-LoRa aligns with the strongest signal and successfully demodulates symbols corresponding to that packet. CoLoRa intentionally misaligns its demodulation window to detect more packets. Although CoLoRa detects most of the packets, it fails to demodulate when the majority of a packet overlaps with other packets. This is because of errors in finding peak ratios accurately. NScale builds on CoLoRa and improves demodulation; PRR using NScale is higher than that of CoLoRa. However, it suffers from not recognizing frequency bins of the detected peaks in presence of collisions, often leading to small offsets in the demodulated symbols. FTrack's use of time and frequency domain features to create frequency tracks and separate out collisions enables it to infer more number of colliding packets. FTrack is able to correctly output 3% to 5% more packets as compared to Std-LoRa. CIC, which improves on FTrack has the highest PRR. CIC's use of interference cancellation and spectral intersection features to demodulate enables it to demodulate most number of colliding packets. Beyond 8 concurrent collisions, PRR of all the approaches is below 0.2.

Summary: We observe a sharp decrease in PRR with increasing concurrency for all techniques because increasing concurrency reduces SINR. In this controlled collision setting, we see that SF10 transmissions result in a higher PRR, in contrast to the observations for scenarios with random collisions. This is because higher packet air-time with SF10 works in favor of demodulators here as there is more leeway in how closely in time the transmitters can collide. As the network scale of LoRa deployments increases, we expect concurrent collisions to occur with higher probability. We study the impact of the time between two colliding transmissions in more detail in the following section. *We believe that there is still room to improve in decoding collisions with more than two concurrent transmissions.*

5.5 Impact of Packet Time Offset (PTO) on Packet Reception Rate

It is evident from the concurrent transmission experiments that as the number of concurrent transmissions increases, the packet reception rate and hence network throughput decreases. Prior work [15] has also observed that the relative time (and frequency) offsets between two colliding packets plays a critical role in the throughput performance. Therefore, the impact of concurrent transmissions on throughput is a function of the time offset between the colliding packets. To understand the impact of time offsets, we design the following experiment: two LoRa nodes are connected to an Arduino microcontroller (MCU). The MCU, using a hardware interrupt, triggers the LoRa nodes such that the difference in the start of their packet transmission is configurable, thus controlling time offsets. Both the nodes transmit the same data. We repeat the experiment 20 times for each offset value and present the PRR metric averaged over the whole data.

Summary: Figs 9 (a) and (b) show the PRR as a function of increasing time offsets between two LoRa nodes. NScale and CoLoRa depend heavily on packet offsets and peak ratios to demodulate symbols and to group symbols from each transmitter together. As the packet time offset increases, their PRR performance improves as accuracy in peak estimation and peak ratio will increase. So, these show significantly better performance as the packet offset time increases, with NScale improving its PRR from 0 to 0.9 when changing collision time offset from 5% to 30% for SF8 packets. Similarly, CoLoRa shows a notable improvement in PRR over the same range, going from 0 to 0.4. Std-LoRa is unaffected by the time offsets since it always decodes the strongest signal. CIC and FTrack demodulate packets iteratively and use time and



(a) (b)

Figure 8: Reception rate of fully correct packets with increasing concurrent transmissions
(a) SF8, 125kHz bandwidth (b) SF10, 250 kHz bandwidth

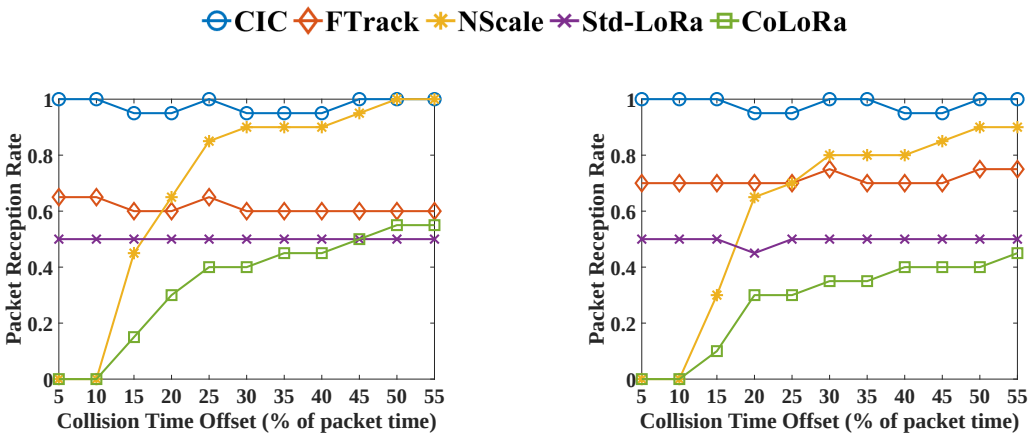
frequency information to separate collisions. The impact of time offsets on FTrack and CIC is also therefore negligible. We conclude that although most demodulators focus on improving network throughput by resolving collisions, some are better suited for collisions with a higher overlap in time than others. We observe that demodulation techniques that utilize both time and frequency resolutions are more resilient to PTO. *Therefore, more study is needed on improving time-frequency resolution when multiple concurrent nodes collide within short time offsets.*

6 Discussions and Limitations

Our results shows that packet decobability and hence network throughput differs significantly under varying network conditions. Therefore, rigorous evaluation of the existing and future LoRa demodulators over a wide variety of network conditions, as described in Section 5. We have made the datasets of the various network conditions and scenarios evaluated publicly available. Although this is an extensive set of scenarios, it certainly is not an exhaustive list. We strongly encourage the community to share new scenarios and datasets.

In *OpenLoRa*, we focused on network throughput comparisons across multiple demodulators. However, we did not compare the computational complexity of existing techniques. Since we relied on accurate recreation of existing works, optimizing each technique's implementation was not the focus. In order to evaluate the practical usability of a demodulator, computational complexity is critical. For example, Std-LoRa and CoLoRa rely on dechirping followed by FFT for packet detection and demodulation and cost the least in terms of number of computations, but also have the lowest throughput improvements, i.e., the computational complexity for both is $Nlog(N)$ where N is the number of samples per symbol and is typically the size of the FFT window as well. On the other hand, FTrack, NScale, and CIC use auto-correlation to detect the start of packets and therefore their packet detection cost these schemes N^2 computations. FTrack computes the spectrogram for the whole received buffer using a sliding window and tracks the frequencies in each window of the spectrogram; therefore, its demodulation block has $N^2log(N)$ complexity per window. CIC computes spectrogram for a fixed number of sliding windows regardless of SF as opposed to sliding over whole demodulation window and therefore has computation cost of $cNlog(N)$ where c is a constant that depends on the number of fixed sliding windows. NScale performs multiple dechirpings followed by FFT to translate timing offsets to frequency features and have a computation cost of $kNlog(N)$ where k is the number of multiple dechirpings. The computation cost for CIC and NScale is therefore lower than that of FTrack and more than standard LoRa's and CoLoRa's computation cost. Thus, further evaluations on the network throughput improvements along with their computational complexity is needed to idenitfy the practical challenges in deploying the demodulators.

In this work, we focused on novel demodulators that receive from commercially available LoRa transmitters. More recent works such as CurvingLoRa [25] and NetScatter [26] have proposed changes to the transmitter to improve resilience to packet collisions, communication range, and network throughput. Although our framework cannot be used for non-standard LoRa transmitters, the new techniques can still use our experimental scenarios to test their performance in different network conditions and compare against standard LoRa.



(a) (b)

Figure 9: Reception rate of fully correct packets with increasing collision time offset
(a) SF8, 125kHz bandwidth (b) SF10, 250 kHz bandwidth

7 Conclusions

In this work, we implement and validate four state-of-the-art collision resolution techniques for LoRa on a variety of system configurations and scenarios. We developed *OpenLoRa*, a Python framework that provides a uniform platform to evaluate existing works over the same datasets and metrics. We also design a standard LoRa decoder in order to study the end-to-end performance. This platform will allow future researchers to plug-in their demodulator and benchmark against existing works. We observe that metrics such as network throughput are more meaningful for practical deployments. We study the impact of interference and variations in SNR on the network throughput performance. We perform a wide range of experiments to emulate practical deployment settings, showcasing the strengths and challenges of existing LoRa demodulators. Our evaluations show that there are open challenges in the low-SNR, long-range regime, with more room for innovations in LoRa packet collision resolution.

8 Acknowledgements

We would like to thank our shepherd Aaron Schulman and the anonymous reviews for the valuable comments and helping us improve the paper. The authors are partially supported through the following NSF grants : CCSS-2034415, CNS-2142978, CNS-2213688, CNS-1838733, CNS-2112562, CNS-1719336, CNS-1647152, CNS-1629833, CNS-2107060, and CNS-2003129 and an award from the US Department of Commerce with award number 70NANB21H043.

References

[1] E. Asimakopoulou and N. Bessis. Buildings and crowds: Forming smart cities for more effective disaster management. In *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 229–234, 2011.

[2] María V Moreno, Miguel A Zamora, and Antonio F Skarmeta. User-centric smart buildings for energy sustainable smart cities. *Transactions on emerging telecommunications technologies*, 25(1):41–55, 2014.

[3] Achim Walter, Robert Finger, Robert Huber, and Nina Buchmann. Opinion: Smart farming is key to developing sustainable agriculture. *Proceedings of the National Academy of Sciences*, 114(24):6148–6150, 2017.

[4] Climate Smart Agriculture. https://www.worldbank.org/en/topic/climate-smart-agriculture.

[5] Fei Tao, Qinglin Qi, Ang Liu, and Andrew Kusiak. Data-driven smart manufacturing. *Journal of Manufacturing Systems*, 48:157–169, 2018.

[6] Harsha V Madhyastha and Chinedum Okwudire. Remotely controlled manufacturing: A new frontier for systems research. In *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, pages 62–67, 2020.

[7] Smart Animal Production. https://lora-alliance.org/wp-content/uploads/2020/12/THE-FARMING-OF-TOMORROW-IS-ALREADY-HERE-HOW-LoRaWAN%C2%AE-TECHNOLOGY-SUPPORTS-SMART-AGRICULTURE-PRECISE-ANIMAL-pdf.

[8] Precision Agriculture. https://cdn2.hubspot.net/hubfs/2507363/Semtech_Smart_Agriculture_White_Paper.pdf.

[9] Smart Building. https://lora-alliance.org/wp-content/uploads/2020/11/LA_WhitePaper_SmartBuildings_0520_v1.1_1.pdf.

[10] Smart Planet. https://info.semtech.com/hubfs/Semtech-UseCase-EBook-SmartPlanet_locked.pdf?hsCtaTracking=a1c4b6e2-c4a4-4fa6-942f-639ad15a6518%7C769d05d8-4178-4854-8ee6-f34d756fc141.

[11] LoRaWAN capacity trial. https://info.semtech.com/hubfs/machineQ_LoRaWAN_Capacity_Trial-2.pdf?hsLang=en-us.

[12] LoRaWAN capacity trial. https://cdn2.hubspot.net/hubfs/2507363/Semtech_Network_Capacity_White_Paper.pdf.

[13] Branden Ghena, Joshua Adkins, Longfei Shangguan, Kyle Jamieson, Philip Levis, and Prabal Dutta. Challenge: Unlicensed lpwans are not yet the path to ubiquitous connectivity. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–12, 2019.

[14] Rashad Eletreby, Diana Zhang, Swarun Kumar, and Osman Yağan. Empowering low-power wide area networks in urban settings. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 309–321. Association for Computing Machinery, 2017.

[15] Xianjin Xia, Yuanqing Zheng, and Tao Gu. Ftrack: Parallel decoding for lora transmissions. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 192–204, 2019.

[16] Shuai Tong, Jiliang Wang, and Yunhao Liu. Combating packet collisions using non-stationary signal scaling in lpwans. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 234–246, 2020.

[17] Shuai Tong, Zhenqiang Xu, and Jiliang Wang. Colora: Enabling multi-packet reception in lora. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 2303–2311. IEEE, 2020.

[18] Xiong Wang, Linghe Kong, Liang He, and Guihai Chen. mlora: A multi-packet reception protocol in lora networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2019.

[19] Muhammad Osama Shahid, Millan Philipose, Krishna Chintalapudi, Suman Banerjee, and Bhuvana Krishnaswamy. Concurrent interference cancellation : Decoding multi-packet collisions in lora. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '21. Association for Computing Machinery, 2021.

[20] Zhenqiang Xu, Pengjin Xie, and Jiliang Wang. Pyramid: Real-time lora collision decoding with peak tracking. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021.

[21] Qian Chen and Jiliang Wang. Aligntrack: Push the limit of lora collision decoding. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

[22] Adafruit Feather M0 with RFM95 LoRa Radio. https://www.adafruit.com/product/3178.

[23] RPPO. https://github.com/rpp0/gr-lora.

[24] Yujun Hou, Zujun Liu, and Dechun Sun. A novel mac protocol exploiting concurrent transmissions for massive lora connectivity. *Journal of Communications and Networks*, 22(2):108–117, 2020.

[25] Chenning Li, Xiuzhen Guo, Longfei Shangguan, Zhichao Cao, and Kyle Jamieson. CurvingLoRa to boost LoRa network throughput via concurrent transmission. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 879–895, Renton, WA, April 2022. USENIX Association.

[26] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. NetScatter: Enabling Large-Scale backscatter networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 271–284, Boston, MA, February 2019. USENIX Association.

[27] Chenning Li, Hanqing Guo, Shuai Tong, Xiao Zeng, Zhichao Cao, Mi Zhang, Qiben Yan, Li Xiao, Jiliang Wang, and Yunhao Liu. Nelora: Towards ultra-low snr lora communication with neural-enhanced demodulation. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, page 56–68, New York, NY, USA, 2021. Association for Computing Machinery.

A Appendix: LoRa Primer

LoRa Modulation. In LoRa, data is modulated using a Chirp Spread Spectrum (CSS) scheme. In CSS, symbols are chirp signals whose instantaneous frequency increases linearly with time as shown in Figure 10 (a). A base chirp starts from a frequency of $\frac{-BW}{2}$ and increases linearly to $\frac{BW}{2}$ over a symbol duration of T_s where BW is the bandwidth of transmission and T_s can be defined as $T_s = \frac{2^{SF}}{BW}$. $SF \in \{7,8,9,10,11,12\}$ defines a packet's Spreading Factor, a value that dictates the data-rate, resistance to interference and range of transmission.

Every symbol $S(t, f_{sym})$ is derived by cyclically shifting a base chirp $C(t)$, as shown in Equation 1. For example in Figure 10 (b) and (c), $S(t, f_1)$ and $S(t, f_2)$ are obtained by introducing a frequency offset of f_1 and f_2 to the base chirp. The data to be transmitted modulated these starting frequencies viz., f_1 and f_2.

$$C(t) = e^{j2\pi(0.5\frac{BW^2}{2^{SF}}t - \frac{BW}{2})t}, 0 \leq t \leq T_s \quad (1)$$

$$S(t, f_{sym}) = C(t) \cdot e^{j2\pi f_{sym}t} \quad (2)$$

LoRa Demodulation. To demodulate a symbol, a LoRa receiver aligns and multiplies a down-chirp $C^{-1}(t)$ (the complex conjugate of $C(t)$) with the received symbol $S(t, f_{sym})$ (Equation 3). Dechirping transforms the chirp signal to a sinusoid with a constant frequency equal to the start frequency f_{sym}. This frequency is located by finding the peak in the FFT of the dechirped signal. The operation of dechirping followed by FFT concentrates the symbol's energy into a single frequency, thus providing the spread spectrum gain necessary to decode symbols in sub-noise conditions.

$$C^{-1}(t) \cdot S(t, f_{sym}) = e^{j2\pi f_{sym}t} \quad (3)$$

Since dechirping requires the downchirp to align with the received symbol, a LoRa receiver determines the onset of a new packet by searching for its preamble and uses it to identify the symbol boundaries of symbols. LoRa preamble comprises of a sequence of $N = 6$ to 65535 consecutive $C(t)$ symbols, followed by two SYNC symbols $S(t, s_1), S(t, s_2)$ ($s_1 \neq 0$, $s_2 = s_1 + 8$) and 2.25 down-chirps $C^{-1}(t)$. To detect a new packet, the receiver continuously de-chirps and performs an FFT until it finds N consecutive peaks with the same frequency. The SYNC words and down-chirps then help locate the symbol boundaries. In CSS, time offsets are equivalent to frequency offsets. For example, as shown in Figure 11 time shifting a chirp symbol by τ will introduce an equivalent frequency offset in the starting frequency. Therefore, frequency offsets in LoRa can easily be compensated by identifying the equivalent time shifts during preamble detection.

Effect of collisions on demodulation. Standard LoRa is incapable of demodulating data symbols in case if multiple packets collide. Should multiple LoRa packets arrive simultaneously at the receiver, there will be multiple f_{sym} values to detect for any given symbol window, making it difficult for standard LoRa to choose one. Standard LoRa assumes that the maximum peak in the FFT always corresponds to the data value of the packet of interest. Whereas, in case of packet collision, multiple peaks from interfering packets show up in the FFT as shown in Figure 12 and the assumption of maximum peak's link to the packet of interest is not guaranteed anymore since height of interfering peaks may surpass the height of true peak.

B Appendix: LoRa Demodulators Validated

B.1 FTrack [15]

FTrack [15] relies on Short Time Fourier Transform (STFT) to obtain time and frequency features. Applying STFT to the LoRa symbols would result in frequency tracks that increases linearly with time. An appropriate STFT window size that offers good frequency resolution to identify the frequency and good time resolution to follow the progression of a chirp is challenging to determine. FTrack proposes to apply STFT on the dechirped LoRa symbol to leverage the spread spectrum gain as well as to remove the linear change in frequency with time. Dechirping the received buffer results in a sinusoid whose frequency remains constant throughout the symbol duration. This allows FTrack to choose a window size of a symbol length that yields a good frequency resolution (of upto 1 bin) if perfectly aligned with the symbol boundaries. Dechirping allows STFT to have the least possible frequency variation with time (single frequency over a symbol duration) and therefore yields the best possible frequency resolution. It yields a constant frequency track over a symbol duration, rendering it simpler to track the frequency of a packet of interest. FTrack, thus, employs dechirping followed by STFT to extract the longest frequency tracks to detect preamble as well as data symbols. Typical LoRa preambles consists of 8 base upchirps, that promise a constant frequency track for a duration of 8 symbols in the final spectrum. FTrack extracts symbol edges from preambles and uses this time information to detect the symbol boundaries of payload. FTrack builds on the observation that all the interfering packets are misaligned in time and hence their symbol boundaries are misaligned in time. FTrack detects the preambles of all colliding packets and leverages the time offset between colliding packets to differentiate transmitters. The receiver aligns itself with the boundaries of a packet of interest : once aligned, it observes the frequency tracks of current packet as well as that of the interfering tracks. The frequency track of the packet of interest will be continuous in the given window whereas all the interfering tracks will change abruptly. Therefore, after detecting all the LoRa packets in a received buffer, FTrack iteratively demodulates each packet. While demodulating a specific packet, FTrack cancels out interfering symbols by tracking the frequency continuity. At the end of this itera-

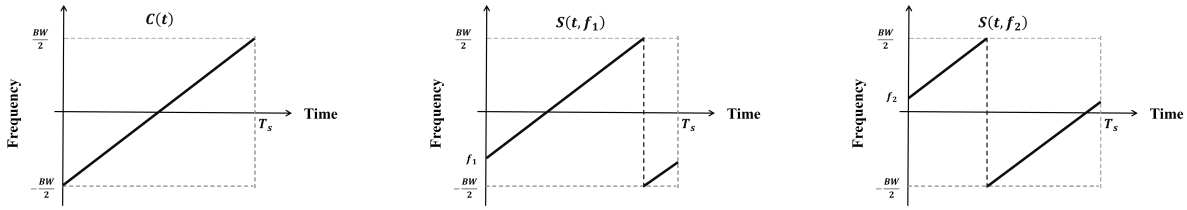


Figure 10: An illustration of Chirp spread spectrum (a) Base Upchirp (b) Data-Chirp 1 (c) Data-Chirp 2

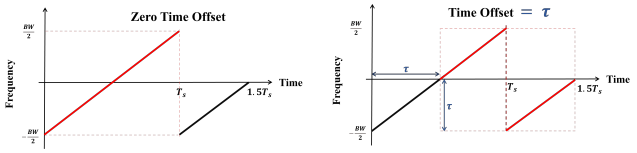


Figure 11: Time offsets in LoRa symbols translates to Frequency offsets

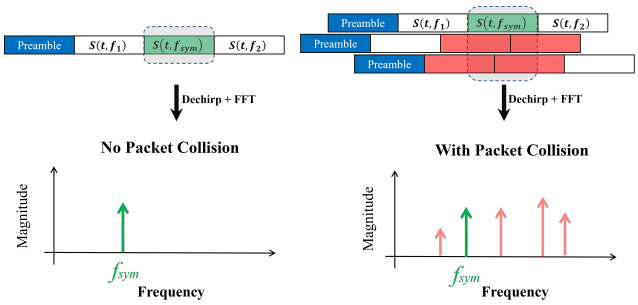


Figure 12: Symbol spectrum with and without collisions

tive process, FTrack receiver detects and demodulates data symbols from multiple transmitters that collided with each other. FTrack's performance suffers in low-SNR conditions. At SNRs below 5 dB, energy of the frequency tracks corresponding to preamble and data symbols are not high enough and hence are buried in the noise floor and is not decoded. Thus, it fails to detect and/or demodulate LoRa packets at SNRs below 5 dB.

B.2 CoLoRa [17]

CoLoRa [17] proposes a novel technique to translate time offsets to frequency features, in turn using that to resolve packet collisions at low-SNR regimes. CoLoRa starts with a misaligned window size of one symbol length. It determines the presence of interference based on the number of peaks appearing in the FFT obtained after dechirping; since multiple peaks imply packet collisions as discussed in Appendix A. Once collisions are confirmed, CoLoRa proposes an interleaved window selection strategy. It chooses a misaligned window such that no chirp is covered fully by the window i.e.,each chirp is segmented and thus falls into two consecutive windows such that the normalized FFT peak is bounded within [1/3,3] in each window. It then jumps the window over received buffer and performs dechirping followed by FFT at each point.The resulting spectrum contains peaks whose height is proportional to the segment of chirp appearing in the current window. CoLoRa observes that when a chirp is split into 2 windows, the frequency at which the peaks appear in the FFT remains the same across the 2 windows. However, the energy and hence the height of the FFT peak at the corresponding frequency in each window is proportional to the duration of the chirp segment within that window.

CoLoRa proposes *Peak Ratio*, which is defined as the ratio of peak heights of a chirp appearing in two consecutive windows; it captures time misalignment through frequency features. It proves that the peak ratio is identical for all chirps of the same packet since the in-window distribution of chirps is identical for all the symbols of the same packet. Additionally, peak ratios differ across packets since the in-window distribution of chirps is different due to misalignment of interfering packets. Since CoLoRa relies on accurate estimation of Peak ratio, it proposes an iterative peak recovery algorithm to estimate the heights of strong peaks first and cancel their contribution while estimating the low-SNR peaks. Since the chirps are segmented, wide side lobes appear around peaks that may bury low SNR peaks. CoLoRa uses k-means clustering (where k is the number of packets detected) to classify the packets of different clients. Peaks with identical peak ratios are clustered together, following the observation that the symbols of the same packet have the same peak ratio. Each cluster represents a unique packet, thus decoding multiple packets from the collided signal.

B.3 NScale [16]

While FTrack and CoLoRa focus on decoding multi-packet collisions, their performance improvements over standard LoRa demodulator is noticeable at high SNR. NScale [16] focuses on decoding packet collisions at SNRs as low as -10dB where the relative performance improvement of FTrack and CoLoRa degrades. Similar to CoLoRa, NScale translates the timing offsets to frequency features and further amplifies the time offsets by non-stationary signal scaling. NScale's strength lies in its ability to decode and resolve collisions of LoRa packets below -10 dB SNR. Instead of sliding the window as FTrack does, NScale jumps the window of size

that promises maximum frequency resolution i.e.,duration of a symbol. To retain sub-noise decodability, NScale relies on dechirping to accumulate energy at the single frequency. While jumping the window across the received buffer, NScale observe that, for a specific LoRa packet, all the symbols of interest will have same in-window distribution in consecutive windows whereas in-window distribution for interfering symbols will be different. Simply put, the location of symbol edges where the symbols transition to next symbol is same across all the windows of a given packet but across different packets, these symbol edges are different. This essentially stems from the fact that collisions are misaligned in time. Similar to CoLoRa, NScale translates symbol edge offsets to the peak heights.

NScale introduces a novel non-stationary scaled window as opposed to conventional rectangular window of FTrack and CoLoRa. Non-stationary scaling across the windows amplifies the timing misalignment of symbols of interfering packets. The linear amplitude scaled window scales the amplitude of peaks with respect to their in-window distribution and therefore amplifies the misalignment of different packets. This amplification helps estimate the time offsets for very low SNR packets. Consequently, different packets get a unique fingerprint in terms of peak heights while symbols of a specific packet share the same fingerprint. NScale detects the number of packets and their corresponding start and end indices using correlation and then performs k-means clustering to classify symbols based on their fingerprint.

B.4 Concurrent Interference Cancellation [19]

Concurrent Interference Cancellation (CIC) [19], similar to FTrack, leverages time and frequency domain analysis to decode multi-packet collisions. CIC introduces the concept of sub-windows, which are a portion of the demodulation window. It observes that, for a given packet of interest, symbols from the packet appear in all the sub-windows; symbols from interfering packets appear only in a subset of the sub-windows. Therefore, the intersection of FFT of all the sub-windows would result in the symbol of interest. CIC proposes a sub-window selection algorithm that maximizes interference cancellation.

CIC looks for the best sub-window which promises an acceptable time resolution while compromising the least on frequency resolution. It uses packet detection to determine the start of all the colliding packets inorder to select the best set of sub-windows. Unlike standard LoRa, CIC uses downchirp correlation to detect the start of a packet. With prior knowledge of symbol duration, CIC determines symbol boundaries within each of the colliding packet. The sub-windows are chosen such that it contains the most of each interfering symbol, using CIC's knowledge of the symbol boundaries of interfering packets. Spectral intersection of the FFT of the demodulating window and the optimum set of sub-windows selected results in a single FFT peak that corresponds to the symbol of interest. It iteratively demodulates the rest of the packets in the collided signal. CIC also proposes fractional frequency offset to filter out interfering peaks that were not cancelled in the spectral intersection. Additionally, it uses power-filtering to estimate received power of each packet from its preamble and discards symbols which do not qualify a certain power threshold. Finally Spectral Edge Difference, filters interfering peaks further and chooses one peak to be the final demodulated peak.

B.5 Other recent works on LoRa demodulation

In addition to the works presented above, other papers have focused on LoRa demodulator design. The modular design of our proposed framework renders it simple to integrate them. Pyramid [20] is one such work which tries to resolve LoRa collisions by tracking the change in FFT peak heights corresponding to different interfering symbols. AlignTrack [21] tracks and translates time offsets to frequency features, i.e. peak heights, similar to CoLoRa. AlignTrack chooses a window which completely overlaps the packet of interest instead of a misaligned window used by CoLoRa. AlignTrack's complete overlap gives highest peaks in FFT and therefore, has least SNR loss. NeLoRa [27] is another work which tries to push the limit of LoRa's range by using deep-neural-network. Their results show that its ability to decode packets with SNR as low as -30dB.

C Appendix: Implementation Overview

C.1 Python Implementation

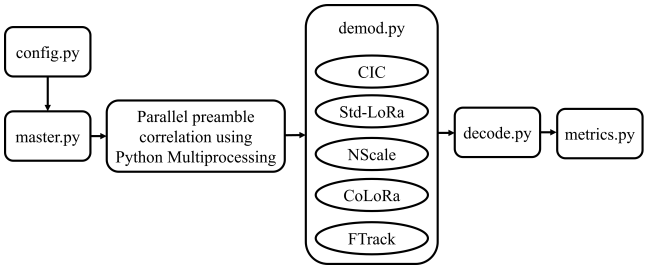


Figure 13: Flowchart of the Python implementation of the proposed framework

C.2 Code Organization

The implementation has been organized into the following major Python modules:

- *config.py*: Configures demodulation parameters such as SF, BW, sampling rate to pass to the demodulators as

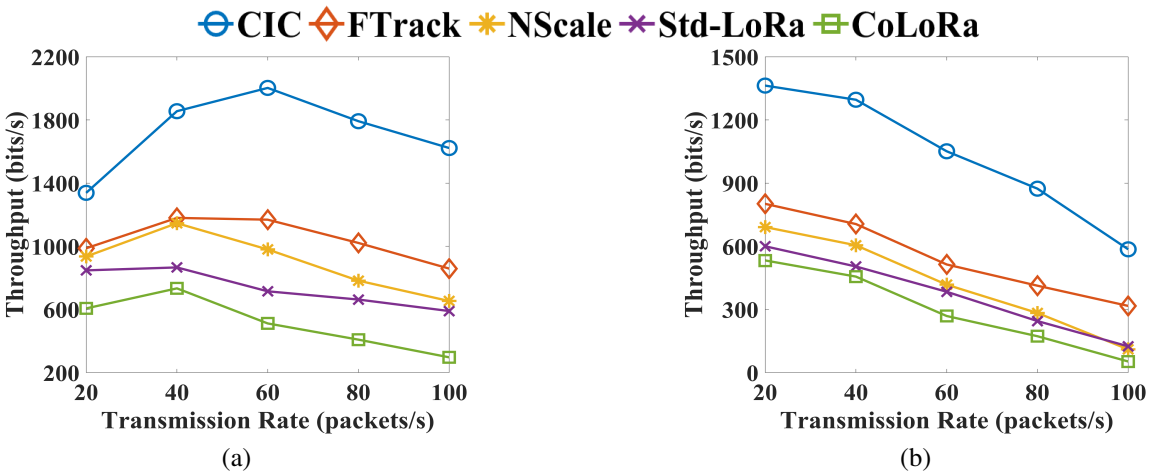


Figure 14: Throughput of a 20-node Indoor LoS network with increasing aggregate transmission rates (a) SF8, 125kHz bandwidth (b) SF10, 250 kHz bandwidth

well as the transmitted symbols and bits to compare against and evaluate the desired metrics.

- *master.py*: Performs the highest level tasks and interfaces with other modules. Reads in the input file, calls each demodulator module to get the symbols and gets calculated metrics after decoding. Also implements parallel upchirp-based preamble detection using Python multiprocessing capabilities.
- *demod.py*: Imports each demodulator implementation and interfaces with each block to pass the data and parameters in appropriate format. Returns demodulated symbols to the *master* block.
- *decode.py*: Implements the standard LoRa decoder as described in the following subsection. Returns decoded bits to the *master* block.
- *metrics.py*: Takes in the demodulated symbols, decoded bits, and the configuration information to calculate all the relevant metrics and saves then in the specified output file.

D Appendix: Additional Results

D.1 Impact of transmission rate on Network Throughput

Indoor LoS: Figs. 14 (a) and (b) depict the average network throughput as a function of aggregate transmission rate in indoor LoS setting. Network throughput is calculated as the sum of the bits of successfully decoded packets per unit time. In case of SF8 transmissions, the network throughput increases with increasing aggregate rate upto the rate of 40 packets/s for most demodulators, due to increase in traffic. However, beyond a threshold, packet collisions are too high for a demodulator to resolve, leading to a drop in throughput. CIC achieves its peak throughput at 60 packets/s because of its power filtering and down-chirp based preamble detection features. FTrack and NScale perform considerably well in this high SNR scenario giving significant gains over Std-LoRa. Even though Std-LoRa is unable to demodulate concurrent transmissions, it latches on to the strongest signal due to capture effect and often correctly demodulates the packet from the strongest transmission. This results is throughput numbers of Std-LoRa being comparable to other demodulators at higher transmission rates when they suffer from abundance of collisions. Another interesting observation is CoLoRa's throughput being slightly lesser than that of Std-LoRa despite having a lower Symbol Error Rate (SER) and Bit Error Rate (BER). Our analysis indicates that the reason is its erroneous identification of peak frequencies and calculation of peak ratios in the presence of high amount of collisions. CoLoRa detects and correctly demodulates larger number of symbols on average but the symbol errors are distributed across all concurrent transmissions. It consistently makes errors in a few of the bits in packets, which result in those packets being discounted from throughput calculation.

This result is an important observation we make in noticing the significance of end-to-end metrics such as Throughput and Packet Reception Rate over Symbol Error Rate, which could be misleading for end users. For SF10 transmissions, due to its longer air-time, the impact of collisions lead to a decreasing network throughput for all demodulators even at transmission rates as low as 1 packet/second per node.

E Appendix: Experimental Setup & Methodology

E.1 Network Experiments

All the network experiments and SNR experiments were performed using 20 transmitting nodes T_1 through T_{20}, a roll-call node R, as well as a beacon node B. R helped in setting up the parameters for each experimental setup without manually changing the setting at each location serially. Each transmitter node (T_1 through T_{20}) would reply only to their specific roll-call message from R. The roll-call messages were sent at a predefined SF and BW. Therefore, all the nodes reset to this SF and BW to listen and respond to the roll-call message. These replies were also used for calculating node-specific SNR at the USRP B200 (serving as a base station for each experiment). Using the received SNR for each node T_i, we could ensure every T_i would hear broadcasts from B. Similarly, B transmits a broadcast to inform the nodes about the settings such as SF, BW, transmission rate for the upcoming experiment. After setup, B would broadcast control messages to all 20 nodes telling each to begin transmitting messages randomly. The beacon information about transmission rates that each node T_i had to follow with a random time offset. The time offsets were generated through Poisson distribution. Each node transmitted to the base station for approximately 30 seconds. All network and SNR experiments were repeated following the aforementioned roll-call and beacon process to ensure no nodes were lost.

E.2 Interference Experiments

Interference tests utilized a similar setup to the Network Experiments, however interfering transmitters were deployed near(<10m) the receiving USRP B200. Interfering nodes included a LoRa transmitter with varying SF's and BW's as well as a GFSK transmitter. Network and interference experiment Arduino code can be located within the Experiment_Setup/Random_Network directory on the OpenLoRa Github page[5].

E.3 Concurrent Transmissions Experiment

Concurrent transmission experiments were performed using a separate beacon node B to continually synchronize up to 12 transmitting nodes T_1 through T_{12}. B would broadcast a short message instructing all nodes T_i to respond within a pre-determined time. These time-limits were determined by recording transmissions from a USRP B200 and measuring total transmit time. Offsets followed a uniform distribution within these time limits thus ensuring random transmission overlaps. The concurrent transmission experiment's Arduino code can be located within the Experiment_Setup/Random_Offset directory on the OpenLoRa Github page[5].

E.4 Packet Time Offset Experiment

To achieve reliable, and precise collisions with microsecond accuracy, we relied on interrupt-driven transmissions. Two transmitting nodes T_1 and T_2 (Adafruit Feather M0 boards), were connected to a third driving node D periodically triggering interrupts via a pin tied on T_1 and T_2. Upon receiving the interrupt, T_1 immediately transmitted its LoRa packet. Node T_2 however utilized a pre-determined delay before transmitting. Delays on T_2 were experimentally gathered ahead of time by measuring packet transmission times on a USRP B200. These delays were then calculated as some fraction of the total transmit time for a single packet, and then flashed onto T_2. Both transmitting nodes were connected to the same breadboard with similarly oriented antennas thus ensuring similar SINR at the receiver. The packet time offset experiment's Arduino code can be located within the Experiment_Setup/Precise_Offset directory on the OpenLoRa Github page[5].

[5] https://github.com/UW-CONNECT/OpenLora

VeCare: Statistical Acoustic Sensing for Automotive In-Cabin Monitoring

Yi Zhang*,†, Weiying Hou*, Zheng Yang†, Chenshu Wu*
*The University of Hong Kong
†Tsinghua University
{zhangyithss@gmail.com, wyhou@cs.hku.hk, hmilyyz@gmail.com, chenshu@cs.hku.hk}

Abstract

On average, every 10 days a child dies from in-vehicle heatstroke. The life-threatening situation calls for an automatic Child Presence Detection (CPD) solution to prevent these tragedies. In this paper, we present VeCare, the first CPD system that leverages existing in-car audio without any hardware changes. To achieve so, we explore the fundamental properties of acoustic reflection signals and develop a novel paradigm of *statistical acoustic sensing*, which allows to detect motion, track breathing, and estimate speed in a unified model. Based on this, we build an accurate and robust CPD system by introducing a set of techniques that overcome multiple challenges concerning sound interference and sensing coverage. We implement VeCare using commodity speakers and a single microphone and conduct experiments with infant simulators and adults, as well as 15 young children for the real-world in-car study. The results demonstrate that VeCare achieves an average detection rate of 98.8% with a false alarm rate of 2.1% for 15 children in various cars, boosting the coverage by over 2.3× compared to state-of-the-art and achieving whole-car detection with no blind spot.

1 Introduction

The ability of cars to sense, and save lives, inside a car remains to be improved. One life-critical feature that is widely missing is in-vehicle *Child Presence Detection* (CPD). Every year, many children have been unintentionally and unknowingly left in parked cars, or have got stuck into a car independently. As the temperature inside a car can rise rapidly[1], especially in hot months, serious injuries or heatstroke deaths could happen to children being left alone inside a car. It takes only a matter of minutes before the heat can overwhelm a child's ability to regulate his/her internal temperature and cause injuries/deaths as a child's core temperatures increase three to five times faster than an adult's [3]. On average, around 40 children dying from hot cars have been witnessed each year (about one every 10 days), leading to over 900 pediatric vehicular heatstroke (PVH) deaths on record since 1998 in the US alone [50]. Despite remarkable advances in automobiles in recent years, unfortunately, the cases of hot car deaths are only increasing, with 2018 and 2019 being the record years of 54 and 53 deaths each [50]. All of these deaths could have been prevented, if the car can detect the unattended child timely and responsively alert concerned parties or take prompt actions to keep the car cool and the child safe (as depicted in Fig. 1).

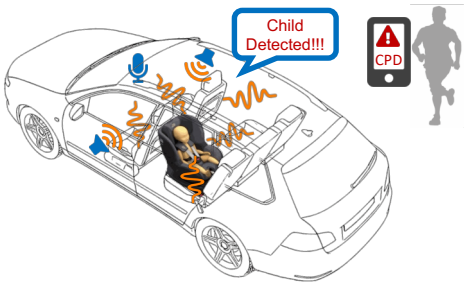


Figure 1: Application scenario of VeCare. It detects an unattended child using in-car audio and alert registered parties for immediate responses and/or activate the air conditioner to keep the child safe automatically.

The widespread and tragic problem has driven governments and the auto industry to take initiatives to make CPD a compulsion for future cars [49, 76], which fosters an expected market of $400 million by 2025 [27]. Existing solutions include early systems using special sensors such as optical/weight/pressure/ultrasonic sensors [4, 20, 21, 54], cameras [10, 13, 85], as well as recent efforts with Ultra-Wideband (UWB) or millimeter-wave (mmWave) radars [26, 28, 66], WiFi [45, 81], etc. These solutions, however, suffer from different limitations. Many works focus on adult passenger monitoring, and cannot generalize well to infants and toddlers. And the sensing coverage is mostly limited to only the seats (for special seat sensors) or a certain Field-of-View (FoV) (for

This work was done when Yi Zhang was a Research Assistant at HKU.

[1] A car can heat up by 19 degrees in just 10 minutes. Even on a mild day, the temperature in a parked car can rise to extremely dangerous and potentially fatal levels for infants and toddlers. As reported, heatstroke can occur even when outside temperatures are just 57°F [3].

cameras/UWB/mmWave radars), leading to degraded accuracy in Non-Line-Of-Sight (NLOS) scenarios and blind spots, e.g., when a child is in a rear-facing car seat, blocked by a seat, or on the car floor. More importantly, these techniques require extra hardware that is not standard offerings in today's cars to be precisely installed[2]. This not only introduces additional hardware and manufacturing costs, which are huge considering more than 80 million new cars annually, but is also backward-incompatible with most of the over one billion existing cars in the world. A truly pervasive system that requires no extra hardware and works for all cars still lacks.

In this paper, we ask the following question: *Can we build an accurate and robust in-cabin monitoring system by using only readily available in-car modules without any hardware modifications?* We present the design and implementation of such a system, named VeCare, by leveraging in-car audio systems, which are widely available in most, if not all, modern cars. As illustrated in Fig. 1, it operates by transmitting sound signals from the speakers and analyzing their reflections recorded on a microphone, which have interacted with the human body, if present. VeCare accurately and responsively detects tiny motions and extremely weak breathing of young children including newborns. It can reliably detect the presence of a child in a car, achieving whole-car detection with no blind spots. Importantly, it can be readily deployed in existing and emerging car models, offering the best ubiquity superior to the aforementioned other solutions.

Albeit acoustic sensing has been extensively studied, VeCare introduces a novel paradigm of *Statistical Acoustic Sensing* (SAS). The mainstream practice in the literature mostly focuses on *geometrical* parameters, e.g., Time of Flight (ToF) and Doppler Frequency Shift (DFS), of a few multipath reflections around the range where a target presents. Differently, inspired by recent advances in WiFi sensing [79, 88, 89], we propose to analyze the *statistical* characteristics of acoustic signals by leveraging *all* multipath reflections, which can be all affected by the target and therefore can contribute to sensing if utilized properly. Towards that end, we explore unseen properties of acoustic multipath signals and accordingly develop a novel SAS model that underpins a unified pipeline for detecting motion, estimating breathing rates, and even measuring moving speeds. The proposed SAS model truly embraces all the reflections and favors complex multipath environments, while requiring only a single microphone rather than a microphone array.

Based on the SAS model, we develop a set of techniques that overcome multiple challenges in translating SAS into a practical CPD system. First, effective acoustic channel estimation is non-trivial, mainly because of ambient sound noises, limited frequency band (up to 24kHz) on commodity devices, and multiple concurrently-transmitting speakers. In VeCare, we adopt Kasami Sequence, a pseudo-noise orthogonal sequence for channel measurement, which provides resilience to environmental noises as well as orthogonality for multi-speaker sensing. Second, acoustic sensing is known to suffer from limited coverage, e.g., typically within 1-meter range [53, 71], mainly because sound reflections off the human body are considerably weak. The problem is aggravated for young children who have even weaker motion/breathing. We boost the sensing coverage by statistically leveraging all multipaths (time diversity), optimally combining multiple subcarriers (frequency diversity), and opportunistically exploiting multiple speakers (space diversity), which ultimately allows comprehensive detection in a car. Last, CPD is a time-critical mission requiring fast response (e.g., detection within 10 seconds [49]). We design an instantaneous motion/breathing detector for CPD based on a time-domain approach, which can detect child presence rapidly (motion in a few seconds and breathing with a minimum delay slightly exceeding one breathing cycle).

We prototype an end-to-end system using commodity off-the-shelf (COTS) microphones and speakers, including car speakers and microphones. We first use infant simulators and recruit adults to systematically evaluate VeCare under various conditions both in buildings and in cars. Then we conduct a real-world study with 15 children, aged 0 to 6 except for one 10-year-old, for testing in various cars. Our results show that VeCare achieves an overall detection rate of 98.8% with a false alarm rate of 2.1%, using a single microphone. It can detect motion accurately up to 5 m, and estimate breathing at a distance of 4.5 m for an adult and 1.6 m for an infant, outperforming the state-of-the-art by 2.3×. Using in-car audio without any hardware changes, VeCare holds great potential to be widely adopted for practical CPD.

Contributions: In summary, our goal is to enable a ubiquitous solution to accurate and robust in-car CPD to prevent PVH deaths. To this end, we make three key contributions to delivering the first CPD system using accessible in-car audio: (1) We introduce a novel statistical acoustic sensing model that can detect motion, track breathing, and estimate speed by leveraging all the reflections. (2) We present a pipeline of techniques to detect motion, speed, and breathing based on the SAS model accurately and robustly, with a significantly enlarged coverage. (3) We design and implement a prototype CPD system VeCare on COTS devices and conduct extensive experiments in the real world with infant simulators and young children. Not only is VeCare a promising solution to the critical application of CPD, we also believe the proposed SAS opens a new paradigm in acoustic sensing for various applications in smart homes, healthcare, and beyond.

2 Design Space

Design Scope: Among over 900 deaths reported since 1998 [50], the primary circumstances resulting in PVH deaths include a caregiver forgetting a child in a vehicle (about 55% of

[2] WiFi is becoming popular in modern cars, but still many do not have it.

the cases), someone knowingly leaving a child in the vehicle (∼20%, e.g., running a quick errand), and the child gaining access to and getting stuck in the vehicle (about 25%). For all of these circumstances, if the car can detect the child left behind and remind/alert parents and caregivers promptly, immediate actions can be taken, by caregivers or by the car, to end these entirely preventable tragedies.

CPD systems are designed for this purpose. Typically, a CPD system is expected to run, for a short period of time, after the driver turns the engine off and locks the doors. Therefore, we mainly focus on in-cabin monitoring of a parked, closed car and do not consider a driving car. The system should then detect a child presenting anywhere inside the car quickly (e.g., within 10s [49]), and take registered actions responsively, such as alerting corresponding parties (e.g., car owners, parents, caregivers) via horn alarms and/or messages, activating the air conditioner automatically if the temperature goes high, etc. Yet how a CPD system exactly reacts is not our focus in this paper. For example, questions like how a CPD system integrates into the car system and responds to child presence (which depends on auto manufacturers, car owners, parents and caregivers), and whether the system should be a built-in component or a standalone module are out of our scope.

Why Acoustics? Although presence detection is not a new topic, existing works mostly focus on adult subjects in buildings. In-car CPD is particularly challenging because it demands a very high detection rate (any miss detection can lead to a potential tragedy) and it requires such a high accuracy for extremely tiny motion/breathing from an infant, which prior methods cannot achieve. Different modalities can be used for CPD, such as WiFi, UWB, mmWave, cameras, etc. In VeCare, we choose audio modules mainly because of the best ubiquity: Audio systems have been standard components in modern cars, which are nowadays commonly equipped with two, four, or more speakers plus *one microphone*. These speakers are most commonly installed around the dashboard, the front/back doors, and/or the rear deck, while the microphone is usually installed on the dashboard, around the rear-view mirror, or behind the steering wheel. They are placed in such a way primarily for high sound quality, which also turns out to support a good whole-car coverage for sensing.

While acoustic sensing is usually vulnerable to ambient sounds, vehicles today are designed and manufactured to provide the necessary level of safety and to muffle as much road noise as possible. Therefore, in the CPD application, the impact from the noise outside a closed car is insignificant. On the other hand, embedding sensing signals on the audible frequency band may result in shrill noises that are intrusive to human ears. Previous work has nicely modulated sensing signals into white noise [71] or on only the inaudible frequency band [9] to solve the problem. In our case, VeCare can work on either the full bandwidth (e.g., up to 24kHz) or only the inaudible band (e.g., above 18kHz), depending on practical choices. While existing works like BreathJunior [71] also monitor infants' vital signs, they are not suitable for in-car CPD as they rely on a large microphone array that is unavailable in commercial cars. Overall, acoustic signals appear to be an attractive choice for ubiquitous and practical in-car CPD, yet it entails numerous challenges to build an accurate and robust system using a single microphone.

3 Statistical Acoustic Sensing

We first present a novel statistical acoustic sensing paradigm. Our model is inspired by the success of statistical approaches in WiFi sensing [78, 79, 88, 89]. To put it briefly, this line of work treats each multipath component as a scatterer and investigates the spatial-temporal statistical properties of WiFi Channel State Information (CSI), which have been shown to imply important information such as motion [89], speed [79, 88], as well as breathing [90]. These approaches show superiority in complex environments and have been commercialized as real-world products on commercial WiFi devices [24, 25, 39]. In below, we first present our new observations on multipath propagation of acoustic signals, and then show that similar statistical properties also hold for acoustics.

Acoustic Multipaths: Unlike WiFi signals, one can effectively resolve multipath signals due to the high range resolution of acoustic signals. As a result, previous works mostly only focus on reflections from the range of interest and segment others out. However, our measurements show that a target at a certain range not only alters the reflection around that range, but also distorts multipath signals arriving later to a considerable extent. As shown in Fig. 2, while a human target contributes the strongest reflection at the Channel Impulse Response (CIR) taps corresponding to her/his range (i.e., 1.25 m), the CIR taps up to 7 m after that range are also altered remarkably. In comparison, the CIR taps are mostly noises in the empty case without human presence. Our key insight is that all these multipath distortions, if aggregated properly, can contribute useful information for sensing. The problem is, how can we truly leverage these weak and noisy multipaths?

SAS Model: CSI, a.k.a Channel Frequency Response (CFR), is the frequency-domain counterpart of CIR. CSI for an acoustic multipath channel of frequency f at time t is denoted as

$$H(f,t) = \sum_{r=1}^{R} a_r(t)\exp(-j2\pi f\tau_r(t)), \quad (1)$$

where $a_r(t)$ and $\tau_r(t)$ are the complex amplitude and propagation delay of the r-th reflection path, respectively, while R denotes the total number of paths. From a rich-scattering perspective, each reflection path can be treated as a scatterer that scatters the incoming energy back to the receiver (i.e., microphone) [22, 33, 89]. Thus, we have

$$H(f,t) = \sum_{i\in R_D} H_i(f,t) + \sum_{j\in R_S} H_j(f,t) + N(f,t), \quad (2)$$

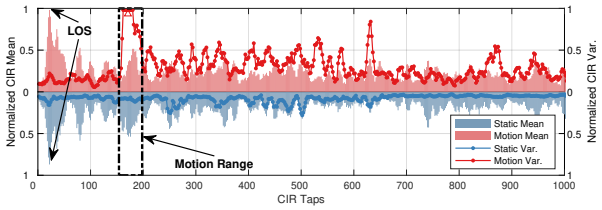


Figure 2: CIR measurements with and without human motion. The large CIR values at the motion range are truncated for the sake of visualization.

where $H_i(f,t)$ denotes the component contributed by the i-th scatterer, $N(f,t)$ is the noise term with variance σ_N^2, and R_S and R_D denote the set of static and dynamic scatterers, respectively. Assuming all scatterers are statistically independent of each other, each with the same variance $\sigma_i^2(f)$ and approximately zero means, it has been established in the context of WiFi signals [22, 79, 88], that the Autocorrelation Function (ACF) of $H(f,t)$ obeys the 0th-order Bessel function of the first kind. That is, denoting $\rho_i(f,\tau)$ as the ACF of $H_i(f,t)$ with time lag τ, we have $\rho_i(f,\tau) = J_0(kv_i\tau)$, where $J_0(x) = \frac{1}{2\pi}\int_0^{2\pi}\exp(-jx\cos(\theta))d\theta$, v_i is the moving speed of $H_i(f,t)$, and k is the wavenumber. Suppose there is one single moving target, and thus all dynamic scatterers have approximately the same speed v, $v_i \approx v, \forall i \in R_D$. This assumption is realistic because, for human subjects, the torso scatterers dominate others and have a similar speed. Then the ACF $\rho(f,\tau)$ of $H(f,t)$ can be associated with the target's moving speed v as follows [79, 88]: For $\tau \neq 0$,

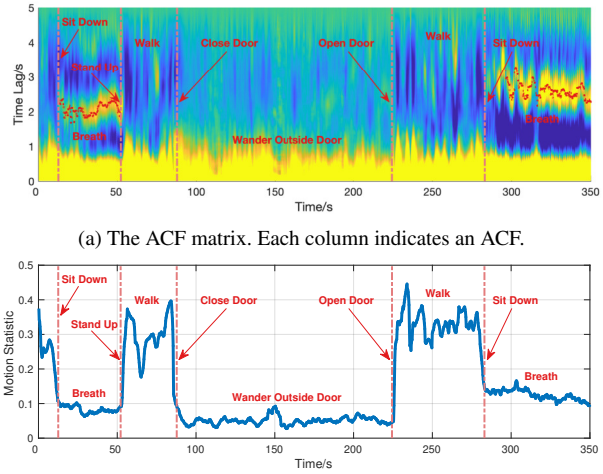
$$\begin{aligned}\rho(f,\tau) &= \frac{\sum_{i\in R_D} 2\pi\sigma_i^2(f) + \sigma_N^2(f)\delta(\tau)}{\sum_{i\in R_D} 2\pi\sigma_i^2(f) + \sigma_N^2(f)} J_0(kv\tau) \\ &\triangleq g(f)J_0(kv\tau),\end{aligned} \tag{3}$$

where $\delta(\cdot)$ is the Dirac's delta function and $g(f)$ is defined as the channel gain of $H(f,t)$. Eq. (3) bridges the ACF of the CSI with the target's moving speed. In practice, we can calculate the sample ACF, $\tilde{\rho}(f,\tau) = \rho(f,\tau) + n(f,\tau)$, from a time series of CSI measurements with a noise term $n(f,\tau)$.

While statistical approaches in WiFi sensing [79, 88–90] have demonstrated success for practical solutions and commercialized products [25, 39, 60, 67], they have not been previously explored in acoustic sensing. VECARE brings statistical sensing approaches to acoustic sensing. In the following, we perform real-world measurements to demonstrate the properties of acoustic CSI and explain how to derive motion, speed, and breathing based on the SAS model.

1) Detecting Motion: Similar to the *motion statistic* defined in [89] for WiFi CSI, we find that the defined channel gain $g(f)$ in Eq. (3) is a sensitive and robust indicator for acoustic motion detection. From Eq. (3), we have $g(f) = \lim_{\tau\to 0}\rho(f,\tau)$ since $\lim_{\tau\to 0} J_0(kv\tau) = 1$. Hence, given a sufficient CSI sampling rate F_s, we can approximate $g(f)$ as the value of the first tap of the ACF, i.e.,

$$g(f) = \tilde{\rho}(f, \tau = 1/F_s). \tag{4}$$



(a) The ACF matrix. Each column indicates an ACF.

(b) The extracted motion level

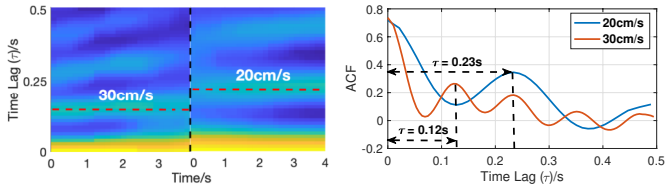
Figure 3: Human motion and breathing in a bedroom.

If any motion presents, the value of $g(f)$ is greater than zero; otherwise $g(f) \to 0$. Fig. 3 shows an example of $g(f)$ in the case of human presence and absence, respectively. As seen, there is a clear gap between the empty level and the values for motion. Additionally, $g(f)$ exhibits larger values when the motion is stronger/closer, implying that it also indicates motion strengths.

2) Tracking Breathing: ACF itself is a time-domain approach to identifying periodic signals. Therefore, we can also detect breathing signals from the ACF, which are periodic signals induced by repeated chest movements. If a breathing signal is captured by CSI, the ACF will observe a prominent peak at the time lag τ_b corresponding to the cycle time, as in Fig. 3(a). Thus, by finding time lags of these peaks over time, we can track one's breathing rates as $60/\tau_b$ BPM (breath per minute). Note that as a time-domain approach, ACF in principle is faster for breathing estimation compared to spectrum-based approaches, which usually require a much longer window to yield better frequency resolution.

3) Estimating Speed: As indicated by Eq. (3), the ACF of CSI is a function of speed v, which underpins a statistical approach entirely different from the Doppler effect for speed estimation [88]. Specifically, as shown in Fig. 4b, the shape of the ACF $\rho(f,\tau)$ resembles the Bessel function $J_0(x)$ with $x = kv\tau$, meaning that we can estimate the speed by aligning $\rho(f,\tau)$ with $J_0(x)$. Assuming x_0 is the constant value corresponding to the first peak of $J_0(x)$, then the moving speed v can be calculated as [79, 88]: $\hat{v} = \frac{x_0}{k\tau_s} = \frac{x_0\lambda(f)}{2\pi\tau_s}$, where τ_s is the time lag corresponding to the first local peak of the ACF $\rho(f,\tau)$ and $\lambda(f)$ is the wavelength of subcarrier f. Fig. 4 illustrates an example of speed estimation with the setup in Fig. 8(c), which shows that the ACF reacts to the moving speed faithfully as the above equation implies.

Remark: As seen, a peak in the ACF can either indicate a speed signal or a periodic signal. However, we notice that the peak locations for breathing (e.g., 1-5s for breathing rates



(a) ACF matrix (b) ACF

Figure 4: Speed estimation. A plate moves along a track programmed with different speeds (frequency @ 10 kHz).

60-12 BPM) are usually of magnitude longer than those for speed (e.g., <0.5s for 0.5 m/s using 10 kHz sound, and the faster the speed, the smaller the delay), a sufficient difference to determine whether to estimate breathing or speed. We do not involve speed in the current VECARE for CPD as motion and breathing will be sufficient. Yet we still present a brief description here, with an analysis of a few more issues in Appendix A.1, to show a unique approach offered by the SAS model. We keep the exploration of the full potential of SAS speed estimation for future work.

4 VECARE Design

Translating the proposed SAS model into a practical CPD system on commodity speakers and microphones still entails multiple challenges. In this section, we overcome these practical challenges and present the design of VECARE.

4.1 Acoustic Channel Estimation

The proposed SAS model leverages acoustic CSI, which demands effective channel estimation. Several unique characteristics of sound waves make it particularly challenging. First, the sound wave speed is orders of magnitude slower than that of light and EM waves, which imposes limitations on the max possible CSI sampling rate of the acoustic channel. For example, given the in-air sound speed of around 343 m/s, the propagation delay of a path of 7 meters in length will be greater than 20 ms, requiring a minimum channel measurement internal larger than 20 ms to avoid signal mixture. Second, acoustic sensing is vulnerable to environmental sound interference, especially when it is limited to a frequency band under 24 kHz on commodity devices. Ambient interference like the human voice, music, and natural sounds, can smear channel measurements for certain frequency bands. Moreover, concurrent sensing signals transmitted on multiple speakers, if used, may also interfere with each other.

In VECARE, we investigate Pseudo-Noise (PN) sequence [57] for CIR measurements. PN sequence is a set of noise-like signals and can be effectively distinguished from both a time-shifted version of itself (a.k.a, excellent auto-correlation properties) and every other signal in the set (a.k.a, excellent cross-correlation properties), which have been used in spread-spectrum communications, radar sensing, etc [52]. Among different types of PN sequences such as m-sequence [58], Golay sequence [64], GSM training sequence [86], and Zadoff-Chu (ZC) [63], we choose Kasami sequence [31] for CSI estimation because of its superior properties of orthogonality and noise tolerance. Fig. 5a shows the auto-correlation and cross-correlation of a pair of example Kasami sequences with period 2^6-1. The auto-correlation produces an impulse-like signal with minor side lobes, while the cross-correlation only produces minor values that are much smaller than the impulse of auto-correlation. Note that our approach is not limited to a particular channel estimation technique, but can work with any approach, including the widely used FMCW, that provides effective CSI.

CIR estimation with Kasami sequence: Fig. 6 shows the channel estimation process in VECARE, with two speakers as an example. We generate two orthogonal Kasami sequences s_1 and s_2 with the same length and periodically transmit them on both speakers simultaneously. The transmitted sequences undergo different time delays and attenuation before being captured by the microphone. On the receiver side, we correlate the microphone recordings with s_1 and s_2 separately to get CIR streams of the two channels, and slice them into segments with the same length as s_1 and s_2, resulting in the CIR estimates $h_1(t)$ and $h_2(t)$. We can then convert $h_1(t)$ and $h_2(t)$ into the frequency domain by performing Fourier transform and obtaining the CSI $H_1(f,t)$ and $H_2(f,t)$.

Since a correlation operation is equivalent to a conjugate multiplication in the frequency domain, the measured CSI $\tilde{H}(f)$ using Kasami sequence can be represented as

$$\begin{aligned}\tilde{H}(f) &= [S(f)\cdot H(f)+N(f)]\cdot S^*(f)\\ &= \|S(f)\|_2\cdot H(f)+N(f)\cdot S^*(f),\end{aligned} \tag{5}$$

where $H(f)$ denotes the ideal CSI, and $S(f)$ and $N(f)$ are the frequency-domain representations of Kasami sequence and sound noises respectively. $S(f)$ is a wideband signal spanning over the whole spectrum, and $S^*(f)$ is its conjugate. The term $\|S(f)\|_2\cdot H(f)$ approximates to a scaled version of $H(f)$. An example of the measured CIR is shown in Fig. 2.

We generate Pulse Coded Modulation (PCM) samples from the Kasami sequence and play them on the speaker without an extra modulation process. Several previous works have exploited more complicated signal modulation techniques to improve measurement performance including Orthogonal Frequency-Division Multiplexing (OFDM) [48] and BPSK [86], which is not necessary for VECARE.

Audibility and Interference: There are two issues with the above channel estimation process. First, the Kasami sequences composed of 1's and -1's with sharp transitions in between can be intrusive to human ears. Second, by transforming CIR, the obtained CSI spans the full spectrum, which might be polluted by the ambient sound noises, especially on the audible frequency band. To circumvent these problems, we apply a high-pass filter (HPF) on both the transmitted and received signals. The passband can be set flexibly, and

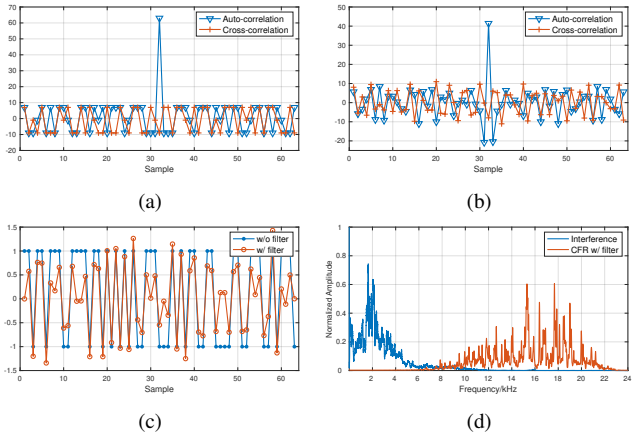


Figure 5: Kasami sequence. The auto-correlation and cross-correlation of (a) Kasami sequence and (b) Kasami sequence after applying a 10 kHz high-pass filter. (c) The Kasami sequence before and after a 10 kHz high-pass filter. (d) The spectrum of background traffic interference and CSI after the 10 kHz high-pass filter. The sound sampling rate is 48 kHz.

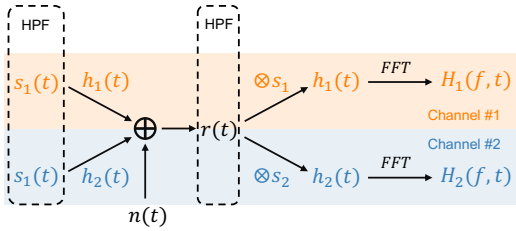


Figure 6: CSI estimation process with Kasami sequence.

VeCare can work reliably even with only the inaudible pseudo-ultrasound band, e.g., above 18kHz. Here we use an empirical passband of 10 kHz as an example for illustration and will evaluate difference choices extensively in §6.

Transmitter Filtering: When we apply the filter on the transmitter side, as shown in Fig. 5c, the binary values on the time domain signal are softened, and the output sound contains fewer intrusive bursts. The major concern is whether this filter operation breaks the auto-correlation and cross-correlation properties of Kasami sequences or not. To validate, we plot the auto-correlation and cross-correlation between the original and filtered Kasami sequence in Fig. 5b. It can be seen that, after applying the filter, the auto-correlation still observes an impulse (with a decrease in SNR) while the cross-correlation approximates the noise.

Receiver Filtering: On the receiver side, the term $N(f) \cdot S^*(f)$ in Eq. 5 is eliminated by high-pass filtering. This is because typical daily sound interference, such as traffic and human voice, mostly occurs in the frequency band below 10 kHz. Fig. 5d shows the spectrum of traffic noise and the measured CSI after applying the HPF. As seen, the noise is successfully removed. Meanwhile, we are left with fewer subcarriers for sensing because of the filtering, motivating us to maximize the sensing signals (§4.2).

Resilience to asynchronization: The speakers and microphones are connected to the same controller in our prototype and in cars as well. Due to hardware imperfections and software latency, however, they are not perfectly synchronized, which makes it difficult to measure accurate channel response. Fortunately, synchronization errors only introduce phase offsets in CSI, which does not affect VeCare because CSI is measured consecutively without blanks in between and we only use the amplitude. We will experimentally verify this in §6 and show that VeCare even works with separate speaker and microphone, while providing proof in Appendix A.2.

4.2 Sensing Signal Enhancement

Sound reflection off human bodies is considerably weak, a major reason confining the coverage of human-centric acoustic sensing [71, 74]. The problem is aggravated when the target is an infant/toddler in CPD applications. Our SAS model utilizes all multipaths for better coverage. We now present an effective technique to exploit subcarrier diversity which further boosts the sensing signals, particularly for breathing.

Subcarrier diversity is attributed to frequency selective fading, a well-known phenomenon in wireless communications. Fig. 7 demonstrates a breathing example, with the calculated ACF matrix on different subcarriers in Fig. 7a and the raw amplitude of good subcarriers (manually selected) in Fig. 7b. Some subcarriers capture dominant breathing signals, while others merely observe noises, even in such an example with strong breathing signals. We also notice that, because of the complex multipath propagation, the most sensitive subcarriers can vary over time randomly. Therefore, it is critical to dynamically find the best subsets of subcarriers and effectively combine them to maximize the signal SNR.

Like in WiFi sensing works [79, 90], we employ Maximal Ratio Combining (MRC) [6], a classical diversity combining method in wireless communications which optimizes the receiving SNR, to combine multiple subcarriers optimally. Since the noise terms on different subcarriers are statistically independent, we can maximize the signal SNR by MRC as

$$\hat{\rho}(\tau) = \sum_{f \in F} w(f)\tilde{\rho}(f,\tau), \tag{6}$$

where $\hat{\rho}(\tau)$ is the combined ACF, $w(f)$ denotes the normalized weight for combining subcarrier f (i.e., $\sum_{f\in F} w(f) = 1$), and F is the set of all subcarriers. The optimal weight $w(f)$ should be linearly proportional to the gain on each subcarrier. Following [90], we adopt the normalized $g(f)$, defined in Eq. (4), as the weight $w(f)$ in VeCare. Note that, some intuitive criteria commonly used like mean/variance of CSI amplitude cannot serve as the optimal weights for MRC, as they are not linearly proportional to the gain $g(f)$ and subcarriers with higher amplitude means/variances do not necessarily better capture the sensing signals, as shown by Fig. 7a.

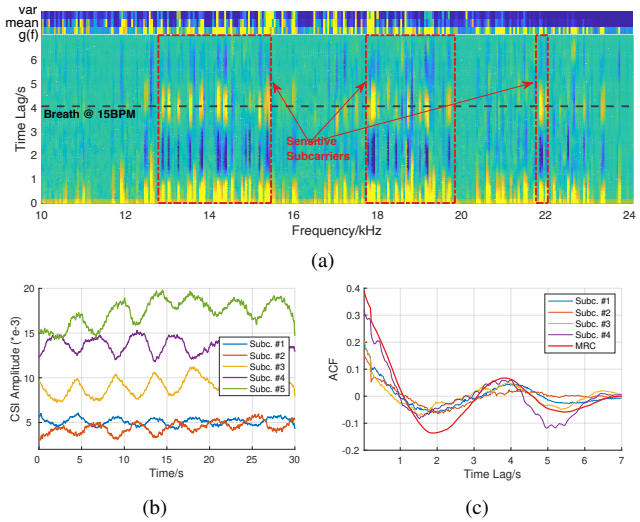


Figure 7: Breathing signals. (a) The ACF over all the subcarriers, with gain $g(f)$, mean and variance of CSI amplitude marked on the top. (b) The CSI amplitude on different subcarriers. (c) ACF on subcarriers and after MRC.

We can combine multiple subcarriers here because, by taking the ACF, the sensing signals (either breathing or speed) are synchronized across different subcarriers (Fig. 7c). It cannot be done directly on the raw amplitude due to the considerable phase offsets of breathing/speed signals on different subcarriers, as demonstrated by Fig. 7b. In case multiple speakers are available, the subcarriers on different speakers can be combined similarly, and again, asynchronization among different speakers is not an issue. By combining them, we can further boost the SNR and extend the sensing coverage.

4.3 Child Presence Detection

VECARE combines motion and breathing detection for CPD.

Motion detection: It is straightforward to detect motion. We first average the gains $g(f)$ across all subcarriers and obtain $\bar{g} = \frac{1}{|F|}\sum_{f\in F} g(f)$. Then given a preset motion threshold ε, the system detects motion at any given time t if $\bar{g}(t) > \varepsilon$; otherwise no motion presents. We use equally averaged $\bar{g}$ instead of using MRC because the averaged values across all subcarriers with equal gains will approximate zero in absence of motion, allowing us to find a generic threshold ε for different environments and cars.

Breathing Detection: To estimate the breathing rate, we first need to find whether there exists a dominant peak in the enhanced sensing signal $\hat{\rho}(\tau)$. To achieve so, we adopt similar criteria in [90] for peak finding. Basically, we first examine the peak prominence, width, and amplitude to identify potential peaks. Then we further check the peak location to sift out those beyond the typical range of human breathing rates, e.g., 10-60 BPM. We also compare the motion level $\bar{g}$ against the peak value as there will be unlikely breathing if the motion level is way larger than the peak value. Once we find the peaks corresponding to breathing, we will estimate the peak location τ_b and accordingly derive the breathing rate.

Real-time CPD: In real-time, we employ a sliding window on the continuous CSI to calculate the ACF. We employ a shorter window of CSI (e.g., 1s) for calculating the ACF for motion detection to make it more responsive while saving computation. While for breathing, a minimum window larger than a typical breathing cycle (e.g., 6s, which can be shorter for children who usually have higher breathing rates) is desired. As motion is more common and the computation is more efficient, we will only further perform breathing estimation when no motion can be detected. Note that the system can output detection decisions as fast as every CSI sample, or at a predefined lower rate, e.g., every 1 second, to save energy. Once we have the time series of motion/breathing decisions, we check them within a certain window, e.g., 5s, and child presence is claimed if there is a certain amount of motion/breathing detection, e.g., >30% of the window.

5 Implementation

Hardware: We implement VECARE using a programming audio prototype, which consists of a MiniDSP UMA-8-SP USB microphone array [44] with 7 built-in Knowles SPH1668LM4H microphones (we use only one of them) and PUI Audio AS07104PO-R speakers [5] connected to the MiniDSP board via cables. As in Fig. 8(g), we also evaluate the performance on a variety of commodity devices used in consumer electronics and cars, including JBL Stage1 621 car speaker [29] and Linhuipad car microphone [38], JBL Clip 4 speaker, Sony SRS-XB23 speaker, Razer Seiren Mini Mic, and speakers and microphones on Macbook and iPhone, etc. Again, we always use *only one single microphone* throughout our experiments, even if more are available. We connect this prototype to either a computer or Raspberry PI 3 Model B+.

Software: We implement signal generation and transmission as well as all our algorithms using MATLAB mainly for benchmark analysis. We also build an end-to-end prototype of our system running in real-time using Python3.9, which can run on embedded devices (Raspberry PI in our case).

Kasami Sequence: A longer period of Kasami Sequence allows higher SNR for channel estimation, which, however, creates an immediate conflict with sampling rates. To trade-off, we use a sequence of period $2^{10}-1$ modulated into 0.02 s, which allows a desired sampling rate of 50 Hz to use in VECARE. By default, we use 3 seconds of CSI for motion calculation and use 8 seconds for breathing rate estimation.

Handling Sharp Interference: By applying a high-pass filter, we successfully get rid of most of the daily environmental noises. However, if there are sharp and short impulse-like noises (e.g., horn honk/beep), the impacts may go above 10 kHz and cause false motion detection. We notice that these kinds of sharp noises will impose a sudden change in the CSI amplitudes, which translates into a special ACF pattern,

which linearly decreases first and then linearly increases (See Appendix A.3 for more details). Therefore, we design a detector to identify this linear decrease-then-increase pattern and skip CPD during the interfered period. By doing so, VECARE becomes immune to sharp noise like horn beep, an important feature making it more practical. Although this would reduce the effective protection time (the system is not working in presence of such noises), we argue the impact is minimal because these noises are usually short (∼1s) while VECARE detects so rapidly that it can find a period for detection.

6 Evaluation

6.1 Experimental Setup

We conduct experiments both in office environments and in cars, as Fig. 8 illustrates, which mainly consist of three parts: 1) Indoor experiments with adults to evaluate the performance in large space against various parameters. 2) Indoor and in-car experiments with infant simulators. Our evaluation involves two infant simulators. One is Laerdal SimNewB (Fig. 8(b)), a high-end model (retail price around $30,000) offered by the clinical facility in our university's medical school, which is co-created by the American Academy of Pediatrics and allows to set the breathing rate as well as move various body parts. The second one comes with a breathing motor and does not support body movements. 3) Real-world in-car experiments with children. We recruit 15 young children and perform CPD in 7 different cars including sedan and SUV.

Ground truth for adult breathing is measured by Plux piezoelectric Respiration (PZT) sensor [7]. Infant simulators have a preset fixed breathing rate. We did not record the ground truth breathing rate of children participants as it is difficult to have their cooperation. Motion and presence ground truths are manually labelled. This work does not raise any ethical issues and has been approved by our university's IRB. No sensitive data like personal identifiers were collected.

To show our performance under extreme responsiveness constraints, by default we use a 2-second window for the decision below. A higher detection rate is expected if a longer decision window is applied. Also, we use only one speaker for evaluation unless otherwise specified. Using more speakers is expected to provide larger coverage. We mainly use detection rate (DR) and false alarm rate (FAR) as the evaluation metrics for motion, breathing, and overall presence detection, while we also evaluate the mean absolute breathing rate error.

6.2 Indoor Performance

We first evaluate with comprehensive indoor experiments to validate motion detection and breathing estimation.

Motion Detection: We first evaluate the motion detection performance in a 7m×5m conference room. We set up one microphone and one speaker in the corner. The room is in an office building, with constant noise from the central fan and occasional footstep sounds when people pass by the outside corridor. An adult is asked to sit in a chair, at various distances from 1m to 5 m, and only move his one hand slowly to mimic the tiny motion of a child. We also test with the speaker facing different angles with respect to the subject. As shown in Fig. 9, VECARE achieves an average detection rate of 98.1%, which maintains 94.1% even when the user is 4∼5 m away from the speaker and microphone, while the false alarm rate is only 1.1%. The performance degrades slightly when the subject is at a distance and at an angle of 60°. Note that the motion detection rate is almost 100% when the user is within 3.5 m, a sufficient distance to cover a typical car.

Figure 8: Experimental setups

Breath Estimation: Now we evaluate the breathing estimation performance in the same environment. First, we also test with an adult subject at different distances, sitting still in a chair. As shown in Fig. 10, we achieve a mean absolute error of 0.88 BPM within the distance of 3 m, including all orientations. More importantly, VECARE can detect breathing rate at a range as far as 4.5 m, with a slight increase in breathing rate error. We also evaluate the case when there is no Line-Of-Signt (LOS) between the speaker and the subject, as well as the case when the user wears a thick down jacket. As portrayed in Fig. 11, even when a user is wearing a thick coat, VECARE can still pick up the breathing rate at distance up to 4 m. When the speaker is blocked, the maximum range of breathing estimation decreases to 2.5 m, still more than enough to cover an entire car. Note that the accuracy under occlusion is not necessarily lower than that for LOS cases since VECARE embraces all multipath reflections to significantly enhance the NLOS scenarios, which may experience richer multipath effects.

Evaluation with SimNewB: Now we carry out a feasibility study with the SimNewB newborn simulator in a clinical facility, which features tens of beds and has continuous machinery and HVAC noises. The experimental setup is illustrated in Fig. 8(a). During the tests, the laboratory technician randomly set the breathing rate of the newborn simulator. As shown in Fig. 13, VECARE can detect the newborn simulator's very weak breathing reliably, achieving an average detection rate of 87.8% with a mean error of 3.43 BPM, which decreases to 78.0% with an increased mean error of 8.6 BPM when the newborn is covered with a blanket. Note that we didn't exhaust various breathing rates due to limited access to the fa-

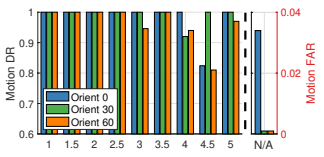


Figure 9: Indoor motion detection performance.

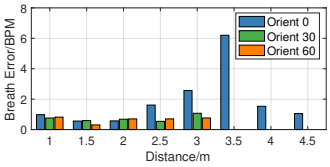


Figure 10: Indoor breath estimation performance.

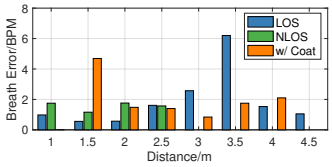


Figure 11: Breath estimation in NLOS and with coat.

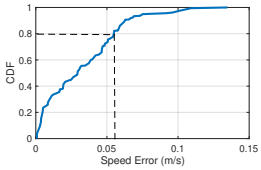


Figure 12: Speed estimation performance.

cility and SimNewB during the pandemic time, yet we believe the results already show the capability to detect a newborn's breathing at a distance. We will further study the impact of breathing rates later. In another test case, we configure the neonatal simulator to move her forearms, for which we detect the motion for 100%.

Speed Estimation: We also conduct a preliminary evaluation of our speed estimation by moving a plate back and forth at speeds from 15 cm/s to 25 cm/s along a 1.8m long straight programming track, as in Fig. 8(c). As shown in Fig. 12, VECARE achieves a considerable 80%ile accuracy of 5 cm/s. Errors mainly occur around the turning points when the plate slows to stop and starts moving again. We believe the results are encouraging and plan to further investigate statistical acoustic speed estimation for other applications in the future.

6.3 Real-World CPD Study

We conduct a real-world study with young children in different cars and parking scenarios, such as parking lots, roadside parking, garage, etc. For each child, we test different locations, with either forward-facing or rear-facing car seats as regulated. For older children who can sit/crawl independently, we also test seats without the baby car seat. All the children wear their regular winter coats. We test motion (awake) cases for every child and evaluate breathing for children who are able to get asleep (or stay very quiet) during the test. The data collection for each child lasts about 30-60 minutes. During tests, the car is parked and locked with windows closed, the typical scenario that hot-car deaths may occur. There are cars parking around and/or passing by, and parents and our experimenters talking/standing/walking around the car. There are frequent traffic noises during most of the tests, done in central downtown Hong Kong. In total, we have 15 children (aged 7 months, 12 months, 18 months, 2 (2×), 3 (4×), 4 (2×), 5 (3×), and 10 years old, respectively) tested in 7 different cars, including Lexus LS430, BMW 330, Mercedes-Benz C200, Mercedes-Benz S320, Tesla Model 3, Honda Jazz, Nissan Serena. We use one or two speakers for the real-world study, considering not all cars have four or more, and always use one single microphone. In most cases, the LOS condition is occluded, provided that the devices are installed in the front row while the kids are seated in the back. Example setups are shown in Fig. 8(d) and (e).

We mainly focus on the overall presence detection rate for this CPD test. Fig. 14 shows that VECARE achieves an average detection rate of 98.8% with an average false alarm rate of 2.1% for all age groups of children. As expected, the detection rate for infants (one 7-month-old and one 12-month-old in our experiments) is relatively lower than older kids, but is still around 90%. The high performance is consistently achieved across different cars, varying from 95% to 100% with marginal differences, as portrayed in Fig. 15. The FARs in Fig. 14 vary slightly because different kids are tested in different cars that have different enclosure materials and in-car noise levels. False negatives are most likely to occur when there is a lack of awake motion and the infant's breathing is extremely weak.

Furthermore, we analyze the performance at different in-car locations. As shown in Fig. 16, we group the results based on where the child seats, i.e., driver seat (L-F), passenger seat (L-R), two back seats (L-B and R-B), as well as the case when the child is on the back row floor (B-G). As seen, VECARE maintains a consistently high detection rate and low false alarm rate across different locations. Overall, the results demonstrate VECARE's remarkable performance in real-world scenarios, promising its potential for practical adoption.

To further understand the detection coverage in a car, we use a small toy car, as shown in Fig. 8(f), to simulate tiny motions at nine different on-seat and on-floor locations. Two speakers are installed on the left and right front doors, respectively. As depicted in Fig. 17(a), VECARE achieves a 100% detection rate for all the 9 testing locations, using either two speakers or only one single speaker on the left or the right.

Long-term Study: Besides the high accuracy, it is also critical to study false alarms, especially over a long period in diverse noisy environments like busy streets, noisy garages, etc. We first note that the above real-world experiments were conducted in noisy urban areas (including noisy garages, busy streets, parking lots next to highways, etc) in downtown Hong Kong in the presence of cars, sirens, pedestrians, etc. To further understand the performance in different environments, we carry out a relatively long-term evaluation in the busy Beijing City. We park the car, without kids inside, in a busy garage and a crowded street for about 10 hours, respectively. We report a false alarm if motion is detected for over 10% of the time for a sliding window of 2s. Our results show that VECARE observes a false alarm rate of 0.12% in the garage and 0.28% for the roadside parking case. In practice, a CPD system may not need to run for a long time, but perhaps only for a few minutes after the car is parked and locked, which will further reduce the chance of observing false alarms.

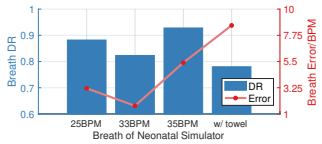


Figure 13: Evaluation with a neonatal simulator.

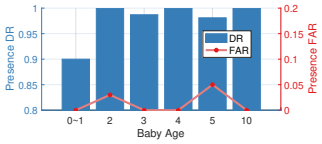


Figure 14: VECARE for children at different ages.

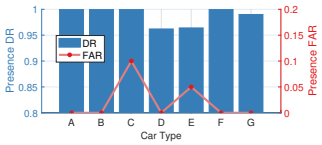


Figure 15: VECARE for children in different cars.

Figure 16: VECARE for different in-car locations.

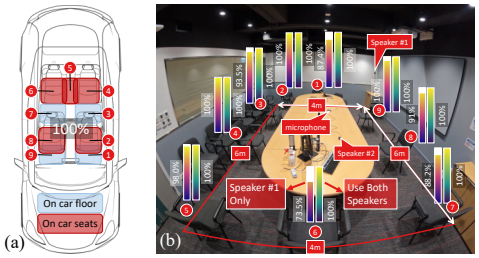


Figure 17: Motion coverage (a) in a car and (b) in a room.

System Latency: As a time-critical task, we now analyze the detection latency of VECARE. To do so, we evaluate the delay of the first decision for each test. We use a 3s window for ACF calculation for motion and an 8s window for breathing, and then use another 2s window for presence detection. Hence, the minimum delay will be 5s if motion is detected and 10s if there is no motion but breathing. With this configuration, the results show that VECARE can output the first detection within 5.7s for 81.9% of the time, 11.2s for 95.2%, and 15.2s for 98.8%. The minimum delays and thus the overall latency can be reduced by using a shorter window (e.g., 1s) for motion detection, the most common case for CPD.

6.4 Comparative Study

Baseline Comparison: We compare VECARE with the state-of-the-art approach BreathJunior [71], the closest to our work which successfully uses white noise for infant breathing monitoring. We implement BreathJunior and perform comparison experiments using an infant simulator. The results demonstrate that VECARE outperforms BreathJunior in both accuracy and coverage. As shown in Fig. 18, the maximum distance BreathJunior achieved is 70 cm (with a considerable error of 8 BPM), while VECARE goes to 1.6 meters under the same condition, which is 2.3× improvement. In addition, while BreathJunior is accurate within 0.5 m, the breathing estimation error quickly increases at a distance of 70 cm. In comparison, VECARE maintains a breathing rate error below 2 BPM at a distance of 90 cm, smaller than the error BreathJunior experiences at 60 cm.

Channel Estimation Methods: As said, VECARE can work with any channel estimation methods that output CIR. We now compare the performance of using Kasami Sequence against using different CIR estimation methods, including chirp signals (FMCW) [15], Golay Sequence [64], MLS [59], Gold Sequence [19]. As shown in Fig. 19, while all these methods produce a high detection rate above 90%, Kasami Sequence demonstrates its superior performance with the lowest breathing rate error and the highest detection rate.

Device Diversity: We now examine VECARE's performance on different devices. We are most interested in how it works on commodity car speakers and microphones. We thus evaluate it using JBL Stage1 621 car speaker and Linhuipad car microphone, both adopted in existing automobile audio systems. As shown in Fig. 20, VECARE maintains high performance and large coverage. We further test motion detection at 2m on various speaker/microphone combinations as summarized in Fig. 21, which indeed show device diversity yet good performance retains in most cases.

6.5 Benchmark Study

In this section, we evaluate the impacts of various factors and validate the robustness of VECARE. For more controllable data collection, we use the infant simulator instead of real babies for this study, and focus more on breathing estimation.

Background Interference Type: We first study the impact of background sound interference of different types, including human voices, traffic noise, rain sound, wind sound, hailstone sound, and music. To better control the experiments, we download sound files of these noises and play them through a loudspeaker around 50 dB next to the VECARE system. As shown in Fig. 22, VECARE maintains high accuracy regardless of different types of natural sound interference, with only marginal differences among them. This has also been partly verified in our real-world testing in §6.3 where we tested under real environments with all different ambient noises.

Background Interference Level: We also evaluate the performance under various background noise levels. We mainly focus on traffic noise and human voices for this test. We play sound files of noise through a loudspeaker at various powers and distances and record the actual sound level received *at the microphone*. As shown in Fig. 23, the BPM error increases with higher surrounding noises, especially over 50 dB level.

Transmitter Sound Level: The transmitting power of the speaker can affect performance. To verify this, we vary the transmitted sound from 46 dB to 53 dB and evaluate the breathing estimation error accordingly. As seen in Fig. 24, the breathing rate error quickly drops from about 7 BPM to below 2 BPM when the sound level exceeds 49 dB. Sensing sound at this level is perceived acceptable, according to our observations of the response of children participants and their parents' feedback, and users outside the car can barely hear the sound. Also, note that previous works [71] use higher sound levels (reportedly 56 dB [71] and 75 dB [70]) than

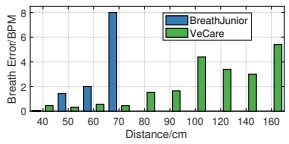


Figure 18: Performance comparison with baseline.

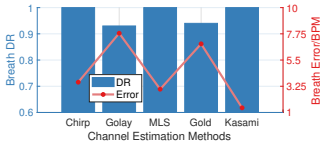
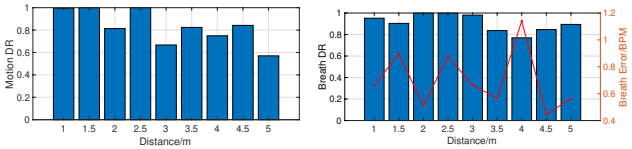


Figure 19: Comparing channel estimation methods.



(a) Motion detection. FAR=3%. (b) Breathing estimation

Figure 20: Performance using car speaker and mic.

Speaker \ Mic		Linhuipad	MiniDSP	Razer	Macbook	iPhone
JBL Stage1 Car Speaker	DR	1.00	1.00	1.00	1.00	1.00
	FAR	0.00	0.00	0.00	0.00	0.00
PUI Audio	DR	1.00	1.00	1.00	1.00	0.96
	FAR	0.00	0.00	0.00	0.00	0.18
JBL Clip4	DR	0.94	0.98	1.00	1.00	0.87
	FAR	0.07	0.00	0.00	0.00	0.03
iPhone	DR	0.92	1.00	0.87	1.00	1.00
	FAR	0.06	0.00	0.03	0.00	0.00
Sony SRS-XB23	DR	1.00	1.00	0.68	0.62	0.88
	FAR	0.00	0.00	0.16	0.41	0.28

Figure 21: Motion detection on various devices.

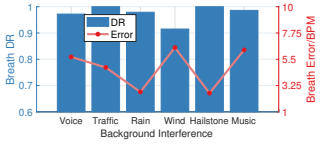


Figure 22: Impact of background interference.

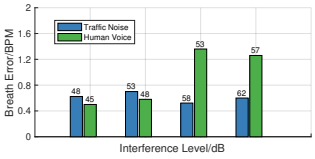


Figure 23: Impact of background interference level.

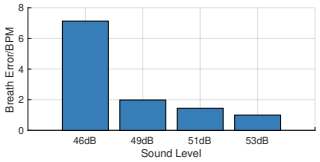


Figure 24: Impact of transmitting sound level.

VeCare. Nevertheless, a higher sound level is more favored in VeCare as a relatively high sound level could benefit CPD applications since it promises a better chance to wake up a sleeping baby for more reliable detection via awake motion. As research [16, 51] reports that a sound level higher than 75 dB will disturb the infants, we set the default sound level as 50 dB. Based on our real-world experiments with children, such a sound level appears to be tolerable to kids including infants and toddlers: We received no cases to complain about the sound intrusiveness and annoyance.

Frequency Bandwidth: We use the band above 10 kHz by default in our experiments, which may still be intrusive to human ears. We now study the performance with narrower and higher frequency bands. To do so, we adapt the passband of the high-pass filter from 10 kHz to 22 kHz with a 2 kHz step. Fig. 25 shows that VeCare retains a good performance until the passband exceeds 20 kHz. Larger bandwidths allow better performance, while VeCare still performs well using 18-24 kHz, the commonly used inaudible band in the literature. Commodity devices like Google Nest start to support acoustic frequencies up to 30 kHz [70], which we believe will become more common in the future. Such devices allow a sufficiently large and truly inaudible band across the age spectrum for non-intrusive acoustic sensing.

Impact of Temperature: As sound speed depends on temperature, we are curious how VeCare works under high temperatures. To do so, we heat up the surrounding air to about 120°F and then let it naturally cool down in a warm room of about 70°F. We keep the system running during the process and show the breathing estimation results in Fig. 26. As seen, VeCare failed to work when the devices overheat, but resumes excellent performance when they slightly cool down (after 30 seconds). We argue that a CPD system is expected to work *before* rather than *after* the car has heated up, as intervention actions are most effective right after the car is parked and locked. Therefore, we believe VeCare's CPD effectiveness will not be affected even though its performance degrades under overly high temperatures.

Multiple Speakers: Multiple speakers, if available, can further increase the sensing coverage. We present a case study with two speakers in a meeting room, where an adult sits in a chair moving one hand. As portrayed in Fig. 17(b), while the coverage with a single speaker is already good, by adding one speaker, we achieve a 100% motion detection rate through the 6m×4m area. We didn't continue with more speakers as the system already covers the entire room using two.

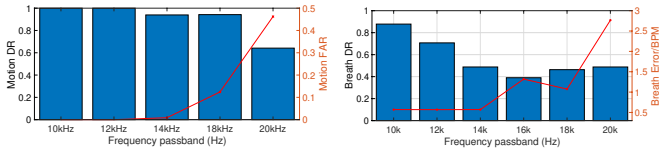
Synchronization Errors: We manually introduce large synchronization errors to show that VeCare is resilient to phase offsets. Particularly, we shift the starting point of the received signals by an amount of time ranging from 0 to 1 second with a step of 0.1 s. As shown in Fig. 27, VeCare maintains similar accuracy without being affected by the time offsets, which confirms our theoretical analysis.

Breath Intensity: Infants and toddlers usually have higher breathing rates than adults. We evaluate the performance of VeCare with respect to a range of breathing rates from 30 BPM to 60 BPM. We fix the breathing rate for each run by controlling the motor of our infant simulator. The results show insignificant differences for various breathing rates.

System Overhead: We benchmark the system overhead on a desktop (Intel i7-11700 @ 4.9GHz CPU), a MacBook Air M1, and a Raspberry Pi 3 Model B+, on which VeCare use 0.52s, 0.73s, and 3.97s respectively to process 10s of the data stream. The results show that VeCare can run in real-time on embedded devices, promising its integration into existing car control systems. The current prototype of VeCare using MiniDSP microphones introduces an extra power consumption of about 3W on Raspberry PI 3 B+, resulting in a total of 6W. The energy consumption can be optimized by improved hardware and software implementation. Additionally, the power usage is overall negligible as, again, we believe the CPD system can run only for minutes after a car is parked.

7 Discussions and Limitations

VeCare takes an important and promising step towards a ubiquitous solution to accurate and robust in-car CPD, an



(a) Motion detection (b) Breathing estimation

Figure 25: Impact of frequency band

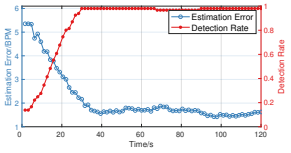


Figure 26: Impact of temperature (cooling from 120 °F)

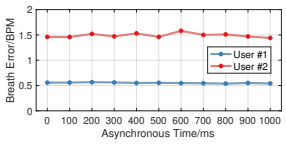


Figure 27: Impact of synchronization errors

extremely challenging task. However, there is certainly room for improvement and more to explore.

First, we cannot differentiate between an adult, a child, or a pet. Neither do we distinguish a single subject from multiple ones, as the proposed SAS currently is limited to one single user. A CPD system is expected to detect the presence of one or more children or pets, and the case of an adult being locked in a car is uncommon. While it is possible to distinguish between a child and an adult, to some extent, by examining the range of the breathing rates, we leave this task as an open challenge for the community. Second, although the proposed SAS model presents a new approach to speed estimation, the capability to estimate high speed is limited by the sampling rate of acoustic CSI. It is worth exploring how to break down this limit and enable speed tracking for normal walking speed, which will foster many applications. Third, there are more applications of the proposed SAS model in smart homes and non-contact healthcare to be explored. Towards that, one particular problem is to further improve the sound audibility, including reducing transmit sound level and shifting more to the inaudible frequency bands, e.g., 20-30 kHz used in commercial smart speakers like Amazon Echo and Google Home. Fourth, while a universal threshold ε applies to different environments and cars, we notice one-time calibration is needed for different devices due to hardware diversity (Fig. 21). Future work explores to relax it. Last, current evaluation is limited to children older than 7 months. Although we have experimented with the neonatal simulator, evaluation with real newborns is a worthwhile exploration.

VeCare can be deployed without any hardware modification: It works with legacy in-car audio systems, and we can leverage existing car control units for the computation and necessary alert functions. Alternatively, it can also be deployed as a standalone system to promote rapid adoption. Additionally, system integration with other available sensing modalities, such as seat sensors, radar, or WiFi, would promise a more reliable CPD system, but at a higher cost and lower ubiquity. These issues are, however, out of the scope of this work, and we seek to learn more about real deployment with industrial collaborations.

8 Related Work

Car Occupancy Sensing: CPD, or car occupancy detection in general, has recently gained tremendous attention with various technologies being explored. Early systems install special sensors, such as optical/weight/pressure sensors [12, 32] as well as capacitive sensors [4, 18, 54], on passenger seats/baby car seats for detection (e.g., for seat belt reminder). These systems cannot detect a left-in-car child not in a designated seat. Nor can they reliably differentiate animate targets from inanimate objects. PIR sensors [21, 55, 87] can extend the range beyond the seats, but are limited to the LOS view and sensitive to temperature changes. Camera-based systems [10, 13, 85] can be accurate given good lighting conditions[3], but cannot see through seats, in addition to being privacy-invasive and computation hungry. Radio-based systems have been recently popular. The radar industry is promoting radar systems for CPD in new car models [26, 28, 66]. UWB and mmWave radars [8, 14, 34] feature high sensing resolution, yet the coverage is limited to the FoV and the performance degrades for NLOS scenarios. Moreover, they need precise installation with wire/cable harnesses, usually on the roof of a car, to provide good coverage. With WiFi becoming prevalent for in-car connectivity, it has been exploited by the industry for CPD [45, 81]. Yet due to the innate limitations, it is challenging to detect vital signs of little infants using WiFi. Ultrasonic motion sensors have been used, e.g., in Hyundai cars [20], yet only report low detection accuracy and are being replaced [20]. Most importantly, all these solutions will incur additional dedicated hardware and/or costly installation as being non-standard offerings in cars, preventing their wide adoption, especially for old car models.

Beyond CPD, there are more works on in-car driver monitoring [68, 72, 80, 82, 83] and/or (adult) passenger detection [1, 2, 40] and even more on general human presence detection in homes especially using radio signals [30, 61, 77, 84, 89]. As the motion and vital signs of adults are of magnitude stronger than those of young children, these approaches cannot be directly applied to solve CPD. The closest to VeCare is BreathJunior [71], which nicely embeds FMCW signals into white noise for infant breathing monitoring, yet relies on a microphone array and has a limited coverage of $<$1m.

Acoustic Sensing: Acoustic sensing has been an active area in recent years. Various applications have been studied, including gesture recognition [74, 75], imaging [37, 42], localization and tracking [17, 41, 43, 56, 65, 69, 73, 86], vital sign monitoring [36, 53, 70, 71, 82], and healthcare [11, 46, 47], etc. A recent work [35] points out several practical challenges of acoustic sensing, such as audible sound leakage, affecting

[3] Infrared cameras can detect humans at night, but could still fail in dangerous cases where the car interior is already heated up.

music play and voice call, which are less concerned in our CPD application as the system is expected to only run for a short period after a car is parked. Most of the existing works explore geometrical features such as phase changes, Doppler shifts, TDoA/ToA, etc [9, 53, 74]. Many even rely on a bulky microphone array [69, 71] for phased signal processing. While these works are mostly resilient to multipaths, they do not fully leverage them. In contrast, VECARE investigates a novel statistical acoustic sensing model, which aims to leverage all reflections. Despite extensive studies of statistical WiFi sensing [23, 78, 79, 88–90] and statistical studies on acoustic communication [33, 62], none of the existing work has studied statistical acoustic sensing. We introduce statistical models to acoustic sensing, which we believe will open new directions and inspire follow-up works in the community.

9 Conclusion

We present VECARE, the first CPD system using in-car speakers and microphones. VECARE is an accurate and robust solution to the critical hot car death problem, which can be deployed on massive cars without any hardware changes. To achieve so, we introduce a novel paradigm of statistical acoustic sensing and develop a pipeline of techniques that allows motion detection, breathing estimation, and speed monitoring in a unified framework. Real-world experiments show the remarkable performance of VECARE, rendering it a promising solution in practice. The proposed SAS will inspire more exciting research in the increasingly hot acoustic sensing area.

Acknowledgments

We are grateful to all children participants in the real-world CPD study and their parents. We thank the School of Nursing at HKU for free access to the Laerdal SimNewB simulator. Thanks also go to the anonymous reviewers and to our shepherd, Nirupam Roy. This work is supported in part by NSFC under grant No. 62222216, No. 61832010 and No. 62202262, and Hong Kong RGC ECS under grant 27204522.

References

[1] Hajar Abedi, Clara Magnier, Vishvam Mazumdar, and George Shaker. Improving passenger safety in cars using novel radar signal processing. *Engineering Reports*, page e12413, 2021.

[2] Hajar Abedi, Clara Magnier, and George Shaker. Passenger monitoring using ai-powered radar. In *Proceedings of the IEEE International Symposium on Antenna Technology and Applied Electromagnetics*, pages 1–2, 2021.

[3] National Highway Traffic Safety Administration. Prevent hot car deaths. https://www.nhtsa.gov/campaign/heatstroke, 2022. Accessed: Sep 2022.

[4] Joan Albesa and Manel Gasulla. Occupancy and belt detection in removable vehicle seats via inductive power transmission. *IEEE Transactions on Vehicular Technology*, 64(8):3392–3401, 2014.

[5] PUI Audio. https://www.puiaudio.com/products/as07104po-r, 2022. Accessed: Jan 2022.

[6] John R. BarryEdward, A. LeeDavid, and G. Messerschmitt. *Digital Communication*. Springer, Boston, MA, 2004.

[7] Plux biosignalsplux piezoelectric Respiration (PZT) sensor. https://plux.info/sensors/316-respiration-pzt.html, 2022. Accessed: Jan 2022.

[8] A Caddemi and E Cardillo. Automotive anti-abandon systems: A millimeter-wave radar sensor for the detection of child presence. In *Proceedings of the IEEE International Conference on Advanced Technologies, Systems and Services in Telecommunications*, pages 94–97, 2019.

[9] Chao Cai, Rong Zheng, and Jun Luo. Ubiquitous acoustic sensing on commodity iot devices: A survey. *IEEE Communication Survey and Toturials*, page to appear, 2022.

[10] Haibin Cai, Donghee Lee, Hwang Joonkoo, Yinfeng Fang, Song Li, and Honghai Liu. Embedded vision based automotive interior intrusion detection system. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 2909–2914. IEEE, 2017.

[11] Justin Chan, Thomas Rea, Shyamnath Gollakota, and Jacob E Sunshine. Contactless cardiac arrest detection using smart devices. *NPJ digital medicine*, 2(1):1–8, 2019.

[12] Charles J Cole. System to detect the presence of an unattended child in a vehicle, January 30 2007. US Patent 7,170,401.

[13] Michel Devy, Alain Giralt, and Antonio Marin-Hernandez. Detection and classification of passenger seat occupancy using stereovision. In *Proceedings of the IEEE Intelligent Vehicles Symposium*, pages 714–719, 2000.

[14] Andreas R Diewald, Jochen Landwehr, Dimitri Tatarinov, Patrick Di Mario Cola, Claude Watgen, Catalin Mica, Mathieu Lu-Dac, Peter Larsen, Oscar Gomez, and Thierry Goniva. Rf-based child occupation detection in the vehicle interior. In *Proceedings of the IEEE International Radar Symposium*, pages 1–4, 2016.

[15] Angelo Farina. Simultaneous measurement of impulse response and distortion with a swept-sine technique. In *Audio Engineering Society Convention 108*. Audio Engineering Society, 2000.

[16] ROLAND GÄDEKE, BERNHARD DÖRING, FRIEDRICH KELLER, and ANDRES VOGEL. The noise level in a childrens hospital and the wake-up threshold in infants. *Acta Paediatrica*, 58(2):164–170, 1969.

[17] Nakul Garg, Yang Bai, and Nirupam Roy. Owlet: Enabling spatial information in ubiquitous acoustic devices. In *Proceedings of the ACM MobiSys*, pages 255–268, 2021.

[18] Boby George, Hubert Zangl, Thomas Bretterklieber, and Georg Brasseur. Seat occupancy detection based on capacitive sensing. *IEEE Transactions on Instrumentation and Measurement*, 58(5):1487–1494, 2009.

[19] Robert Gold. Optimal binary sequences for spread spectrum multiplexing (corresp.). *IEEE Transactions on information theory*, 13(4):619–621, 1967.

[20] Hyundai Motor Group. The new radar-based occupant alert system to keep your children safe. https://tech.hyundaimotorgroup.com/article/the-new-radar-based-occupant-alert-system-to-keep-your-children-safe/, 2020. Accessed: Sep 2022.

[21] NMZ Hashim, HH Basri, A Jaafar, MZAA Aziz, A Salleh, and AS Ja. Child in car alarm system using various sensors. *ARPN Journal of Engineering and Applied Sciences*, 9(9):1653–1658, 2014.

[22] David A Hill. *Electromagnetic fields in cavities: deterministic and statistical theories*, volume 35. John Wiley & Sons, 2009.

[23] Yuqian Hu, Feng Zhang, Chenshu Wu, Beibei Wang, and KJ Ray Liu. Defall: Environment-independent passive fall detection using wifi. *IEEE Internet of Things Journal*, 9(11):8515–8530, 2021.

[24] Origin Wireless Inc. Hex home: Redefining smart home security. https://myhexhome.com/, 2022. Accessed: Sep 2022.

[25] Origin Wireless Inc. Origin health - remote patient monitoring. https://www.ces.tech/Innovation-Awards/Honorees/2021/Best-Of/O/Origin-Health-Remote-Patient-Monitoring.aspx, 2022. Accessed: Sep 2022.

[26] Infineon. Infineon in-cabin monitoring. https://www.infineon.com/cms/en/tools/aurix-embedded-sw/AURIX-Applications-software/in-cabin-monitoring/, 2021. Accessed: Sep 2022.

[27] UnivDatos Market Insights. Child presence detection system market: Current analysis and forecast (2019-2025). https://univdatos.com/report/child-presence-detection-system-market-current-analysis-and-forecast-2019-2025/, 2019. Accessed: Sep 2022.

[28] Texas Instruments. Vehicle occupant detection reference design. https://www.ti.com/lit/ug/tidue95a/tidue95a.pdf, 2020. Accessed: Sep 2022.

[29] JBL. Jbl stage1 621 two way car speaker. https://www.jbl.com.my/car-speakers/STAGE1+621.html, 2022. Accessed: Sep 2022.

[30] Avinash Kalyanaraman, Elahe Soltanaghaei, and Kamin Whitehouse. Doorpler: A radar-based system for real-time, low power zone occupancy sensing. In *Proceedings of the IEEE RTAS*, pages 42–53, 2019.

[31] Tadao Kasami. Weight distribution formula for some class of cyclic codes. *Coordinated Science Laboratory Report no. R-285*, 1966.

[32] K. N. Khamil, S. Rahman, and M. Gambilok. Babycare alert system for prevention of child left in a parked vehicle. *ARPN Journal of Engineering and Applied Sciences*, 10(22):17313–17319, 2015.

[33] Heinrich Kuttruff. *Room acoustics*. CRC Press/Taylor & Francis Group, Boca Raton, sixth edition edition, 2017.

[34] Antonio Lazaro, Marc Lazaro, Ramon Villarino, and David Girbau. Seat-occupancy detection system and breathing rate monitoring based on a low-cost mm-wave radar at 60 ghz. *IEEE Access*, 9:115403–115414, 2021.

[35] Dong Li, Shirui Cao, Sunghoon Ivan Lee, and Jie Xiong. Experience: Practical problems for acoustic sensing. In *Proceedings of ACM MobiCom*, 2022.

[36] Dong Li, Jialin Liu, Sunghoon Ivan Lee, and Jie Xiong. Lasense: Pushing the limits of fine-grained activity sensing using acoustic signals. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(1):1–27, 2022.

[37] David B Lindell, Gordon Wetzstein, and Vladlen Koltun. Acoustic non-line-of-sight imaging. In *Proceedings of the IEEE/CVF CVPR*, pages 6780–6789, 2019.

[38] Linhuipad. Linhuipad car stereo microphone. https://www.amazon.co.uk/LINHUIPAD-Microphone-External-Compatible-Navigation-Black/dp/B08LR48PZK, 2022. Accessed: Sep 2022.

[39] LinkSys. Linksys aware: Introducing the first-ever mesh wifi motion sensing technology. https://www.linksys.com/us/linksys-aware/, 2022. Accessed: Sep 2022.

[40] Yongsen Ma, Yunze Zeng, and Vivek Jain. Carosense: Car occupancy sensing with the ultra-wideband keyless infrastructure. *Proceedings of the ACM IMWUT*, 4(3):1–28, 2020.

[41] Wenguang Mao, Wei Sun, Mei Wang, and Lili Qiu. Deeprange: Acoustic ranging via deep learning. *Proceedings of the ACM IWMUT*, 4(4):1–23, 2020.

[42] Wenguang Mao, Mei Wang, and Lili Qiu. Aim: Acoustic imaging on a mobile. In *Proceedings of the ACM MobiSys*, pages 468–481, 2018.

[43] Wenguang Mao, Zaiwei Zhang, Lili Qiu, Jian He, Yuchen Cui, and Sangki Yun. Indoor follow me drone. In *Proceedings of the ACM MobiSys*, pages 345–358, 2017.

[44] UMA-8-SP USB mic array. https://www.minidsp.com/products/usb-audio-interface/uma-8-sp-detail, 2022. Accessed: Jan 2022.

[45] Murata. Wi-fi sensing for child presence detection (cpd). https://solution.murata.com/en-eu/technology/child-presence-detection, 2021. Accessed: Sep 2022.

[46] Rajalakshmi Nandakumar, Shyamnath Gollakota, and Jacob E Sunshine. Opioid overdose detection using smartphones. *Science translational medicine*, 11(474):eaau8914, 2019.

[47] Rajalakshmi Nandakumar, Shyamnath Gollakota, and Nathaniel Watson. Contactless sleep apnea detection on smartphones. In *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, pages 45–57, 2015.

[48] Rajalakshmi Nandakumar, Vikram Iyer, Desney Tan, and Shyamnath Gollakota. Fingerio: Using active sonar for fine-grained finger tracking. In *Proceedings of ACM CHI*, pages 1515–1525, 2016.

[49] European New Car Assessment Programme (Euro NCAP). Test and assessment protocol - child presence detection. https://cdn.eurocap.com/media/64101/euro-ncap-cpd-test-and-assessment-protocol-v10.pdf, 2021. Accessed: Sep 2022.

[50] noheatstroke.org. Heatstroke deaths of children in vehicles. https://www.noheatstroke.org/, 2022. Accessed: Sep 2022.

[51] M Kathleen Philbin. The influence of auditory experience on the behavior of preterm newborns. *Journal of perinatology*, 20(1):S77–S87, 2000.

[52] Markku Pukkila. Channel estimation modeling. *Nokia Research Center*, 17:66, 2000.

[53] Kun Qian, Chenshu Wu, Fu Xiao, Yue Zheng, Yi Zhang, Zheng Yang, and Yunhao Liu. Acousticcardiogram: Monitoring heartbeats using acoustic signals on smart devices. In *Proceedings of the IEEE INFOCOM*, pages 1574–1582, 2018.

[54] Abhishek Ranjan and Boby George. A child-left-behind warning system based on capacitive sensing principle. In *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*, pages 702–706. IEEE, 2013.

[55] Fairuz RM Rashidi and Ikhwan H Muhamad. Vehicle's interior movement detection and notification system. *Recent advances in automatic control, modelling and simulation*, pages 139–144, 2013.

[56] Mirco Rossi, Julia Seiter, Oliver Amft, Seraina Buchmeier, and Gerhard Tröster. Roomsense: an indoor positioning system for smartphones using active sound probing. In *Proceedings of the 4th Augmented Human International Conference*, pages 89–95, 2013.

[57] D.V. Sarwate and M.B. Pursley. Crosscorrelation properties of pseudorandom and related sequences. *Proceedings of the IEEE*, 68(5):593–619, 1980.

[58] M. R. Schroeder. Integrated-impulse method measuring sound decay without using impulses. *The Journal of the Acoustical Society of America*, 66(2):497–500, 1979.

[59] Manfred R Schroeder. Integrated-impulse method measuring sound decay without using impulses. *The Journal of the Acoustical Society of America*, 66(2):497–500, 1979.

[60] Signify. Wiz enables its products with new motion detection technology. https://www.signify.com/en-us/our-company/news/press-releases/2022/20220916-surprisingly-thoughtful-lighting-for-everyone-signify-brings-to-light-motion-detection-technology-spacesense, 2022. Accessed: Sep 2022.

[61] Elahe Soltanaghaei, Rahul Anand Sharma, Zehao Wang, Adarsh Chittilappilly, Anh Luong, Eric Giler, Katie Hall, Steve Elias, and Anthony Rowe. Robust and practical wifi human sensing using on-device learning with a domain adaptive model. In *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pages 150–159, 2020.

[62] Milica Stojanovic and James Preisig. Underwater acoustic communication channels: Propagation models and statistical characterization. *IEEE communications magazine*, 47(1):84–89, 2009.

[63] Ke Sun, Ting Zhao, Wei Wang, and Lei Xie. Vskin: Sensing touch gestures on surfaces of mobile devices using acoustic signals. In *Proceedings of ACM MobiCom*, pages 591–605, 2018.

[64] Chin-Chong Tseng and C. Liu. Complementary sets of sequences. *IEEE Transactions on Information Theory*, 18(5):644–652, 1972.

[65] Yu-Chih Tung and Kang G Shin. Echotag: Accurate infrastructure-free indoor location tagging with smartphones. In *Proceedings of the ACM MobiCom*, pages 525–536, 2015.

[66] Vayyar. Vayyar child presence detection. https://vayyar.com/auto/solutions/in-cabin/cpd/, 2021. Accessed: Sep 2022.

[67] Verizon. Verizon launches new tech to monitor activity on home wifi. https://www.verizon.com/about/news/verizon-launches-new-tech-monitor-activity-home-wifi, 2023. Accessed: Feb 2023.

[68] Eric Wahlstrom, Osama Masoud, and Nikos Papanikolopoulos. Vision-based methods for driver monitoring. In *Proceedings of the IEEE International Conference on Intelligent Transportation Systems*, volume 2, pages 903–908. IEEE, 2003.

[69] Anran Wang and Shyamnath Gollakota. Millisonic: Pushing the limits of acoustic motion tracking. In *Proceedings of the ACM CHI*, pages 1–11, 2019.

[70] Anran Wang, Dan Nguyen, Arun R Sridhar, and Shyamnath Gollakota. Using smart speakers to contactlessly monitor heart rhythms. *Communications biology*, 4(1):1–12, 2021.

[71] Anran Wang, Jacob E Sunshine, and Shyamnath Gollakota. Contactless infant monitoring using white noise. In *Proceedings of the ACM MobiCom*, pages 1–16, 2019.

[72] Fengyu Wang, Xiaolu Zeng, Chenshu Wu, Beibei Wang, and KJ Ray Liu. Driver vital signs monitoring using millimeter wave radio. *IEEE Internet of Things Journal*, 2021.

[73] Mei Wang, Wei Sun, and Lili Qiu. MAVL: Multiresolution analysis of voice localization. In *Proceedings of the IEEE USENIX NSDI*, pages 845–858, 2021.

[74] Wei Wang, Alex X Liu, and Ke Sun. Device-free gesture tracking using acoustic signals. In *Proceedings of the ACM MobiCom*, pages 82–94, 2016.

[75] Yanwen Wang, Jiaxing Shen, and Yuanqing Zheng. Push the limit of acoustic gesture recognition. *IEEE Transactions on Mobile Computing*, 2020.

[76] Safe Kids Worldwide. https://www.safekids.org/, 2022. Accessed: Sep 2022.

[77] Chenshu Wu, Zheng Yang, Zimu Zhou, Xuefeng Liu, Yunhao Liu, and Jiannong Cao. Non-invasive detection of moving and stationary human with wifi. *IEEE Journal on Selected Areas in Communications*, 33(11):2329–2342, 2015.

[78] Chenshu Wu, Feng Zhang, Yusen Fan, and K. J. Ray Liu. Rf-based inertial measurement. In *ACM SIGCOMM*, 2019.

[79] Chenshu Wu, Feng Zhang, Yuqian Hu, , and K. J. Ray Liu. Gaitway: Monitoring and recognizing gait speed through the walls. In *IEEE Transactions on Mobile Computing*, pages 2186–2199, June 2021.

[80] Xiufeng Xie, Kang G Shin, Hamed Yousefi, and Suining He. Wireless csi-based head tracking in the driver seat. In *Proceedings of the ACM CoNext*, pages 112–125, 2018.

[81] Qinyi Xu, Beibei Wang, Feng Zhang, Deepika Sai Regani, Fengyu Wang, and KJ Ray Liu. Wireless ai in smart car: How smart a car can be? *IEEE Access*, 8:55091–55112, 2020.

[82] Xiangyu Xu, Jiadi Yu, Yingying Chen, Yanmin Zhu, Linghe Kong, and Minglu Li. Breathlistener: Fine-grained breathing monitoring in driving environments utilizing acoustic signals. In *Proceedings of the ACM MobiSys*, pages 54–66, 2019.

[83] Jie Yang, Simon Sidhom, Gayathri Chandrasekaran, Tam Vu, Hongbo Liu, Nicolae Cecan, Yingying Chen, Marco Gruteser, and Richard P Martin. Detecting driver phone use leveraging car speakers. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 97–108, 2011.

[84] Yuzhe Yang, Yuan Yuan, Guo Zhang, Hao Wang, Ying-Cong Chen, Yingcheng Liu, Christopher G Tarolli, Daniel Crepeau, Jan Bukartyk, Mithri R Junna, Aleksandar Videnovic, Terry D Ellis, Melissa C Lipford, Ray Dorsey, and Dina Katabi. Artificial intelligence-enabled detection and assessment of parkinson's disease using nocturnal breathing signals. *Nature medicine*, 28(10):2207–2215, 2022.

[85] Hee Jung Yoon, Ho-Kyeong Ra, Can Basaran, Sang Hyuk Son, Taejoon Park, and Jeonggil Ko. Fuzzy bin-based classification for detecting children's presence with 3d depth cameras. *ACM Transactions on Sensor Networks*, 13(3):1–28, 2017.

[86] Sangki Yun, Yi-Chao Chen, Huihuang Zheng, Lili Qiu, and Wenguang Mao. Strata: Fine-grained acoustic-based device-free tracking. In *Proceedings of the ACM MobiSys*, pages 15–28, 2017.

[87] Piero Zappi, Elisabetta Farella, and Luca Benini. Tracking motion direction and distance with pyroelectric ir sensors. *IEEE Sensors Journal*, 10(9):1486–1494, 2010.

[88] Feng Zhang, Chen Chen, Beibei Wang, and KJ Ray Liu. Wispeed: A statistical electromagnetic approach for device-free indoor speed estimation. *IEEE Internet of Things Journal*, 5(3):2163–2177, 2018.

[89] Feng Zhang, Chenshu Wu, Beibei Wang, Hung-Quoc Lai, Yi Han, and K. J. Ray Liu. Widetect: Robust motion detection with a statistical electromagnetic model. In *Proceedings of the ACM IMWUT*, Sep 2019.

[90] Feng Zhang, Chenshu Wu, Beibei Wang, Min Wu, Daniel Bugos, Hangfang Zhang, , and K. J. Ray Liu. Smars: Sleep monitoring via ambient radio signal. In *IEEE Transactions on Mobile Computing*, pages 217–231, Jan 2019.

A Appendix

A.1 Speed Estimation

Sampling Rate and Speed: A sufficient sampling rate of CSI is required to estimate speed. We now discuss the relationship. Given a sound frequency f with wavelength $\lambda(f)$, a moving speed v is expected to experience a peak at the delay of $\tau = \frac{x_0\lambda(f)}{2\pi v}$. Assume we will need at least Q samples to reliably detect a peak, which corresponds to a delay of $\tau_{\min} = Q/F_s$. Then we can derive the minimum sampling rate required to measure a speed of v by $\tau = \frac{x_0\lambda(f)}{2\pi v} > \tau_{\min} = Q/F_s$, which implies $F_s > \frac{2\pi Q v}{x_0\lambda(f)}$. In other words, the maximum speed we can support can be calculated as $v < \frac{x_0\lambda(f)F_s}{2\pi Q}$, which becomes about 0.1 m/s at $f = 20$kHz (wavelength 1.7 cm), about 0.2 m/s at $f = 10$ kHz, and about 2 m/s at $f = 1$kHz, assuming $Q = 5$ and a sampling rate of about 50 Hz (considering the sound speed of $c = 343$m/s). Using lower frequencies immediately allows to support higher speed, which however may suffer more from ambient noises. How to break down the sampling rate limitations and achieve estimation of daily speed (e.g., 0.5 m/s to 2 m/s) using pseudo-ultrasound frequencies remains worthwhile direction.

Speed MRC: For breathing signals, since the periodicity is independent of subcarrier frequency, we can directly perform

MRC across subcarriers. However, a further trick is needed to combine speed signals because, for acoustic signals from 10 kHz to 24 kHz, the difference in the wavelengths cannot be neglected (the wavelength at 10 kHz is approximately twice of that at 24 kHz). Recall $\hat{v} = \frac{x_0 \lambda(f)}{2\pi\tau_s}$. Given the same speed v, the first local peaks of the ACF on different subcarriers will appear at different delays τ_s. Hence, to combine subcarriers for speed signals, we need to first compensate the linear offsets due to different wavelengths. Specifically, we can express the ACF $\tilde{\rho}(f,\tau)$ w.r.t a unit linearly proportional to $\lambda(f)$, i.e., $\mu = \frac{\tau}{\lambda(f)}$, and then average on $\tilde{\rho}(f,\mu)$. The operation is equivalent to scaling the ACF in the time lag dimension, which can be achieved by interpolation in practice.

A.2 Synchronization

Here we show a simple but stringent proof of that synchronization errors do not affect VECARE. Denote the CIR measured under synchronization offsets as $\tilde{h}(t)$:

$$\tilde{h}(t) = circshift(h(t), \tau_{off}), \tag{7}$$

where $h(t)$ is the true CIR, τ_{off} is the timing offset caused by asynchronization, and $circshift(\cdot)$ represents circular shift. The time offsets correspond to phase shifts in the frequency domain. Thus we have the asychronized CSI:

$$\tilde{H}(f) = H(f) \cdot e^{-j2\pi f \tau_{off}}, \tag{8}$$

where $H(f)$ is the true CSI. Thus we get $|\tilde{H}(f)| = |H(f)|$, meaning that VECARE is resilient to sychronization errors.

A.3 ACF Outliers

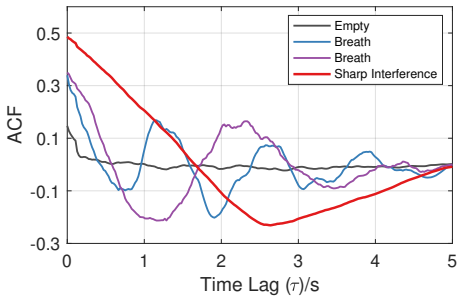


Figure 28: ACF outliers of sharp interference.

Sudden impulse-like noises will cause an abrupt status change in the CSI stream, which will smear the ACF calculation and may lead to false motion detection. We notice that ACF calculated with CSI of an abrupt status change will exhibit a special pattern, which linearly decreases and then linearly increases, as shown in Fig. 28. We also confirmed this by simulation with data stream containing a sudden change in the middle. The ACF pattern is unique and distinguishable from normal ACF in case of motion, breathing, or empty environment. Based on this observation, we design a detector to identify such abnormal ACFs and sift them out for presence detection. The idea is to examine a single valley in the ACF with linear increasing/decreasing trends on the two sides of the valley. The approach turns out to be effective and accurate.

SlimWiFi: Ultra-Low-Power IoT Radio Architecture Enabled by Asymmetric Communication

Renjie Zhao[1], Kejia Wang[2], Kai Zheng[1], Xinyu Zhang[1], Vincent Leung[2]
[1]*University of California San Diego,* [2]*Baylor University*
[1]*{r2zhao, kazheng, xyzhang}@ucsd.edu,* [2]*{kejia_wang1, vincent_leung}@baylor.edu*

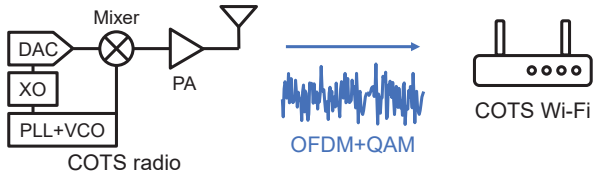
Abstract

To communicate with existing wireless infrastructures such as Wi-Fi, an Internet of Things (IoT) radio device needs to adopt a compatible PHY layer which entails sophisticated hardware and high power consumption. This paper breaks the tension for the first time through a system called SlimWiFi. A SlimWiFi radio actively transmits on-off keying (OOK) modulated signals. But through a novel *asymmetric communication* scheme, it can be directly decoded by off-the-shelf Wi-Fi devices. With this measure, SlimWiFi radically simplifies the radio architecture, evading power hungry components such as data converters and high-stability carrier generators. In addition, it can cut the transmit power by around 18 dB, while keeping a similar link budget as standard Wi-Fi. We have implemented SlimWiFi through PCB prototype and IC tape-out. Our experiments demonstrate that SlimWiFi can reach around 100 kbps goodput at up to 60 m, while reducing power consumption by around 3 orders of magnitude compared to a standard Wi-Fi transmitter.

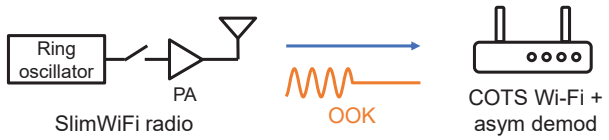
1 Introduction

The Internet of Things (IoT) is playing a key role in bridging the physical and digital worlds. IoT will act as the workhorse to fully automate human life, through a new wave of applications in environment/behavior sensing, asset tracking, ambient human-computer interaction, *etc.* As of 2021, the population of active IoT endpoints already reached 12.2 billion, and will surge towards 27 billion in 2025 [27]. Maintaining the connectivity between the IoT fabric and the existing Internet infrastructure entails non-trivial human efforts, and will ultimately be feasible only if the IoT devices can sustain themselves, *e.g.*, through RF energy harvesting. In practice, RF energy harvesting can usually reach at most tens of μW [75] for IoT devices, so any self-sustainable communication paradigm has to adhere to this limit. RFID represents one such paradigm, which is truly battery-free and communicates by merely harvesting and remodulating the RF power from an interrogator (reader). Yet to date, RFID has witnessed limited adoption in consumer applications, due to its limited communication range, relatively high cost of the reader, and limited functionality (mostly restricted to reading preprogrammed information on passive tags).

Ideally, we would prefer to reuse the existing wireless infrastructures (*e.g.*, the pervasive Wi-Fi) as gateways to connect the ultra-low-power (ULP) IoT radios to the Internet. Unfortunately, mainstream wireless communication standards cannot support battery-free operations due to their high *peak power*. For example, the commercial off-the-shelf (COTS) Wi-Fi, BLE, ZigBee, NB-IoT, and LoRa devices all require tens to hundreds of mW of peak power [34, 60, 61, 70], orders of magnitude higher than that available from RF energy harvesting. Their self-sustained operations are feasible only under an extremely low duty cycle (a few dozen bytes per day) while supported by a bulky power source (*e.g.*, a solar panel).



(a) COTS Wi-Fi symmetric communication.



(b) SlimWiFi asymmetric communication.

Figure 1: Comparison between COTS Wi-Fi and SlimWiFi.

We argue that the root cause of the high power consumption of such systems lies in the requirement of *symmetric communication*, *i.e.*, the IoT radios must adopt the same high-profile modulation/demodulation hardware as the existing wireless infrastructures. As illustrated in Fig. 1a, to be compatible with existing Wi-Fi access points (APs), an IoT radio needs to support OFDM and QAM, which entails stringent hardware requirements, such as accurate and stable carrier frequency, low phase noise, wideband and high-resolution ADC/DAC, and a high-gain high-linearity (but often low-efficiency) power amplifier, all of which translate into power hungry components. We thus pose an important question: *Is it possible to relax such requirements and make the communication hardware and modulation asymmetric?*

We explore the answers through a novel system design called *SlimWiFi*. SlimWiFi adopts a novel *asymmetric communication* scheme to realize Wi-Fi-compatible ULP radio. Specifically, the SlimWiFi ULP radio builds on a highly simplified architecture as shown in Fig. 1b, capable of only modulating/demodulating on-off keying (OOK) waveforms. But it can directly communicate with existing Wi-Fi APs that

are designed to modulate/demodulate sophisticated OFDM waveforms. Essentially, SlimWiFi shifts the PHY layer complexity to the high-power infrastructure side, and by doing so, it can improve the energy efficiency of the IoT radio by orders of magnitude. Unlike the backscatter-based systems [29, 35, 42, 79] that rely on additional helper devices to generate external carrier signals, SlimWiFi is an active, standalone radio transceiver. To materialize the design principles behind SlimWiFi, we need to address two key challenges.

(a) How to enable direct communication between asymmetric hardware, i.e., the OFDM-based Wi-Fi device and the OOK based SlimWiFi device? The uplink communication, *i.e.*, demodulating the OOK signal with an unmodified Wi-Fi OFDM device, is very challenging due to the highly incompatible waveforms and demodulation hardware. Note, however, that any demodulation process is essentially sampling and mapping analog waveforms into a binary sequence. The SlimWiFi Wi-Fi receiver thus reverses its OFDM demodulation steps, as well as the Forward-Error-Correction (FEC) decoder, and descrambler, and then reconstruct the incoming OOK symbols merely based on the payload bits reported by the Wi-Fi driver. With this measure, an ordinary Wi-Fi AP can decode the OOK signals from the ULP SlimWiFi transmitter, *without any hardware modifications*. On the other hand, the downlink modulation is straightforward, as recent work [35, 78, 79] has well-explored ways of mapping a sequence of bits into a pseudo-OOK waveform using a WiFi transmitter. To achieve MAC layer compatibility, SlimWiFi delegates the carrier sensing task to the Wi-Fi AP, which uses the CTS-to-self packets to virtually reserve the channel, and then informs the SlimWiFi node to start its transmission.

(b) How to optimize the SlimWiFi radio hardware to minimize power consumption while maintaining Wi-Fi compatibility? In commensurate with the complicated modulation, the typical hardware architecture of a COTS Wi-Fi radio necessarily consists of a power amplifier (PA) for a high transmit power, high precision and wideband digital-to-analog converter (DAC) for high-order modulation, and phase-locked loop (PLL) and voltage-controlled oscillator (VCO) for accurate carrier generation. The power consumption of these components is fundamentally governed by physical laws, and almost impossible to fall below several mW [9, 16, 55, 63]. SlimWiFi circumvents the fundamental limitation with a highly simplified radio architecture that leverages asymmetric communication. The SlimWiFi ULP radio eliminates the power hungry DAC/ADC and PLL and affords a more efficient PA owing to the lower power and linearity requirements. As for carrier generation, we adopt a free-running ring oscillator [82], which bears a low frequency stability, but suffices for SlimWiFi as its narrowband OOK signal can be asymmetrically demodulated as long as the carrier falls within the 2.4 GHz ISM band.

To verify the effectiveness of our design, we implement asymmetric communication with a COTS Wi-Fi device and a prototype SlimWiFi device. Our experiments demonstrate that the OOK based SlimWiFi signals can be decoded from the payload bits of the Wi-Fi device over a range of 60 m, with a goodput of around 100 kbps. We have also designed and taped out a SlimWiFi IC based on the aforementioned SlimWiFi radio architecture. Our measurement shows that the SlimWiFi only consumes around 90 µW of power, approximately 3 orders of magnitude lower compared with COTS WiFi radios.

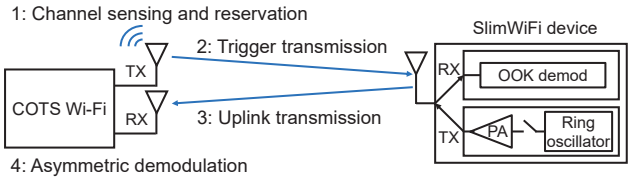


Figure 2: Workflow of a SlimWiFi uplink transmission.

To summarize, we make the following contributions through the SlimWiFi design and implementation.

- We propose SlimWiFi, a novel asymmetric communication paradigm that enables COTS Wi-Fi devices to decode OOK signals from ULP radios. The design enables such ULP radios to reuse the existing Wi-Fi as the IoT infrastructure, which can substantially reduce the deployment cost for attaining ubiquitous connectivity.
- We introduce a new SlimWiFi ULP radio architecture, which leverages the asymmetric communication to enable the first *active* Wi-Fi-compatible transmitter at a peak power of tens of µW.
- We implement the asymmetric communication system through a PCB prototype and IC tape-out. Our experiments verify the potential of SlimWiFi in supporting self-sustained IoT communication.

2 System Workflow

The SlimWiFi design mainly focuses on the IoT uplink, consisting of the SlimWiFi device and the COTS Wi-Fi radio. The former transmits OOK modulated data, through a highly simplified ULP radio architecture. The latter acts as the demodulator and gateway to connect the SlimWiFi device to the Internet. As illustrated in Fig. 2, a typical uplink transmission attempt involves the following workflow.

(1) The Wi-Fi device first runs standard carrier sensing to acquire the channel and reserves access by transmitting the CTS-to-self frame.

(2) The Wi-Fi device emulates an OOK modulated trigger frame by manipulating the Wi-Fi bit sequence. The SlimWiFi device's ULP OOK receiver decodes the information and synchronizes with the trigger frame.

(3) Following step (2) immediately, the Wi-Fi device initiates the demodulation procedure of its receiver chain, and meanwhile, the SlimWiFi device sends an OOK modulated uplink signal to the Wi-Fi device.

(4) The Wi-Fi device decodes the OOK modulated signal

by applying asymmetric demodulation.

In what follows, we introduce the SlimWiFi asymmetric communication design (Sec. 3) and the SlimWiFi ULP radio hardware (Sec. 4). Our exposition mainly focuses on the novel uplink design (steps 3 and 4). The ULP downlink design (step 2) follows the same asymmetric modulation + simplified hardware principle. It builds on recent cross-technology communication (CTC) and backscatter techniques [20, 35, 45, 67, 78], and will be discussed briefly in Sec. 4.4.

3 Asymmetric Demodulation for SlimWiFi

In this section, we first provide a quick primer on the standard Wi-Fi receiver. Then we introduce the Wi-Fi compatible asymmetric communication in SlimWiFi.

3.1 A Primer on Standard Wi-Fi Receiver

Without loss of generality, we focus on 802.11n, a standard adopted by most modern COTS Wi-Fi devices, running on a 20 MHz channel and single antenna [43]. The upper part of Fig. 3 shows the 802.11n demodulation procedure, which is hard coded into the receiver's IC. The incoming analog signals are first captured by the RF front end and converted into baseband samples. The receiver searches across the samples to identify a standard 802.11 preamble–a predefined OFDM modulated training sequence. If no valid preamble is detected, the samples will be discarded. Otherwise, the receiver will proceed to additional demodulation steps.

The samples are first sliced into *OFDM symbols*, each consisting of 16 samples of cyclic prefix (CP) and 64 samples of data. The CP is redundant samples used to overcome inter-symbol interference due to the multi-path effect. The Wi-Fi demodulator needs to remove the CP and apply a 64-point FFT to convert the 64 data samples into frequency-domain, which essentially slices the entire 20 MHz band into 64 *subcarriers*. Only 52 of the subcarriers are extracted as valid data. The remaining are either null subcarriers to mitigate adjacent channel interference or pilots for calibrating the residual offsets of the channel estimation.

Afterwards, a QAM block demaps the complex sample on each subcarrier into one or more bits, depending on the baseband modulation method, *i.e.*, BPSK, QPSK, 16-QAM, and 64-QAM. The resulting bit sequence X contains redundant bits due to forward-error-control (FEC) and needs to be decoded into a sequence Y. The ratio between the length of Y and X is called *coding rate* and can be 1/2, 2/3, 3/4, or 5/6.

The decoded bits Y need to be further reordered to recover the original transmitted bits. This so-called *descrambling* is performed by an XOR operation with a repeatedly generated 127-bit sequence whose initial state is determined by a *scrambler seed*. The PHY layer processing ends here and the output bits will be reported to the upper layer as a MAC frame. We emphasize that *the entire PHY-layer demodulation is implemented in the Wi-Fi IC and thus cannot be bypassed without hardware modification*.

On the other hand, the MAC layer control, management, and frame processing are usually implemented in software (Soft MAC) or firmware (Full MAC) [31, 47]. The MAC frames will be passed to the Wi-Fi driver and can be post-processed in software.

3.2 Overview and Challenges in Asymmetric Demodulation

The asymmetric demodulation design is grounded on a key observation: *The Wi-Fi OFDM demodulation procedure is deterministic and at least partially reversible.* An OFDM receiver essentially converts the incoming time domain samples into frequency domain through FFT, and then "quantizes" the samples through QAM demapping. Theoretically, any signals within the 20 MHz bandwidth can be *reconstructed from the OFDM receiver's bit sequence output*, by reversing the Wi-Fi demodulation procedure. *The SlimWiFi asymmetric demodulator essentially performs such reconstruction in software at the Wi-Fi receiver to recover the incoming OOK waveforms and subsequently demodulate them*, as illustrated in the bottom part of Fig. 3.

Unfortunately, the standard Wi-Fi receiver blocks, such as CP removal, QAM, and FEC, inevitably induce information loss or ambiguities. As a result, SlimWiFi must address the following key challenges.

(1) How to design the OOK signal in order to avoid the impact of information loss while enabling asymmetric demodulation? The hard-coded OFDM demodulation procedure does eliminate certain incoming samples. For example, CP removal erases part of the signal in the time domain, and data subcarrier extraction removes all information in the non-data subcarriers (*i.e.*, null and pilot subcarriers). If the removed segments contain useful data symbols from the SlimWiFi device, it would be hard to reconstruct them. We thus need to carefully design the SlimWiFi OOK waveform to avoid the impact of information loss (Sec. 3.3).

(2) How to deal with the reconstruction errors introduced by the COTS receiver? Besides the information loss from the OFDM block, the QAM and FEC blocks also cause two types of reconstruction errors: *Quantization error*, *i.e.*, the difference between the SlimWiFi signal and the closest point in Wi-Fi's QAM constellation; and *coding error*, *i.e.* the mismatch between the Wi-Fi demodulated bit sequence X and the regenerated bit sequence X' after reversing the FEC, as shown in Fig. 3. SlimWiFi addresses the reconstruction errors by (i) judiciously configuring the receiver parameters and (ii) performing additional channel coding on top of the SlimWiFi signals, as to be described in Sec. 3.4.

(3) How to integrate SlimWiFi with standard Wi-Fi protocols? To make SlimWiFi fully compatible with standard Wi-Fi, several PHY/MAC layer primitives are needed, *e.g.*, generating PHY preamble, PHY/MAC headers, and triggering the Wi-Fi receiver to start demodulation. We address these practical challenges in Sec. 3.5.

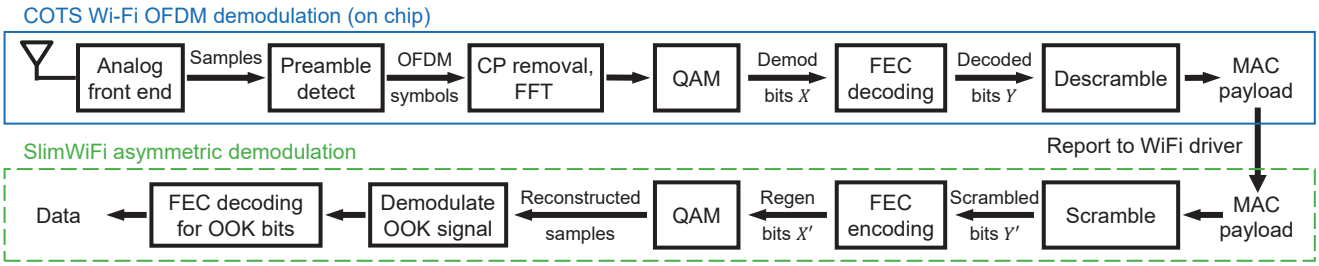
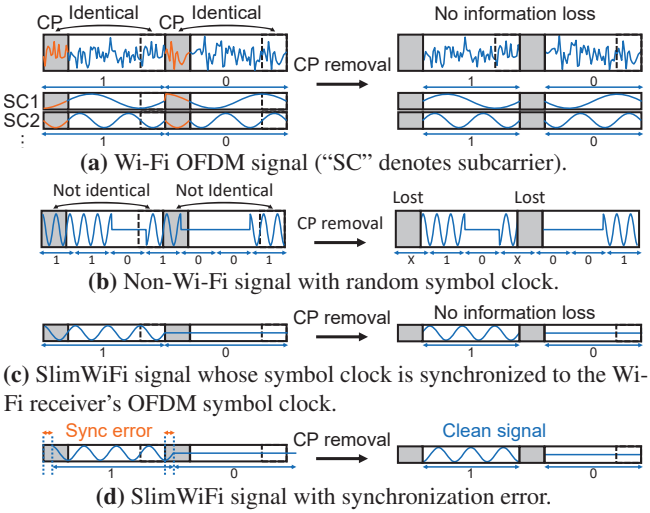


Figure 3: Receiving procedure of a SlimWiFi uplink receiver, *i.e.*, the COTS Wi-Fi device.



(a) Wi-Fi OFDM signal ("SC" denotes subcarrier).

(b) Non-Wi-Fi signal with random symbol clock.

(c) SlimWiFi signal whose symbol clock is synchronized to the Wi-Fi receiver's OFDM symbol clock.

(d) SlimWiFi signal with synchronization error.

Figure 4: Symbol clock sync to counteract CP removal.

3.3 SlimWiFi Signal Design

3.3.1 Overcoming signal erasures on the COTS Wi-Fi demodulator

In this section, we introduce the transmission waveform of the SlimWiFi device which are designed to circumvent the signal erasures on the COTS Wi-Fi demodulator.

As shown in Fig. 4a, the standard Wi-Fi waveform inside a CP is a replica of the last 0.8 µs of the OFDM symbol (4 µs in total) hosting the CP. Therefore, removing the CP does not cause any information loss for the Wi-Fi demodulator. In contrast, for a non-Wi-Fi signal with an arbitrary symbol clock (Fig. 4b), this operation may inadvertently erase 20% of the original signal which makes the demodulation unreliable. To overcome this issue, we choose to synchronize the OOK symbol clock of the SlimWiFi device with the OFDM symbol clock of the Wi-Fi receiver, *i.e.*, 250 kHz for 802.11n. Fig. 4c shows that, with such symbol-level clock synchronization, the SlimWiFi signal acts the same as the signal of one Wi-Fi subcarrier (in Fig. 4a). Therefore, the signal erasure caused by CP removal can be avoided. To realize the symbol level clock synchronization, the SlimWiFi device simply generates a 250 kHz clock and aligns its transmission time to the aforementioned trigger frame (Sec. 2). Such synchronization relies on symbol energy detection and may not be precise. However, as shown in Fig. 4d, the redundant CP part can be utilized to tolerate the synchronization errors, which we will further verify in Sec. 6.2.

Recall that 12 out of the 64 subcarriers within the 20 MHz Wi-Fi channel are null or pilot subcarriers, eventually discarded by the Wi-Fi demodulator. Therefore, to prevent information loss, the SlimWiFi device should avoid modulating its OOK waveform at the same frequencies as the non-data subcarriers. This in turn imposes more constraints on its signal bandwidth and carrier frequency, which we address below.

3.3.2 Relaxing the hardware requirements on the SlimWiFi radio device

Range, TX power, and bandwidth. The communication range of the SlimWiFi uplink can be estimated based on the classical link budget equation [85]:

$$k_b T_a B + NF + SNR_o = P_{TX} + G_{TX} + G_{RX} - 20log_{10}(4\pi d f_c/c)$$

where k_b is the Boltzmann constant, and T_a is the equivalent noise temperature in [K]. B, NF, and SNR_o denote the signal bandwidth, RX noise figure, and SNR threshold for robust decoding, respectively. P_{TX}, G_{TX}, and G_{RX} are TX power, TX, and RX antenna gain, respectively. d is the operating range, f_c is the carrier frequency and c is the light speed.

To achieve a target range d while keeping the SlimWiFi device at ULP, we propose to reduce B, which can in turn lower the total transmit power P_{TX}. This design choice hinges on the observation that we can treat each subcarrier of the OFDM receiver as an individual narrow-band (312.5 kHz) channel. As long as the SlimWiFi signal falls within one of the subcarriers, it can be captured and demodulated by the OFDM receiver. Therefore, even if its P_{TX} is reduced by $10log_{10}(20000/312.5) = 18$ dB, the total power of a SlimWiFi symbol can still be equivalent to that of a Wi-Fi subcarrier, and SlimWiFi can still keep the same transmission range as a normal Wi-Fi! The operating range can be further traded off for even lower transmit power. In fact, with the 250 kHz OOK symbol rate the SlimWiFi signal bandwidth is 250 kHz which can already fit within one Wi-Fi subcarrier.

Carrier frequency requirement. Most existing communication standards require an accurate carrier frequency. In particular, a highly stable carrier is crucial for synchronizing OFDM TX and RX, and reducing leakage between subcarriers. However, this usually entails a high-profile carrier generator,

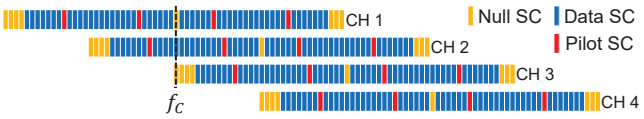


Figure 5: Subcarrier mapping between different channels of Wi-Fi on the 2.4 GHz band. Only 4 channels are illustrated.

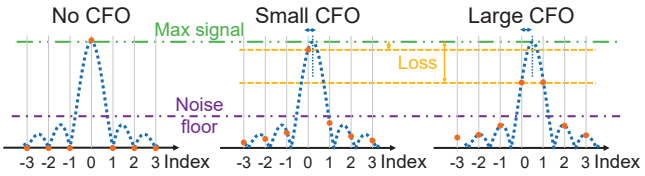


Figure 6: Amplitude of different subcarriers with or without the CFO, when receiving a single tone OOK signal.

consisting of a VCO and PLL which consumes several mW power [54, 66, 71]. The SlimWiFi asymmetric demodulation circumvents this requirement for the first time. As long as the OOK signal's carrier frequency f_C is located within the 20 MHz Wi-Fi band, it can be captured and recovered by demodulating the Wi-Fi receiver's subcarrier that covers f_C. However, two issues need to be solved to accommodate the inaccurate carrier frequency.

First, f_C might be in the non-data subcarriers which are discarded by the Wi-Fi receiver. We overcome this problem by making use of the partially overlapped Wi-Fi channel designated in the 2.4 GHz band, where the non-data subcarriers of one channel are the data subcarriers of an adjacent channel, as shown in Fig. 5. With this mechanism, the carrier frequency requirement can be further relaxed from 20 MHz (a single Wi-Fi channel) to 80 MHz (the entire 2.4 GHz ISM band covering 13 Wi-Fi channels). Note that, the Wi-Fi receiver can identify the subcarrier where the SlimWiFi signal is located by simply checking the subcarrier energy level. If the Wi-Fi receiver does not observe any uplink signal after the trigger frame (Sec. 2), then the signal may fall on a non-data subcarrier, and the receiver should switch to an adjacent channel instead.

The second issue is that the OOK carrier frequency f_C may not be aligned exactly with an OFDM subcarrier. Although OOK can be demodulated non-coherently, the carrier frequency offset (CFO) leads to non-orthogonality in the Wi-Fi receiver's FFT processing, which may in turn affect the asymmetric demodulation. Fig. 6 illustrates a case where a single tone signal (OOK with ON state) spreads to multiple subcarriers due to CFO. Demodulating the OOK signal on a single subcarrier will result in a low SNR. Combining the signal energy across subcarriers does not necessarily help either because it increases the noise bandwidth. Nonetheless, the worst-case SNR loss due to CFO is only 3 dB (signal spreads evenly between two adjacent subcarriers), which will be verified in Sec. 6.2.

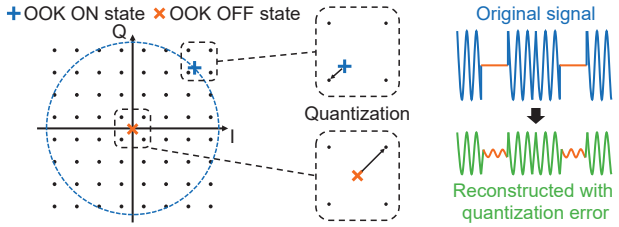


Figure 7: OOK modulated signal with QAM demodulation.

3.4 Resolving Quantization and Coding Errors

3.4.1 QAM and quantization error

The Wi-Fi receiver's QAM demapping block quantizes the phase and amplitude of the signal on each subcarrier. Fig. 7 illustrates the case when a SlimWiFi OOK signal is demapped on a 64-QAM constellation diagram. For the ON state of OOK, the signal sample will have a non-zero amplitude with an arbitrary phase, hence falling at the outer circle. For the OFF state, the sample will have a near-zero amplitude, falling at the origin point. For other subcarriers where no active signals are located, the demapped sample will be the same as the OFF state.

Essentially, the QAM demapping is performing quantization in the complex domain. Thus the original OOK signal on the active subcarrier can be easily reconstructed through the reverse operation, *i.e.*, QAM mapping which converts bits to a complex number. However, this process will introduce quantization errors, which compromises the SNR of the reconstructed signal. The quantization error depends on the precision of quantization which is determined by QAM modulation order. We thus configure the Wi-Fi receiver to the highest modulation order 64-QAM, leading to the lowest quantization error.

3.4.2 FEC and coding error

When receiving the non-OFDM SlimWiFi signal, the FEC block causes a mismatch between the demodulated bit sequence X and regenerated bit sequence X' shown in Fig. 3. The fundamental reasons are two-fold: (i) The demodulated bit sequence can be treated as an arbitrary bit sequence instead of a valid codeword of FEC; (ii) The standard Wi-Fi FEC decoding is a many-to-one mapping, whereas the reverse operation (*i.e.*, FEC encoding in Fig. 3) is a one-to-one mapping. So there is no guarantee that the reconstructed X' can match the original X by simply reversing the FEC.

Fortunately, we found that the number of mismatched bits is limited and can be mitigated with a careful design. The coding errors induced by the two standard FEC schemes in Wi-Fi, *i.e.*, binary convolutional coding (BCC) and low-density parity check (LDPC), are different. Here we only summarize their properties. The detailed proofs are in Appendix A.

(1) Both BCC and LDPC incur fewer coding errors at a higher coding rate. Therefore, we configure the Wi-Fi receiver to the highest available coding rate (*i.e.*, 5/6) when performing the asymmetric demodulation. With this measure, the fraction

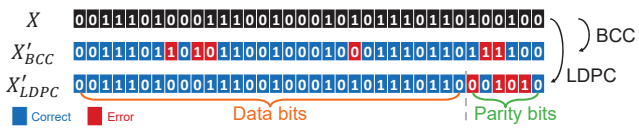


Figure 8: Distribution of coding errors (mismatch between X and X'), for BCC and LDPC, respectively.

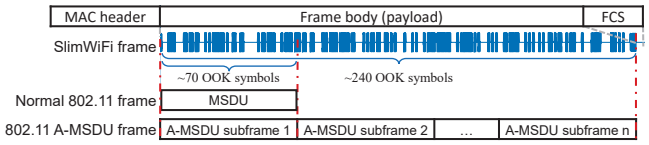


Figure 9: Mapping between the standard Wi-Fi MAC frame and SlimWiFi signal waveform.

of FEC-induced errors can be reduced to around 1/6 and can be further reduced if we apply a separate FEC coding on the SlimWiFi OOK transmitter.

(2) When the Wi-Fi receiver runs the LDPC decoder, the locations of the FEC errors are known a priori on the time-frequency domain. Fig. 8 shows an example of the error distributions when using BCC and LDPC with 5/6 coding rate. X'_{BCC} and X'_{LDPC} are the regenerated bit sequence under BCC and LDPC, respectively. The mismatched bits of the BCC scheme are spread randomly all over the bit sequence X'_{BCC} due to the BCC decoding and interleaving. In contrast, the mismatched bits of the LDPC scheme is always located at the parity bits block (also proven in Appendix A.2).

Based on this observation, we configure the Wi-Fi receiver to LDPC mode in the asymmetric demodulation, which brings two advantages: (i) The error bits are distributed in a periodic way across the reconstructed sequence X' (more details in Sec. 3.6). Therefore, they can be easily corrected by applying a convolutional encoding on the data from SlimWiFi device and using a convolutional decoder on the asymmetric demodulator. (ii) The receiver knows which bits are parity bits (*i.e.*, where the coding errors are clustered). The convolutional decoder can adopt a soft decision decoder which sets those bits with a low log-likelihood ratio, thus improving the decoding performance.

3.5 Practical Challenges

3.5.1 MAC layer configuration

To ensure the MAC payload bits can be used to reconstruct the SlimWiFi signal, we need to resolve two issues: (i) incorrect frame check sequence (FCS), and (ii) limited MAC frame length.

Incorrect FCS. As shown in Fig. 9, the FCS, a 32-bit cyclic redundancy check (CRC) located at the end of the whole frame, is adopted for error protection. Since the received signal is an OOK modulated instead of a valid Wi-Fi signal, it is nearly impossible that the FCS is correct. But we need to capture the data frames through the Wi-Fi driver, even if they fail the FCS check. This is supported by many COTS Wi-Fi devices [2, 21]. A simple software/firmware update can enable the same capability on other Wi-Fi devices.

Data frame length. The length of the payload in a normal Wi-Fi frame is limited by the 2,304 bytes maximum size of the MAC Service Data Unit (MSDU). Recall that SlimWiFi needs to configure the Wi-Fi receiver to the highest data rate (64-QAM, 5/6 code rate, Sec. 3.4). Under this configuration, the maximum number of OFDM symbols is less than 70, corresponding to only 70 OOK symbols as illustrated in Fig. 9. To create a longer frame, we choose to use the aggregate MAC service data unit (A-MSDU) with a quality of service (QoS) data frame, whose maximum size is 7,935 bytes, which extends the frame length to about 240.

3.5.2 Scrambler seed

Since the descrambling is a one-to-one mapping operation on the Wi-Fi receiver, it can be easily reversed by applying a scrambling block with the same scrambler seed. Although the scrambler seed is not reported to the driver, it is set by the PHY header which triggers the receiver's demodulation process (Sec. 3.5.3). Therefore, we can just set a fixed scrambler seed, which can be used to reverse the descrambling block.

3.5.3 Initiating the receiving procedure on Wi-Fi

The final practical challenge lies in generating a valid Wi-Fi preamble and PHY/MAC header. The preamble is needed for triggering the Wi-Fi receiver to start the receiving procedure (packet detection), and is also used for auto gain control, synchronization, and channel estimation. The PHY/MAC header is needed for specifying demodulation parameters such as QAM order, coding rate, scrambling seed, and packet length. Unfortunately, the Wi-Fi preamble and PHY/MAC header are complex OFDM modulated signals, and cannot be directly generated by the SlimWiFi ULP transmitter.

Note that many Wi-Fi devices have separate but co-located transmitter and receiver modules. For example, many Wi-Fi APs [7, 8, 59] usually have multiple transceiver chips (to support concurrent multi-band and multi-antenna operation) which can be configured as co-located TX and RX modules. Therefore, we repurpose the co-located Wi-Fi TX module as an *initiator* to emit a self-initiation frame, comprised of the legitimate preamble and PHY/MAC header but without any payload. Such zero-payload frames are supported by Wi-Fi drivers such as Nexmon [69], or through Wi-Fi frame emulation methods [37]. Upon receiving the initiation frame, the receiver starts its Wi-Fi demodulation workflow followed by the asymmetric demodulation (Fig. 3). Notably, since the transmission of the initiation frame and the reception of OOK data occur consecutively, there is no self-interference between the co-located transmitter and receiver. Therefore, unlike backscatter communication systems, the link budget and receiving sensitivity is not affected by direct Tx leakage or near-far problems [40]. For those Wi-Fi devices with integrated transceivers, a firmware update is needed to enable the receiver to start its demodulation workflow immediately after the transmitter sends out the trigger frame.

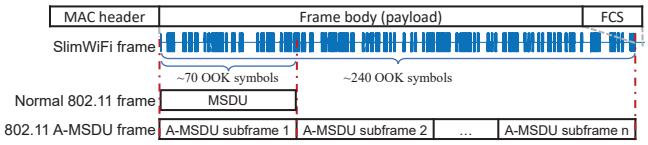


Figure 10: Demodulating the SlimWiFi OOK symbols directly in the frequency domain. The subcarrier with non-zero signal power contains the OOK symbols.

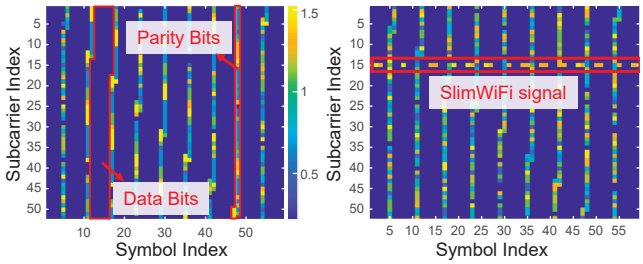
Optimizing receiver gain and sensitivity. A standard Wi-Fi receiver performs automatic gain control (AGC) based on the signal strength of the preamble from the transmitter. For SlimWiFi, since the preamble is from the co-located initiator instead of the actual transmitter, the AGC may be misconfigured. If the initiation frame is too strong, the receiver will set a low gain, leading to insufficient amplification of the incoming SlimWiFi signals. In this situation, the demodulation performance will be bottlenecked by the quantization error (Sec. 3.4.1). Therefore, to achieve the best receiver sensitivity, we would prefer to reduce the power of the initiation frame. This may risk forcing the receiver to tune to a high gain, resulting in the clipping of high amplitude signals. Fortunately, for OOK signals, the clipping effect will not impact demodulation, since clipped signals are recognized as "1" regardless of their amplitude. We will evaluate the effects of the receiver gain in Sec. 6.2.

3.6 Putting Everything Together

Overall, the Wi-Fi receiver follows the processing blocks shown in Fig. 3 to perform the asymmetric demodulation. At a high level, the incoming OOK samples go through the hard-coded normal Wi-Fi demodulation steps which result in a MAC frame. Our asymmetric demodulator reconstructs the complex samples from the MAC frame, by reversing the Wi-Fi demodulation steps, and then decodes the desired bit sequence from the reconstructed samples.

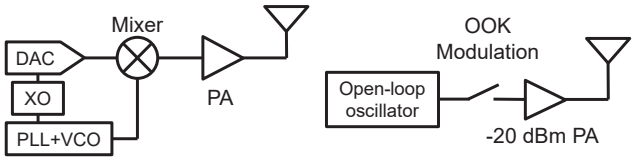
Note that the reverse processing skips the IFFT. Since the OOK signal is narrowband and only occupies one subcarrier, we can directly process the complex samples on that subcarrier, without IFFT-converting them to the time domain, as shown in Fig. 10. The amplitude of the complex sample is used directly to decode the OOK modulated symbol.

To visualize the samples in the time-frequency domain, we collect an example trace with the following configurations: 802.11n with 20 MHz bandwidth, 64-QAM modulation, 5/6 coding rate, LDPC code, and frame length of 2,000 bytes. The waterfall plot in Fig. 11a shows the case without any active transmission. The x and y axis are the symbol index in the time domain, and the subcarrier index in the frequency domain, respectively. The color represents the amplitude of the samples. It can be seen that the samples corresponding to the data bits of the LDPC coded sequence always have a low amplitude (since no coding errors occur there), while the ones corresponding to the parity bits have uncertain results. If we pick the time domain symbols within one subcarrier, the symbols with coding errors (*i.e.* contain parity bits) appear once every 6 symbols. The result corroborates our observations in Sec. 3.4.2.



(a) Without active transmission. **(b)** With OOK signal.

Figure 11: Waterfall plots of reconstructed time-frequency domain samples.



(a) Traditional active transmitter. **(b)** SlimWiFi active transmitter.

Figure 12: Transmitter radio hardware architecture.

Fig. 11b shows the case when a SlimWiFi device is transmitting signals, causing a high amplitude to appear at subcarrier 15 of the Wi-Fi demodulator. The other subcarriers remain the same as the idle case. The OOK signals can thus be demodulated using the samples on subcarrier 15.

4 SlimWiFi ULP Radio Hardware Design

In this section, we focus on the SlimWiFi transmitter hardware, which is designed for asymmetric demodulation. We also provide a brief discussion on the ULP OOK receiver which explains how SlimWiFi device interacts with the COTS Wi-Fi device on the downlink.

4.1 High Power Consumption in Traditional IoT Radios

Modern IoT radio designs need to make challenging trade-offs between power consumption and other competing requirements, including range, bit rate, spectrum efficiency, *etc.* Regardless of how they bias the trade-offs, the IoT radio architecture invariantly comprises 3 key components (Fig. 12a): a high power PA to ensure sufficiently high transmit power; a crystal oscillator (XO) reference and carrier generator consisting of a PLL and VCO, to ensure a stable carrier frequency; a high-resolution DAC to support complex modulation schemes. These high-profile hardware components are the main culprit behind the high power consumption [10].

For example, the industry's most power efficient Wi-Fi radio consumes around 300 mW for TX and 100 mW for RX [34]. BLE consumes 5.1 mW at -20 dBm transmit power and 8.1 mW for RX [61]. ZigBee chip consumes 6.9 mW for transmission and 6 mW receiving [60]. LoRa takes 32.4 mW

Table 1: Power break down of IC implementation

	BLE [63]	SlimWiFi (Simulated)
Power amplifier	2.5 mW	43 µW
Carrier generation	0.7 mW	30 µW
Modulation	0.5 mW	∼0 µW
Rest	0.2 mW	N/A
Sum	3.9 mW	73 µW

and 14.8 mW for TX and RX, respectively [70]. Even the most advanced low power BLE IC [63] which adopts many aggressive optimizations consumes more than 3.9 mW. Table. 1 shows a breakdown of the power consumption of each component. All in all, to achieve extremely low power and open the pathways for battery-free operations, a fundamentally different architecture is needed that evades all the power hungry components.

4.2 SlimWiFi Transmitter Architecture

Owing to the asymmetric communication design (Sec. 3), the SlimWiFi device only needs to generate signals with low transmit power, low-accuracy carrier frequency, and simple OOK waveforms. Therefore, we propose the SlimWiFi active transmitter architecture shown in Fig. 12b. Compared to the traditional active transmitters, the SlimWiFi transmitter: (i) replaces the high-power PA with a low-power PA optimized for constant-amplitude signals at -20 dBm output power; (ii) replaces the closed-loop PLL+VCO with a simple open-loop oscillator; (iii) removes the DAC and uses an RF switch for OOK modulation. With such optimizations, SlimWiFi can bring the power consumption down to 73 µW in simulation. Table. 1 provides the power breakdown of SlimWiFi in comparison with the aforementioned BLE IC. Now we explain how the extremely low power is achieved.

4.2.1 Transmit power

Existing IoT radio designs aim for long-range, high throughput, and robust communication, which in turn requires a high transmit power. For example, Wi-Fi devices usually transmit at more than 20 dBm (*i.e.*, 100 mW). BLE, ZigBee, or LoRa devices are at around 0 dBm (*i.e.*, 1 mW). The transmit power, and the associated PA hardware, dominates the power consumption of the entire transmitter.

For SlimWiFi, recall it can reduce the transmit power by 18 dB while keeping the same link budget, owing to the narrower bandwidth (250 kHz) (Sec. 3). This comes at the cost of a lower bit-rate, but is a much preferred trade-off for most IoT applications, especially considering the existing Wi-Fi infrastructure can be reused. Since the Wi-Fi preamble is generated by the initiator instead of the SlimWiFi device, the PA only needs to support a narrow bandwidth and can be optimized for high efficiency. Our actual on-chip PA is optimized for -20 dBm, whose power consumption can be as low as 43 µW with 24 % drain efficiency. This would be equivalent to a Wi-Fi transmitter at $18 - 20 = -2$ dBm, and comparable to the emission power of BLE, LoRa, and ZigBee radios.

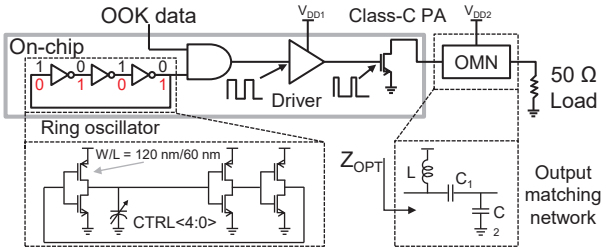


Figure 13: Circuit diagram of the SlimWiFi chip.

However, reducing the transmit power alone cannot bring the peak power to tens of µW. For example, a BLE IC [61] still consumes 4.5 mW when transmitting at -40 dBm (1 μW), and [63] still consumes 1.4 mW even without a PA (Table. 1). At an extremely low transmit power, the carrier generator and modulation blocks will become the bottleneck.

4.2.2 Open-loop carrier generation

Traditional closed-loop carrier generators are based on PLL, which can generate a highly accurate carrier frequency but consumes high power due to the requirement of phase detection. For example, typical analog PLLs for IoT consume power in the mW level [54, 71]. All digital PLLs can potentially bring down the power consumption to several hundred µW [9, 49, 63], but still around one order of magnitude higher than our target power consumption. The asymmetric demodulation design enables SlimWiFi to drastically relax the requirements of frequency stability. Instead of tolerating around 48 kHz (± 20 ppm) of carrier frequency offset as in COTS Wi-Fi devices [43], SlimWiFi works as long as its carrier falls within the 80 MHz range of the entire 2.4 GHz Wi-Fi band! Therefore, SlimWiFi can use an open-loop oscillator with low frequency accuracy as the carrier generator. More specifically, we chose an open-loop ring oscillator for the 2.4 GHz carrier generation which consumes only around 30 µW when implemented on an IC (more details in Sec. 4.3.1).

4.2.3 Low power modulation

To synchronize with the symbol clock of the Wi-Fi receiver (Sec. 3.3.1), the SlimWiFi transmitter uses an RF switch at 250 kHz switching rate to generate the OOK symbols. In fact, our IC implementation realizes OOK by simply powering on and off the PA, without the need of an additional RF switch. Since the open-loop ring oscillator's start-up time (ns level) is much shorter than the symbol period, it can also be power-cycled with the PA, which together can reduce the modulation power consumption to nearly zero.

4.3 IC design

Fig. 13 shows the circuit diagram of our SlimWiFi IC, consisting of an open-loop ring oscillator and a PA optimized for OOK signal at -20 dBm.

4.3.1 Ring oscillator

The ring oscillator consists of an odd number (3-stage in our design) of inverters cascaded into a ring, as illustrated in Fig. 13. The logic input is inverted after passing through the inverters, which causes oscillation between two voltage levels. The open-loop design circumvents the requirement of an external reference clock (*e.g.*, crystal oscillator), thus further reducing the radio cost and form-factor.

The zoom-in plot in Fig. 13 shows the detailed on-chip design of the ring oscillator. It is composed of minimum size transistors (W/ L = 120 nm/ 60 nm) for the minimum area and lowest power consumption. The ring oscillator's actual carrier frequency output is affected by the process, voltage and temperature (PVT) variations. We introduce a 5-bit binary weighted capacitor bank (CTRL$\langle 4:0 \rangle$) loading the first stage of the inverter to tune the propagation delay across different stages of the circuit. This in turn allows us to empirically adjust the oscillation frequency at design time, so it falls within the 2.4 GHz band under typical PVT conditions.

4.3.2 Class-C PA

The carrier is directly modulated by a 250 kHz data sequence and then fed to the inverter-based driver to drive a PA. We choose a Class-C PA for its easy implementation in terms of harmonic terminations and better efficiency at low output power [36]. This comes at the cost of low linearity but is acceptable for SlimWiFi since its OOK waveform is insensitive to clipping distortion (Sec 3.5.3). For a Class-C PA, the relationship between the output power P_{out}, optimal load impedance Z_{OPT} and supply voltage V_{DD} follows [36]:

$$P_{out} = V_{DD}^2/(2 \cdot Z_{OPT})$$

For the target of -20 dBm output power, the optimal load impedance can be 18 kΩ, which would be impractical to match to the standard 50 Ω. To alleviate this problem, a dual-supply voltage scheme [32] is applied for efficiency enhancement. Specifically, we use a 0.9 V V_{DD1} to supply the VCO and driver stage, and 0.3 V V_{DD2} to supply the final PA stage. Off-chip high-Q components [77] are utilized in the tapped-capacitor output matching network to achieve the impedance transformation.

Table. 2 compares the simulated IC performance with and without the PCB parasitic S-parameter (SP) model (extracted using ADS Momentum). Both simulation results are obtained with chip post-layout parasitic extraction (LPE). The table shows that, when co-simulated with the PCB SP model, the output power and efficiency are degraded, indicating that the PCB parasites can have a detrimental effect on the IC performance. This problem can be solved by integrating the capacitors on-chip to ensure a good match and carefully modeling the inductor on PCB to co-optimize the performance.

Another potential solution is to replace the 50 Ω termination with a non-50 Ω antenna. For example, a patch antenna can have an input impedance of 100-400 Ω at resonance [11], which can effectively lower the impedance transformation ratio, thus reducing loss in the matching network.

Table 2: Simulated IC performance

	LPE	LPE +PCB SP
Frequency (MHz)	2451	2438
Pout (dBm)	-19.9	-21.3
Pdrain (μW)	42.9	43.4
Pvco+driver (μW)	29.2	29.3
Drain efficiency (%)	23.7	16.9
Global efficiency (%)	14.1	10.1

4.4 Downlink ULP Receiver

To enable downlink communication for SlimWiFi, the COTS Wi-Fi transmitter needs to emulate OOK waveforms using OFDM. Such emulation has been well explored in recent cross-technology communication and backscatter systems [20, 35, 45, 67], and can be directly adopted by SlimWiFi. The resulting OOK receiver does not need a carrier generator or PA, and thus consumes even less power than the transmitter.

Considering that the TX power of the COTS Wi-Fi device can be 30 dBm, 50 dB higher than the SlimWiFi device's transmit power, a similar uplink and downlink range can be achieved even if the downlink OOK receiver's sensitivity is 50 dB worse than the uplink Wi-Fi receiver. To achieve a 100 m target range, the required receiver sensitivity is 30 dBm + 6 dBi + 2 dBi - 80 dB (FSPL) = -42 dB, which has been achieved in many existing systems. For example, [78] achieves -42.6 dBm sensitivity at 2.8 μW power; [15] achieves -50 dBm sensitivity at 4.5 μW. Much better sensitivity (smaller than -70 dBm) can be achieved with wake-up radio designs [3, 17, 30] at tens of μW power consumption.

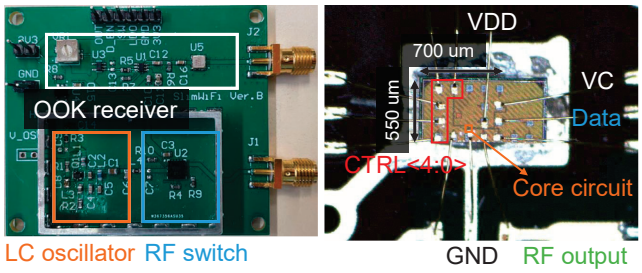
Other than the 2.4 GHz carrier, the SlimWiFi device also requires a 250 kHz symbol clock. Such low frequency clock can be generated with a ULP oscillator (*e.g.*, 0.3 μW [14]) or extracted from the 2.4 GHz carrier through a ULP fraction counting clock as proposed in [84]. The symbol clock can also be calibrated based on the downlink trigger frame which has a 250 kHz OFDM symbol rate.

5 Implementation

5.1 SlimWiFi Device

We have implemented three versions of the SlimWiFi device for different evaluation purposes.

Emulation. To benchmark the performance of the asymmetric demodulation, we need to flexibly control SlimWiFi's signal transmission, such as carrier frequency, symbol time, transmit power, *etc.* Therefore, we use the WARP software radio [56] to emulate the SlimWiFi signals. To faithfully represent the performance of a real SlimWiFi device, we carefully tune the amplitude of the samples and the RF gain of the WARP board, so that the emulated signal has a calibrated transmission power of -20 dBm, consistent with other versions of implementation.



(a) Prototype version. **(b)** IC version.

Figure 14: Two versions of the SlimWiFi device implementation.

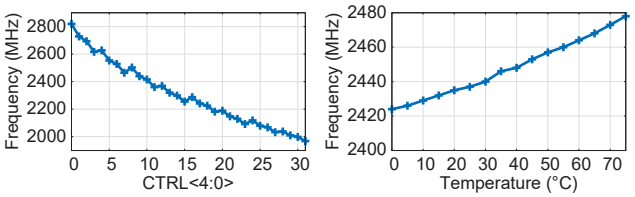
Discrete circuit prototype. The prototype version thoroughly implements both the SlimWiFi TX and RX on a PCB (Fig. 14a), and is used for end-to-end functional validation of the SlimWiFi design. Following the hardware architecture in Sec. 4.2, the TX device consists of an open-loop LC oscillator BFP720 [33] and an RF switch HMC8038 [5] for OOK modulation. The RLC components of the oscillator are carefully designed to tune the oscillation frequency to the 2.4 GHz ISM band. The OOK RX is implemented by a power detector LT5534 [6] and the sensitivity is tuned to -45 dBm. A Cmod A7 [24] FPGA evaluation board is used to process the trigger frame, synchronize the symbol clock, and generate TX data.

IC fabrication. We also tape out a SlimWiFi transmitter following Sec. 4.3 in TSMC 65 nm RF LP process [74] to evaluate its functionality and power consumption. Die photo of the fabricated chip is shown in Fig. 14b, whose core size is 30×25 μm^2. The die is directly bonded to a PCB for testing. More advanced process nodes can be utilized to further scale down the chip size and power consumption.

5.2 COTS Wi-Fi Device

We use DWA-192 [21], a Wi-Fi dongle that supports LDPC code and A-MSDU, to communicate with the SlimWiFi device. To calibrate the antenna gain, we replace the original antennas of unknown gain with two 8 dBi antennas [4]. To implement the asymmetric demodulation on this Wi-Fi receiver, we capture the data frames with CommView [73] on the user space of the PC host and implement the signal processing workflow in Matlab. No additional software, firmware, or hardware modification is needed for receiving.

For the initiation procedure discussed in 3.5.3, the DWA-192 firmware does not support the generation of a zero-payload initiation frame. As a workaround, we verified that a COTS Nexus 5 smartphone with Nexmon Wi-Fi driver [57, 69] can be used as the initiator to send the CTS-to-self, trigger frame and initiation frame, thus triggering the demodulation procedure on DWA-192. However, the signal strength of the COTS devices cannot be well calibrated and controlled which hinders us from benchmarking the impact of the power difference between the initiation frame and the SlimWiFi's signal. Therefore, we use the WARP software radio [56] to send the initiation frame for emulation-based evaluation (Sec. 6.2).



(a) Frequency v.s. CTRL$\langle 4:0\rangle$. **(b)** Frequency v.s. temperature.

Figure 15: SlimWiFi IC carrier frequency drift corresponding to CTRL$\langle 4:0\rangle$ and temperature.

Table 3: SlimWiFi prototype and chip performance

	Frequency (Drift)	Power Consumption @ TX Power
Emulation	Tunable	N/A @ -20 dBm
Prototype	2460 ($\pm$ 5) MHz	1 mW @ -20 dBm
Simulated IC	2438 ($\pm$ 10) MHz	73 μW @ -21 dBm
Fabricated IC	2465 ($\pm$ 10) MHz	90 μW @ -24 dBm

6 System Evaluation

Our evaluation mainly focuses on the SlimWiFi uplink, since the OFDM-to-OOK downlink has been studied in prior research (Sec. 4.4).

6.1 SlimWiFi Device Microbenchmark

We first benchmark the different implementations of the SlimWiFi device. Table. 3 summarizes some important parameters of the SlimWiFi device.

Carrier frequency. We first profile the frequency stability of the SlimWiFi IC with the open-loop ring oscillator. Fig. 15a illustrates the measured carrier frequency when varying the CTRL$\langle 4:0\rangle$ from 0 to 31 with 0.95 V supply voltage at room temperature (25 $^{\circ}C$). We see that the ring oscillator design achieves a wide tuning range (around 1 GHz) and fine steps (30 MHz) compared to the 80 MHz frequency tolerance. In addition, as shown in Fig. 15b, the frequency variance is within 54 MHz even when considering a very wide temperature range of 0 to 75 $^{\circ}C$. Therefore, it suffices to perform a one-time calibration to tune the oscillator to the center of the the 2.4 GHz band and let it run freely.

We found that the emulated and prototype version of SlimWiFi show consistent behavior compared with the IC version. The prototype board also has an inaccurate carrier frequency, though a relatively lower drift (around 5 MHz). The WARP setup can emulate arbitrary carrier frequencies for evaluation purposes.

Power consumption and transmit power. The discrete prototype version of the SlimWiFi transmitter consumes around 1 mW power when transmitting at -20 dBm. This is already superior to state-of-the-art IoT ICs (Sec. 4). The chip version further cuts the power consumption by an order of magnitude owing to the highly optimized oscillator and PA. Sub-100 μW of power consumption is achieved, for both the simulated and fabricated SlimWiFi chips. The measured

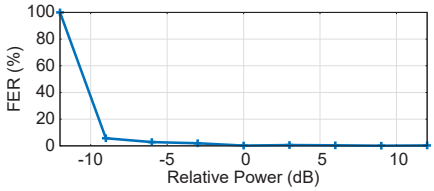


Figure 16: Frame error rate with different relative power between the SlimWiFi signal and the initiation signal.

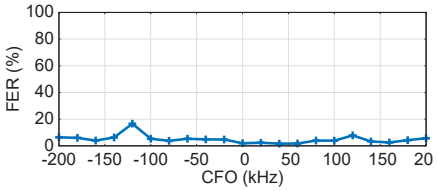


Figure 17: Frame error rate under different carrier frequency offset.

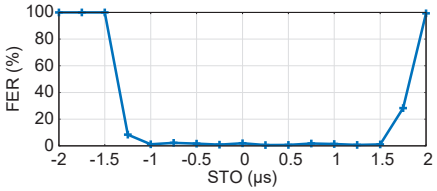


Figure 18: Frame error rate under different symbol time offset.

output power is -24 dBm, which is 3 dB lower than the simulated results. We suspect this is due to the tolerance of the inductor and capacitors used for the high-Q output matching and/or the PCB parasitics (*e.g.* bond wire inductance) not fully captured by the EM simulation. We expect much lower power consumption and a higher PA efficiency is feasible by optimizing the PCB peripherals and by using advanced fabrication processes (lower than 65 nm).

6.2 Microbenchmark for Asymmetric Demodulation

The demodulation performance depends on various parameters, including CFO, symbol time offset (STO), receiver gain, *etc.* Since SlimWiFi uses an open-loop carrier generator that keeps drifting, it is impossible to manually fix these parameters for controlled experiments. We thus calibrated the signal strength and used WARP to decouple and benchmark the impact of each parameter individually.

We conduct link-level experiments in an outdoor parking space, with the following default configurations of the Wi-Fi receiver: 20 MHz 802.11n OFDM, 64-QAM modulation, LDPC coding with 5/6 coding rate and 7935 bytes frame length. Meanwhile, we use WARP to emulate the SlimWiFi device transmitting OOK modulated signals with a frame length of 240 bits and 1/2 BCC coding rate. By default, the link distance is 20 m.

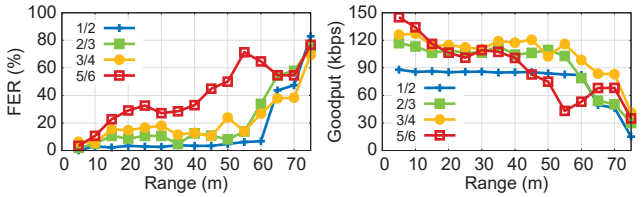
Impact of receiver gain. Recall that the mismatch of signal strength between the initiation signal and the SlimWiFi signal may mislead the Wi-Fi receiver towards a suboptimal gain setting (Sec. 3.5.3). To evaluate its impact, we use the WARP board to transmit the initiation frame along with the emulated signal, so that the strength difference can be intentionally controlled. We consider the relative power of the emulated SlimWiFi signal as 0 dB when the signal strength is the same as that of one subcarrier in the initiation frame. Fig. 16 shows that the receiver performance does not degrade significantly until the relative power is lower than -9 dB, when the receiver gain is too low for robust demodulation. This corroborates our explanation in Sec. 3.5.3. Therefore, instead of adjusting the power of the initiation frame which will lead to complicated management overhead, we can just transmit an initiation frame at a fixed low power. By default, our experiments control the relative power to -6 dB to prevent degrading the demodulation performance.

Impact of carrier frequency offset. Note that the 802.11n subcarrier spacing is 312.5 kHz, and asymmetric demodulation works as long as the SlimWiFi signals overlap with one of the subcarriers. We thus only evaluate the case when the SlimWiFi transmitter's carrier frequency deviates from a representative Wi-Fi subcarrier 15. To achieve higher SNR, we combine the samples of the two subcarriers that partially overlap with SlimWiFi's signals, only when the frequency offsets by 140 to 180 kHz (around half of the subcarrier width). Otherwise, the combination may induce more noise (Sec. 3.3.2). With this setting, the worst-case SNR loss is only 3 dB, *i.e.*, when nearly half of the signal power spills into an unusable adjacent subcarrier. To summarize, the asymmetric demodulator can tolerate arbitrary frequency offsets of the SlimWiFi signals in common cases.

Impact of synchronization. To evaluate how the symbol time offset (STO) influences the receiver performance, we manually introduced a delay between the emulated SlimWiFi signal and the initiation frame (both transmitted by the WARP board). The result in Fig. 18 shows that within an STO from -1 μs to 1.5 μs, the receiver performance is not affected in a noticeable manner. Therefore, the system performance should not be affected by the STO since a much better symbol level synchronization can be achieved by the OOK receiver [78,79]. Notably, the performance is not symmetric around 0 offset (*i.e.*, there is around 0.5 μs more tolerance on positive STO), because of the 0.8 μs redundancy introduced by the CP.

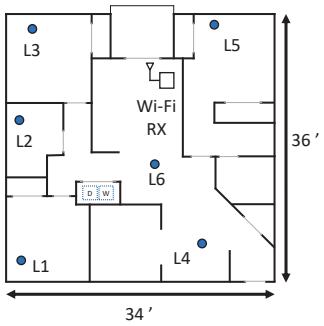
Range and coding rate on SlimWiFi device. Fig. 19a and Fig. 19b show the frame error rate (FER) and goodput with different link distances and BCC coding rate (applied on the data from SlimWiFi device to combat with the coding error discussed in Sec. 3.4.2). The goodput is calculated by only counting the frames with no bit error and including the overhead of channel access, initiation, and trigger frame as discussed in Sec. 2. It can be seen that SlimWiFi maintains a low FER of below 5% even at 60 m of communication range. A goodput of around 100 kbps can be achieved within the range of 60 m. A higher coding rate leads to higher goodput, with some sacrifice on the FER.

Non-line-of-sight (NLoS). We finally evaluate SlimWiFi in an indoor NLoS environment with rich multipath. Fig. 20 shows the deployment setup. We place the Wi-Fi receiver in the living room of a 3B2B apartment, and vary the location of the SlimWiFi transmitter (emulated by WARP). It can be



(a) Frame error rate. **(b)** Goodput.

Figure 19: Performance of the asymmetric demodulation receiver *w.r.t.* (a) frame error rate (FER) and (b) goodput at different range and coding rate.



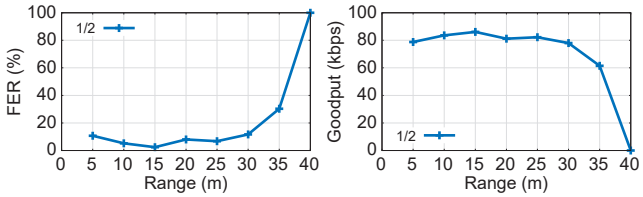
Location	FER
L1	1.382%
L2	0.311%
L3	0.340%
L4	0.218%
L5	0.411%
L6	0.043%

Figure 20: Experimental setup and result for NLoS deployment.

seen that a FER lower than 0.5% is achieved for all the locations except "L1", despite the multipath and under NLoS. A FER of 1.3% can be achieved at "L1" even though the emulated transmitter is placed at the furthest end of the apartment with 2 concrete walls blocking the LoS. We note that the non-coherent demodulation of SlimWiFi is insensitive to the signals' phase variations and naturally resilient to the multipath effects. In addition, as discussed in Sec. 4.2.1, although SlimWiFi bears a low transmit power, it still keeps an ample link budget owing to the high sensitivity of asymmetric demodulation, thereby easily achieving whole-home coverage even with NLoS links.

6.3 System Level Evaluation

We now put the workflow in Fig. 2 together and evaluate the SlimWiFi system end-to-end. We use the prototype SlimWiFi device to transmit an OOK signal with a 1/2 coding rate. The initiator's output power is tuned for the highest receiving gain. The experiments are conducted in an outdoor parking lot. Fig. 21 shows that SlimWiFi can achieve a working range of around 30 m at a FER of 11% and goodput of 78.0 Kbps, and 35 m at a FER of 30% and goodput of 61.5 Kbps. Compared to the result in Fig. 19, the range is reduced by around 1/2. This is reasonable because the impacts of receiver gain, CFO, synchronization error, *etc.* are combined together. For example, unlike the emulated SlimWiFi device, the carrier frequency of the prototype device or IC is not strictly controlled. The resulting carrier frequency offset is unpredictable and will cause up to 3 db of SNR loss (Sec. 6.2) which translates into a range reduction. The result also indicates that the proposed symbol synchronization scheme based on a simple



(a) Frame error rate. **(b)** Goodput.

Figure 21: Performance of the SlimWiFi system *w.r.t.* (a) frame error rate (FER) and (b) goodput at different range.

OOK receiver can satisfy the synchronization requirement.

7 Discussion

Other Wi-Fi standards. We use 802.11n Wi-Fi as the Internet gateway for SlimWiFi devices because the 802.11n standard is supported by mainstream Wi-Fi devices. Other OFDM-based Wi-Fi standards can also support asymmetric modulation, albeit with a few limitations: 802.11a/ac only resides in the 5 GHz band which is not ideal for ULP communication due to the larger path loss; 802.11g, the predecessor of 802.11n, does not support A-MSDU and hence can only accommodate 70 OOK symbols in one frame (Sec. 3.5.1); 802.11ax devices are still not widely deployed and the longer symbol period will lead to lower SlimWiFi throughput.

Initiating the Wi-Fi demodulation. The current SlimWiFi implementation requires an initiator as a workaround to trigger the standard Wi-Fi receiver's demodulation procedure (Sec. 3.5.3). We expect a firmware update to the receiver can enable its self-triggering of the demodulation following the CTS-to-self, as discussed in Sec. 3.5.3. An alternative way to circumvent the initiator is to use the spectral scan function of certain Wi-Fi cards (*e.g.*, the Atheros Wi-Fi [48]), which can continuously report the samples before the QAM block without explicit triggering. We leave the implementation of these approaches for future work.

Ethical consideration. This paper does not involve human subjects and thus does not raise any ethical issues.

8 Related Work

Low-power communication hardware. ULP radio hardware design has been the holy grail of the IoT industry. Many RFIC techniques have been proposed for ULP radios, such as harmonic injection-locked carrier generator [28, 46, 64], crystal-free design [13, 65], power oscillator [58], *etc.* However, these radical radio designs are incompatible with existing IoT network infrastructures. In contrast, SlimWiFi demonstrates for the first time that signals from a ULP OOK radio can be demodulated by a COTS Wi-Fi device. The SlimWiFi ULP radio is extremely simple and can be easily mass-produced and embraced into the existing IoT ecosystem.

We note that most modern network standards have protocol-level power-saving mechanisms [1, 23, 43] based on sleep

Table 4: Comparing SlimWiFi with representative state-of-the-art low-power communication

	Radio architecture	Power	Data rate	Interference	Range	Infrastructure
Wi-Fi [34]	Active	100s mW	High	Low	Long	COTS Wi-Fi
BLE [62]	Active	$\sim$ 5 mW	Medium	Low	Medium	COTS BLE
Wi-Fi backscatter [41]	Direct backscatter	1s µW	Low	Low	Short	COTS Wi-Fi
Braidio [29]	Direct backscatter	10s µW	Medium	Low	Short	Customized device
PassiveWiFi [42]	FS backscatter	10s µW	Medium to high	High	Medium	Single tone generator + COTS Wi-Fi
HitchHike [79]	FS backscatter	10s µW	Medium	High	Medium	COTS Wi-Fi
SlimWiFi	Slim active	10s µW	Medium	Low	Long	COTS Wi-Fi

scheduling. These mechanisms cannot reduce the peak power consumption–a more essential metric for battery-free communication hardware. Nevertheless, they are complementary to the SlimWiFi design and can be used to further reduce its average power consumption.

Cross technology communication (CTC). The primary motivation behind CTC is to allow different communication standards to exchange messages, so as to reduce interference and enable sharing of data/control information. Recent work has explored both receiver-transparent CTC [18, 20, 39, 44, 45, 50] and transmitter-transparent CTC [26, 37, 38, 51]. However, CTC mainly sticks to the complex modulation adopted by the COTS IoT devices. In contrast, SlimWiFi aims to design the SlimWiFi signal so that it can be effectively decoded by high-profile OFDM demodulators while relaxing the hardware requirement of the transmitter. In addition, existing CTC systems can not be used in ULP settings due to two reasons. First, none of the existing CTC designs can reduce power consumption because they rely on standard transceivers such as Wi-Fi, BLE, ZigBee, and LoRa. Second, they have relatively low communication performance. For example, the recently proposed XFi [51] can only reach 10 m range at 3% FER. For such CTC systems, the majority of the energy is wasted to maintain an unreliable link between heterogeneous hardware, which is not desired in ULP IoT applications. In contrast, SlimWiFi is optimized to achieve a reasonable communication performance targeting IoT applications, with around 3 orders of magnitude lower power than standard transceivers.

Backscatter communication. Recent work has extended classical UHF RFID backscatter communication to realize ambient backscatter, which piggybacks on existing communication links to convey information. For example, Wi-Fi backscatter *et al.* [12, 29, 41, 52, 68] adopt direct backscatter where the tag data is directly modulated to the excitation signal. But due to the self-interference, they usually operate within a very short range and have a very low data rate. PassiveWiFi *et al.* [42, 72, 76, 81, 84] introduces frequency shifting backscattering to deal with the self-interference issues. A single-tone excitation signal is required as an RF carrier source for a low-power backscatter tag, and the tag can reflect and remodulate standard-compatible signals (Wi-Fi, BLE, LTE, ZigBee, *etc.*). HitchHike *et al.* [19, 25, 35, 44, 53, 78–80, 83] apply codeword translation, so that a COTS transmitter, instead of a dedicated single-tone generator, can be used as an excitation signal source.

Tab. 4 compares SlimWiFi with the representative communication schemes discussed above. Unlike these systems that backscatter signals from existing links, the SlimWiFi device is a *standalone active transmitter* and does not require an external RF carrier signal transmitter. Moreover, as verified in [22], Wi-Fi backscatter systems can cause interference to adjacent Wi-Fi channels, and may inadvertently remodulate and interfere with 5G NR links due to lack of frequency selectivity. Active transmitters like SlimWiFi do not have such out-of-band interference problems. On the other hand, the asymmetric demodulation design in SlimWiFi can also facilitate existing backscatter systems. Owing to the asymmetric demodulation design of SlimWiFi, the backscatter tag can generate a simple modulated signal instead of the sophisticated Wi-Fi compatible signal. Therefore, the tag can evade the need for an accurate and high frequency (tens of MHz) clock source for channel level frequency shifting, which can potentially cut its power consumption by multi-folds.

9 Conclusion

To our knowledge, SlimWiFi represents the first active OOK-modulated radio that can directly communicate with existing Wi-Fi infrastructures. Such asymmetric communication capabilities enable radical simplifications to the radio architecture, opening pathways towards standalone, battery-free Wi-Fi compatible IoT communication. Our SlimWiFi IC achieves a peak power consumption of 90 µW, but still leaves ample space for optimization, *e.g.*, through more advanced fabrication processes. The asymmetric communication paradigm can be similarly applied to other wireless standards, which we leave for future exploration.

Acknowledgments

We thank our shepherd Rajalakshmi Nandakumar and the anonymous reviewers for their insightful comments and feedback. The work reported in this paper is supported in part by the NSF under Grant CNS-1901048, CNS-1925767, and CNS-2128588.

References

[1] 3GPP. Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3 (3GPP TS 24.301). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1072.

[2] ALFA Network Inc. AWUS036ACM. https://www.alfa.com.tw/products/awus036acm.

[3] Erkan Alpman, Ahmad Khairi, Richard Dorrance, Minyoung Park, V. Srinivasa Somayazulu, Jeffrey R. Foerster, Ashoke Ravi, Jeyanandh Paramesh, and Stefano Pellerano. 802.11g/n compliant fully integrated wake-up receiver with -72-dbm sensitivity in 14-nm finfet cmos. *IEEE Journal of Solid-State Circuits*, 53(5):1411–1422, 2018.

[4] Amazon. 2 x 8dBi WiFi RP-SMA Male Antenna 2.4GHz 5.8GHz Dual Band. https://www.amazon.com/Antenna-Pigtail-Wireless-Routers-Repeater/dp/B07R21LN5P/ref=pd_lpo_1?pd_rd_i=B07R21LN5P&psc=1.

[5] Analog Devices. HMC8038. https://www.analog.com/en/products/hmc8038.html.

[6] Analog Devices. LT5534. https://www.analog.com/en/products/lt5534.html.

[7] ASUSTeK Computer Inc. RT-AC68U. https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-WiFi-Routers/RTAC68U/.

[8] ASUSTeK Computer Inc. RT-AX3000. https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-WiFi-Routers/RT-AX3000/.

[9] Masoud Babaie, Feng-Wei Kuo, Huan-Neng Ron Chen, Lan-Chou Cho, Chewn-Pu Jou, Fu-Lung Hsueh, Mina Shahmohammadi, and Robert Bogdan Staszewski. A fully integrated bluetooth low-energy transmitter in 28 nm cmos with 36% system efficiency at 3 dbm. *IEEE Journal of Solid-State Circuits*, 51(7):1547–1565, 2016.

[10] Torikul Islam Badal, Mamun Bin Ibne Reaz, Mohammad Arif Sobhan Bhuiyan, and Noorfazila Kamal. Cmos transmitters for 2.4-ghz rf devices: Design architectures of the 2.4-ghz cmos transmitter for rf devices. *IEEE Microwave Magazine*, 20(1), 2019.

[11] Constantine A Balanis. *Antenna Theory: Analysis and Design*. John Wiley & Sons, 2016.

[12] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 283–296, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Mengye Cai, Alireza Asoodeh, Yi Luo, and Shahriar Mirabbasi. An ultralow-power crystal-free batteryless tdd radio for medical implantable applications. *IEEE Transactions on Microwave Theory and Techniques*, 68(11):4875–4885, 2020.

[14] Sheng-Kai Chang, Zhi-Ting Tsai, and Kuang-Wei Cheng. A 250 khz resistive frequency-locked on-chip oscillator with 24.7 ppm/°c temperature stability and 2.73 ppm long-term stability. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2020.

[15] Shih-En Chen, Chin-Lung Yang, and Kuang-Wei Cheng. A 4.5 μw 2.4 ghz wake-up receiver based on complementary current-reuse rf detector. pages 1214–1217, 2015.

[16] Xing Chen, Jacob Breiholz, Farah B. Yahya, Christopher J. Lukas, Hun-Seok Kim, Benton H. Calhoun, and David D. Wentzloff. Analysis and design of an ultra-low-power bluetooth low-energy transmitter with ring oscillator-based adpll and 4 $\times$ frequency edge combiner. *IEEE Journal of Solid-State Circuits*, 54(5):1339–1350, 2019.

[17] Kuang-Wei Cheng and Shih-En Chen. An ultralow-power ook/bfsk/dbpsk wake-up receiver based on injection-locked oscillator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(7):1379–1391, 2021.

[18] Zicheng Chi, Yan Li, Yao Yao, and Ting Zhu. Pmc: Parallel multi-protocol communication to heterogeneous iot radios within a single wifi channel. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[19] Zicheng Chi, Xin Liu, Wei Wang, Yao Yao, and Ting Zhu. Leveraging ambient lte traffic for ubiquitous passive communication. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 172–185, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Hsun-Wei Cho and Kang G. Shin. Bluefi: Bluetooth over wifi. In *Proceedings of the ACM SIGCOMM Conference*, 2021.

[21] D-Link. DWA-192. https://us.dlink.com/en/products/dwa-192-ac1900-ultra-wi-fi-usb-adapter.

[22] Farzan Dehbashi, Ali Abedi, Tim Brecht, and Omid Abari. Verification: Can wifi backscatter replace rfid? In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2021.

[23] Artem Dementyev, Steve Hodges, Stuart Taylor, and Joshua Smith. Power consumption analysis of bluetooth low energy, zigbee and ant sensor nodes in a cyclic sleep scenario. In *2013 IEEE International Wireless Symposium (IWS)*, pages 1–4, 2013.

[24] Digilent. Cmod A7. https://digilent.com/reference/programmable-logic/cmod-a7/start.

[25] Manideep Dunna, Miao Meng, Po-Han Wang, Chi Zhang, Patrick Mercier, and Dinesh Bharadia. SyncScatter: Enabling WiFi like synchronization and range for WiFi backscatter communication. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[26] Xiuzhen Guo, Yuan He, Xiaolong Zheng, Zihao Yu, and Yunhao Liu. Lego-fi: Transmitter-transparent ctc with cross-demapping. *IEEE Internet of Things Journal*, 8(8):6665–6676, 2021.

[27] Mohammad Hasan. State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally. https://iot-analytics.com/number-connected-iot-devices/.

[28] Huan Hu, Chung-Ching Lin, and Subhanshu Gupta. A 197.1-μw wireless sensor soc with an energy-efficient analog front-end and a harmonic injection-locked ook tx. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(6):2444–2456, 2021.

[29] Pan Hu, Pengyu Zhang, Mohammad Rostami, and Deepak Ganesan. Braidio: An integrated active-passive radio for mobile devices with asymmetric energy budgets. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 384–397, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Xiongchuan Huang, Simonetta Rampu, Xiaoyan Wang, Guido Dolmans, and Harmke de Groot. A 2.4ghz/915mhz 51µw wake-up receiver with offset and noise suppression. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 222–223, 2010.

[31] Hugues Anguelkov. Reverse-engineering Broadcom wireless chipsets. https://blog.quarkslab.com/reverse-engineering-broadcom-wireless-chipsets.html.

[32] Shunta Iguchi, Akira Saito, Kazunori Watanabe, Takayasu Sakurai, and Makoto Takamiya. Design method of class-f power amplifier with output power of − 20 dbm and efficient dual supply voltage transmitter. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(10):2978–2986, 2014.

[33] Infineon Technologies. BFP720. https://www.infineon.com/cms/en/product/rf/rf-transistor/low-noise-rf-transistors/bfp720/.

[34] InnoPhase. Talaria TWO Modules. https://innophaseinc.com/talaria-two-modules/.

[35] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[36] Daechul Jeong, Hankyu Lee, Taeyoung Chung, Seokwon Lee, Jaesup Lee, and Bumman Kim. Optimized ultralow-power amplifier for ook transmitter with shaped voltage drive. *IEEE Transactions on Microwave Theory and Techniques*, 64(8):2615–2622, 2016.

[37] Woojae Jeong, Jinhwan Jung, Yuanda Wang, Shuai Wang, Seokwon Yang, Qiben Yan, Yung Yi, and Song Min Kim. Sdr receiver using commodity wifi via physical-layer signal reconstruction. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2020.

[38] Wenchao Jiang, Song Min Kim, Zhijun Li, and Tian He. Achieving receiver-side cross-technology communication with cross-decoding. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 639–652, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Wenchao Jiang, Zhimeng Yin, Ruofeng Liu, Zhijun Li, Song Min Kim, and Tian He. Bluebee: A 10,000x faster cross-technology communication via phy emulation. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2017.

[40] Mohamad Katanbaf, Anthony Weinand, and Vamsi Talla. Simplifying backscatter deployment: Full-Duplex LoRa backscatter. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[41] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 607–618, New York, NY, USA, 2014. Association for Computing Machinery.

[42] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Passive Wi-Fi: Bringing low power to Wi-Fi transmissions. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[43] LAN/MAN Standards Committee of the IEEE Computer Society. Ieee standard for information technology–telecommunications and information exchange between systems - local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pages 1–4379, 2021.

[44] Yan Li, Zicheng Chi, Xin Liu, and Ting Zhu. Passivezigbee: Enabling zigbee communication in iot networks with 1000x+ less power consumption. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2018.

[45] Zhijun Li and Tian He. Webee: Physical-layer cross-technology communication via emulation. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2017.

[46] Chung-Ching Lin, Huan Hu, and Subhanshu Gupta. Improved performance tradeoffs in harmonic injection-locked ulp tx for sub-ghz radios. *IEEE Transactions on Microwave Theory and Techniques*, 69(6):2885–2898, 2021.

[47] Linux Wireless. About mac80211. https://wireless.wiki.kernel.org/en/developers/documentation/mac80211.

[48] Linux Wireless. ath9k spectral scan. https://wireless.wiki.kernel.org/en/users/drivers/ath9k/spectral_scan.

[49] Hanli Liu, Dexian Tang, Zheng Sun, Wei Deng, Huy Cu Ngo, and Kenichi Okada. A sub-mw fractional- *N* adpll with fom of -246 db for iot applications. *IEEE Journal of Solid-State Circuits*, 53(12):3540–3552, 2018.

[50] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. Lte2b: Time-domain cross-technology emulation under lte constraints. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.

[51] Ruofeng Liu, Zhimeng Yin, Wenchao Jiang, and Tian He. Xfi: Cross-technology iot data collection via commodity wifi. In *IEEE International Conference on Network Protocols (ICNP)*, 2020.

[52] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 39–50, New York, NY, USA, 2013. Association for Computing Machinery.

[53] Xin Liu, Zicheng Chi, Wei Wang, Yao Yao, Pei Hao, and Ting Zhu. Verification and redesign of OFDM backscatter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[54] Yao-Hong Liu, Johan Van Den Heuvel, Takashi Kuramochi, Benjamin Busze, Paul Mateman, Vamshi Krishna Chillara, Bindi Wang, Robert Bogdan Staszewski, and Kathleen Philips. An ultra-low power 1.7-2.7 ghz fractional-n sub-sampling digital frequency synthesizer and modulator for iot applications in 40 nm cmos. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(5), 2017.

[55] Paolo Madoglio, Hongtao Xu, Kailash Chandrashekar, Luis Cuellar, Muhammad Faisal, William Yee Li, Hyung Seok Kim, Khoa Minh Nguyen, Yulin Tan, Brent Carlton, Vaibhav Vaidya, Yanjie Wang, Thomas Tetzlaff, Satoshi Suzuki, Amr Fahim, Parmoon Seddighrad, Jianyong Xie, Zhichao Zhang, Divya Shree Vemparala, Ashoke Ravi, Stefano Pellerano, and Yorgos Palaskas. 13.6 a 2.4ghz wlan digital polar transmitter with synthesized digital-to-time converter in 14nm trigate/finfet technology for iot and wearable applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 226–227, 2017.

[56] Mango Communications. Wireless Open-Access Research Platform (WARP), 2016.

[57] Matthias Schulz, Daniel Wegemer and Matthias Hollick. Nexmon: The C-based Firmware Patching Framework. https://nexmon.org/.

[58] Patrick P. Mercier, Saurav Bandyopadhyay, Andrew C. Lysaght, Konstantina M. Stankovic, and Anantha P. Chandrakasan. A sub-nw 2.4 ghz transmitter for low data-rate sensing applications. *IEEE Journal of Solid-State Circuits*, 49(7):1463–1474, 2014.

[59] NETGEAR. AX1800 WiFi Router (RAX20). https://www.netgear.com/home/wifi/routers/rax20/.

[60] Nordic Semiconductor. NCS36510. https://www.onsemi.com/products/wireless-connectivity/wireless-rf-transceivers/ncs36510.

[61] Nordic Semiconductor. nRF5340. https://www.nordicsemi.com/Products/nRF5340.

[62] NXP Semiconductors. QN908x. https://www.nxp.com/products/wireless/bluetooth-low-energy/qn908x-ultra-low-power-bluetooth-low-energy-system-on-chip-solution:QN9080.

[63] SeongJin Oh, SungJin Kim, Imran Ali, Truong Thi Kim Nga, DongSoo Lee, YoungGun Pu, Sang-Sun Yoo, Minjae Lee, Keum Cheol Hwang, Youngoo Yang, and Kang-Yoon Lee. A 3.9 mw bluetooth low-energy transmitter using all-digital pll-based direct fsk modulation in 55 nm cmos. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(9), 2018.

[64] Jagdish Pandey and Brian P. Otis. A sub-100 μ w mics/ism band transmitter based on injection-locking and frequency multiplication. *IEEE Journal of Solid-State Circuits*, 46(5):1049–1058, 2011.

[65] Giuseppe Papotto, Francesco Carrara, Alessandro Finocchiaro, and Giuseppe Palmisano. A 90-nm cmos 5-mbps crystal-less rf-powered transceiver for wireless sensor network nodes. *IEEE Journal of Solid-State Circuits*, 49(2):335–346, 2014.

[66] Naser Pourmousavian, Feng-Wei Kuo, Teerachot Siriburanon, Masoud Babaie, and Robert Bogdan Staszewski. A 0.5-v 1.6-mw 2.4-ghz fractional-n all-digital pll for bluetooth le with pvt-insensitive tdc using switched-capacitor doubler in 28-nm cmos. *IEEE Journal of Solid-State Circuits*, 53(9):2572–2583, 2018.

[67] Mohammad Rostami, Xingda Chen, Yuda Feng, Karthikeyan Sundaresan, and Deepak Ganesan. Mixiq: Re-thinking ultra-low power receiver design for next-generation on-body applications. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2021.

[68] Mohammad Rostami, Jeremy Gummeson, Ali Kiaghadi, and Deepak Ganesan. Polymorphic radios: A new design paradigm for ultra-low power communication. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 446–460, New York, NY, USA, 2018. Association for Computing Machinery.

[69] Matthias Schulz, Jakob Link, Francesco Gringoli, and Matthias Hollick. Shadow wi-fi: Teaching smartphones to transmit raw signals and to extract channel state information to implement practical covert channels over wi-fi. In *Proceedings of the 16th ACM Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[70] SEMTECH. SX1261. https://www.semtech.com/products/wireless-rf/lora-core/sx1261.

[71] Kuan-Yueh Shen, Syed Feruz Syed Farooq, Yongping Fan, Khoa Minh Nguyen, Qi Wang, Mark L. Neidengard, Nasser Kurd, and Amr Elshazly. A flexible, low-power analog pll for soc and processors in 14nm cmos. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(7), 2018.

[72] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R. Smith, and Shyamnath Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3), sep 2017.

[73] TamoSoft. CommView for WiFi. https://www.tamos.com/products/commwifi/.

[74] TSMC. 65nm RF LP Process. https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_65nm.

[75] Rudd J.M. Vullers, Rob van Schaijk, Hubregt J. Visser, Julien Penders, and Chris Van Hoof. Energy harvesting for autonomous wireless sensor networks. *IEEE Solid-State Circuits Magazine*, 2(2), 2010.

[76] Anran Wang, Vikram Iyer, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakota. FM backscatter: Enabling connected cities and smart fabrics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[77] Kejia Wang, Sravya Alluri, Xinyu Zhang, and Vincent W. Leung. A sub-100μw 2ghz ook pa for iot applications. In *IEEE Texas Symposium on Wireless and Microwave Circuits and Systems (WMCS)*, 2022.

[78] Po-Han Peter Wang, Chi Zhang, Hongsen Yang, Dinesh Bharadia, and Patrick P. Mercier. 20.1 a 28µw iot tag that can communicate with commodity wifi transceivers via a single-side-band qpsk backscatter communication technique. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 312–314, 2020.

[79] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. In *Proceedings of the ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2016.

[80] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. Freerider: Backscatter communication using commodity radios. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[81] Pengyu Zhang, Mohammad Rostami, Pan Hu, and Deepak Ganesan. Enabling practical backscatter communication for on-body sensors. In *Proceedings of the*

2016 ACM SIGCOMM Conference, SIGCOMM '16, page 370–383, New York, NY, USA, 2016. Association for Computing Machinery.

[82] Xuan Zhang and Alyssa B. Apsel. A low-power, process-and- temperature- compensated ring oscillator with addition-based current source. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(5):868–878, 2011.

[83] Jia Zhao, Wei Gong, and Jiangchuan Liu. Spatial stream backscatter using commodity wifi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[84] Renjie Zhao, Fengyuan Zhu, Yuda Feng, Siyuan Peng, Xiaohua Tian, Hui Yu, and Xinbing Wang. Ofdma-enabled wi-fi backscatter. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[85] Jim Zyren and Al Petrick. Tutorial on Basic Link Budget Analysis. http://www.sss-mag.com/pdf/an9804.pdf.

A FEC Errors in Asymmetric Demodulation

In this section, we discuss the behavior of BCC and LDPC when decoding a non-Wi-Fi frame which supports our design in Sec. 3.4.2

A.1 BCC

Viterbi algorithm is widely adopted for BCC decoding. To achieve the maximum likelihood decoding, the algorithm searches among all valid codewords $\{C\}$, to identify the codeword C^l which has the shortest Hamming distance with the input bit sequence. It then outputs the decoded bit sequence Y which can generate the codeword C^l by performing BCC encoding. This means that when we use the decoded bits Y to get the regenerated bits, the regenerated bits $X' = C^l$ will be the exact codeword that has the shortest hamming distance with the original bit sequence X. Since the demodulated bit sequence X has a very low chance to be the same as a valid codeword, the mismatch between X' and X is almost inevitable. However, we found that the number of mismatches between regenerated bit sequence X' and demodulated bit sequence X has an upper limit. Here we provide a quick proof.

For the BCC code with the basic coding rate 1/2, the codewords are generated by bitwise XOR in Eq. 1 where $d[k]$ is the k-th input data bit and $c_1[k]$ and $c_2[k]$ are the corresponding bits in the codeword.

$$\begin{aligned} c_1[k] &= d[k] \oplus d[k-2] \oplus d[k-3] \oplus d[k-5] \oplus d[k-6] \\ c_2[k] &= d[k] \oplus d[k-1] \oplus d[k-2] \oplus d[k-3] \oplus d[k-6] \end{aligned} \tag{1}$$

Consider a data sequence $D = \{d[1], d[2], ..., d[K]\}$ where K is the length of the input sequence. The corresponding codeword will be $C = \{c_1[1], c_2[1], \cdot, c_1[K], c_2[K]\}$. For one valid codeword C^l generated by D^l, the bitwise inverted version (complementary codeword) $\bar{C}^l = C^l \oplus 1$ will also be a valid codeword whose corresponding data bits is $\bar{D}^l = D^l \oplus 1$. When we get the regenerated bit sequence $X' = C^l$, if the mismatch number between X' and X is more than 1/2 of the total bit number, the mismatch number between $\bar{C}^l$ and X will be smaller than 1/2 of the total bit number. Therefore, the hamming distance between X' and X will be higher than $\bar{C}^l$ and X, which is against the shortest hamming distance principle of the decoder. Therefore, the number of mismatches between regenerated bit sequence X' and demodulated bit sequence X should be lower than 1/2 of the total bit number. Fig. 22 gives an example that illustrates the proof.

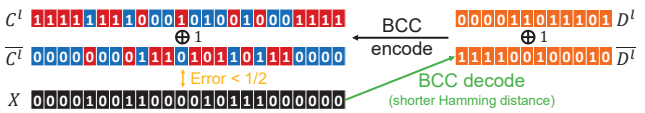


Figure 22: An example of the BCC decoding with complementary codewords at 1/2 coding rate.

For a higher coding rate, the codeword is generated by puncturing the codeword generated by the basic coding rate. Fig. 23 provides an example of how the puncturing is conducted with a 3/4 coding rate while processing the same

sequence in Fig. 22. So the proof still holds, but only for the depunctured sequence. Therefore, to reduce the number of mismatches, we should choose the highest coding rate of 5/6.

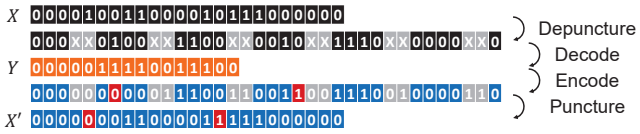


Figure 23: An example of the BCC decoding and regeneration at 3/4 coding rate.

In the previous proof, we only explained the BCC decoding with a hard decision and optimal maximum likelihood decoding. In practice, the error number might vary when considering the soft decision and imperfect maximum-likelihood decoder implementation. But the variation will not diverge the claim.

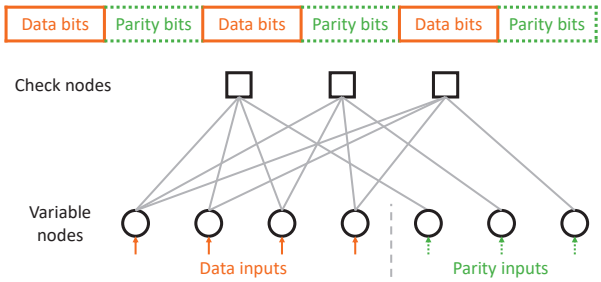


Figure 24: Bit sequence slicing of LDPC coding and an example connection between data bits and parity bits corresponding to one parity-check matrix.

A.2 LDPC

As illustrated in Fig. 24, an LDPC-coded bit sequence is organized into blocks. Each block consists of data bits and parity bits. A predefined parity-check matrix characterizes the connection between variable nodes and check nodes. For LDPC decoding, belief propagation decoders based on the message-passing algorithm are widely adopted. For a soft decision decoder, the inputs of the variable nodes are log-likelihood of the corresponding bits instead of quantized bits. The decoder iteratively updates the log-likelihood of the variable nodes and check nodes based on the inputs and the previous status of the nodes by using the sum-product or min-sum algorithm. After iteratively repeating the log-likelihood update, whether the data or parity bits should be flipped will be determined by the final bit log-likelihood of the variable nodes. The bit-flip of the variable nodes happens when the sum of the log-likelihood from the connected check nodes is larger than the input, which in exchange requires that the inputs have a predefined relation corresponding to the parity-check matrix.

Specific to the SlimWiFi asymmetric demodulation, the bit-flip ratio will be extremely low. This is mainly because the inputs are from the OOK signal which does not have the aforementioned relation. Under such conditions, the LDPC decoder is ineffective when decoding, and thus an extremely limited number of the demodulated bits will be falsely "corrected". A theoretical proof of this conclusion can be found in [51]. Therefore, the data bits part of the regenerated bit sequence will be nearly the same as that of the demodulated bit sequence.

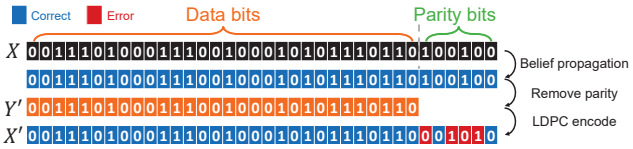


Figure 25: An example of LDPC decoding procedure and the regenerated bit sequence at 5/6 coding rate.

One thing to note is that even though the parity bits part will not be falsely corrected by the decoder, they will be removed after decoding. Since the original data bits do not have a high correlation with the parity bits, the parity bits part of the regenerate bits are not related to that of the demodulated bits. Thus the parity bits part should be treated as unreliable after the regeneration. Then, all bit errors introduced by decoding will be on the parity bits part as illustrated in Fig. 25. Therefore, it is preferable for SlimWiFi to reduce the ratio of parity bits which requires a higher coding rate.

SLNet: A Spectrogram Learning Neural Network for Deep Wireless Sensing

Zheng Yang[1], Yi Zhang[1], Kun Qian[2], Chenshu Wu[3*]

[1] Tsinghua University, [2] University of California San Diego, [3] The University of Hong Kong
{hmilyyz,zhangyithss,qiank10}@gmail.com,chenshu@cs.hku.hk

ABSTRACT

Advances in wireless technologies have transformed wireless networks from a pure communication medium to a pervasive sensing platform, enabling many sensorless and contactless applications. After years of effort, wireless sensing approaches centering around conventional signal processing are approaching their limits, and meanwhile, deep learning-based methods become increasingly popular and have seen remarkable progress. In this paper, we explore an unseen opportunity to push the limit of wireless sensing by jointly employing learning-based spectrogram generation and spectrogram learning. To this end, we present SLNet, a new deep wireless sensing architecture with spectrogram analysis and deep learning co-design. SLNet employs neural networks to generate super-resolution spectrogram, which overcomes the limitation of the time-frequency uncertainty. It then utilizes a novel polarized convolutional network that modulates the phase of the spectrograms for learning both local and global features. Experiments with four applications, *i.e.*, gesture recognition, human identification, fall detection, and breathing estimation, show that SLNet achieves the highest accuracy with the smallest model and lowest computation among the state-of-the-art models. We believe the techniques in SLNet can be widely applied to fields beyond WiFi sensing.

1 INTRODUCTION

We are entering the era of Artificial Intelligence of Things (AIoT) where trillions of devices are pervasively connected and, more importantly, equipped with advanced sensing intelligence. They can sense the physical space and gain awareness of contexts such as locations, activities, motion, vital signs, etc. With advances in wireless sensing, all these could be achieved using pervasive wireless infrastructure, without dedicated sensors, wearables, or cameras. As promising as it is, existing wireless sensing is approaching its limits using conventional signal processing methods and faces performance bottlenecks in distinguishing task-relevant features from entangled irrelevant features in signals.

With its remarkable success in numerous fields, deep learning has become increasingly popular, and also seemingly effective, for wireless sensing, promising the next breakthrough for practical wireless sensing systems for AIoT. Most of the prior works perform conventional signal processing (*e.g.*, frequency transformation) in tandem with deep neural networks, such as convolutional neural networks, which are mainly designed for visual data like images and videos. RF data, most commonly Channel State Information (CSI) data, however, fundamentally differs from visual data in multiple unique aspects: 1) *Non-visual*[1]: RF data contains physical and geometric connotations in time, space, and frequency domains that are not visually intelligible; *2) Complex*: RF data is complex-valued with both amplitude and phase information; *3) High-dimensional*: While visual data are mostly 2D or 3D, RF data comes with multiple dimensions of time, subcarriers, antennas, and/or transceivers. In addition, it is generally more difficult to build a large RF dataset for training than in the computer vision field, both because that RF data collection is cumbersome as it depends on many environmental factors and that RF data cannot be labeled offline since they are not visually intelligible to human eyes. There exists a gap between prior neural networks and the distinct RF data, rendering existing deep wireless sensing systems suboptimal in performance yet over-complicated in model complexity. While there are also other non-visual, complex, and/or high-dimensional data like speech [28, 50], the unique characteristics of RF sensing call for a separate design to push the limit of deep wireless sensing.

We present SLNet, a novel neural network architecture with a *spectrogram analysis-deep learning co-design* for RF data. Rather than performing spectrogram analysis separately from deep learning, SLNet couples them tightly based on an in-depth understanding of their respective limitations in processing RF data. By doing so, SLNet significantly boosts the effectiveness and efficiency of deep wireless sensing. It consists of three major modules:

Learning-Assisted Spectrogram Enhancement: Many

*Zheng Yang is the corresponding author and Yi Zhang is the first student author.

[1] RF data can certainly be visualized in many ways. However, we argue that RF data itself is not visually intelligible like images to humans.

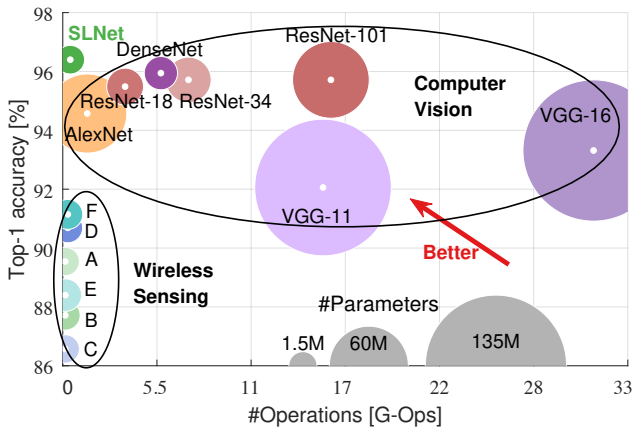


Figure 1: A comparison between SLNet and the state-of-the-art neural networks for WiFi-based gesture recognition task. CV models are accurate but bigger, while existing networks for wireless sensing are relatively small but less accurate. SLNet achieves the highest performance on wireless sensing tasks while reducing computing and memory consumption for practical applications. The radii of the circles represent the number of model parameters. (References: A: [90], B: [22], C: [87], D: [84], E: [30], F: [46])

wireless sensing approaches, either model-driven or data-driven, employ the Fast Fourier Transform (FFT) on a time series of RF data to obtain time-frequency spectrograms of human activities. FFT suffers from errors due to an effect known as leakage, when the block of data is not periodic (the most common case in practice), which results in a smeared spectrum of the original signal and further leads to misleading data representation for learning-based sensing: First, the side lobes "pollute" the spectrograms as they are not from actual human motions but simply the results of spectral leakage. Second, human activities typically contain multiple frequency responses that may be severely affected by the leakage, leading to a "blurred" spectrogram with mixed lobes. Classical approaches reduce leakage by windowing, which cannot eliminate leakage entirely. In effect, they only change the shape of the leakage with different windowing functions to achieve a trade-off between temporal and frequency resolutions. Differently, utilizing learning-based methods promises to push the boundaries beyond classical limitations and, in turn, provide high-fidelity spectrograms for further learning tasks. SLNet introduces a spectrogram enhancement network (§3.1) to learn the best function to minimize or nearly eliminate the leakage, thereby outputting an unparalleled spectrum with high accuracy.

Multi-Resolution Spectrogram Fusion: The frequency resolution of FFT depends on its window size, *i.e.*, the length of the input data block. Using a larger window promises higher resolution, but only generates a more accurate spectrum when the underlying frequency is quasi-static within the window. In contrast, applying a shorter window improves the responsiveness to fast-changing frequencies, but immediately loses high resolution. Therefore, instead of balancing between conflicting goals of resolution and responsiveness by finding a fixed window length, SLNet employs multiple windows jointly and generates a hologram of multi-resolution spectrograms, which then serves as multi-channel inputs that a neural network can adaptively learn from (§3.2).

Polarized Convolutional Network: The hologram is like an image by format, with each spectrogram serving as a "color" channel. Thus it is straightforward to employ convolutional neural networks (CNN) to extract underlying features from it. Invented for visual data, CNN mainly learns local features irrespective of global locations of objects in an image, allowing images to be shifted. Unfortunately, the locality property makes CNN inappropriate for spectrogram learning, as the global locations, *i.e.*, frequencies, are correlated with the physical properties of a person's activities, which is not shift-invariant. To preserve global discrimination, we propose a Polarized Convolutional Network (PCN, §3.3). First, we polarize the spectrograms via specially modulated phase information, making them locally unaltered while globally differentiated. Then we design a special convolutional operator to extract features from the polarized and thus complex-valued spectrogram. Compared to CNN, PCN preserves the local features and the global discrimination simultaneously and thus boosts the learning performance. Based on this, we further adopt a compression network for feature deduction and build a task-adaptive network that can be flexibly customized for different sensing tasks.

We implement SLNet on commodity off-the-shelf (COTS) WiFi devices and evaluate its performance for four human-centered sensing applications, i.e., gesture recognition, gait-based person identification, fall detection, and breathing rate estimation. Extensive experiments are conducted in four typical indoor environments including a classroom, a hall, an apartment, and an office. Our results show that SLNet achieves 96.6% accuracy on gesture recognition, 98.9% accuracy on gait identification, 99.8%/97.2% precision/recall for fall detection, and an average error of 2.4 BPM for multi-person breath estimation. Experimental comparisons with over 10 state-of-the-art deep learning models demonstrate that SLNet achieves the highest accuracy with the fewest model parameters and computation operations, as illustrated in Fig. 1, making it more practical and preferable for edge devices (e.g., home routers).

Contributions: SLNet presents a spectrogram analysis-deep learning co-design network distinctively customized for deep wireless sensing on the time series of high-dimensional, complex-valued RF data. We envision that SLNet inspires

tailored deep-learning architectures that are generalizable to multiple tasks and environments of wireless sensing. Further, we believe the techniques introduced in SLNet, including SEN and PCN, are applicable to many fields involving time–frequency signal analysis and spectrogram learning. SLNet is open-sourced here [41].

2 PRIMER

2.1 Preprocessing of RF Data

CSI reflects the channel through which wireless signals propagate. When a person performs an activity, his or her impact on the channel is encoded in CSI, and the activity can thus be inferred from the CSI. Suppose that the person creates L propagation paths between the transmitter and the receiver, the measured CSI is [36]:

$$H(t) = H_s + \sum_{l=1}^{L} \alpha_l(t) e^{j2\pi \int_{-\infty}^{t} f_{D_l}(u)\mathrm{d}u} + n(t), \tag{1}$$

where α_l, f_{D_l} are the complex attenuation and Doppler frequency shift of the signal of the l-th path, H_s is the static part of the channel between the transmitter and the receiver, and n is the additive Gaussian noise.

To recognize the activity of the person, the raw temporal CSI signals are usually transformed into spectrograms via Short-Time Fourier Transform (STFT):

$$S(f,t) = \mathrm{STFT}[H(t)] = \mathrm{FFT}[\varpi] * \sum_{l=1}^{L} \alpha_l(t)\delta(f - f_{D_l}) + N(f,t), \tag{2}$$

where ϖ represents the windowing function in the time domain, $*$ the convolution operation, and δ the impulse function. N is the frequency response of the Gaussian noise. In Eq. 2, $\sum_{l=1}^{L} \alpha_l(t)\delta(f - f_{D_l})$ reflects the activity of the person. However, it is distorted by the windowing effect of $\mathrm{FFT}[\varpi]$ and the noise N. As a result, the data fidelity of a spectrogram in representing a person's activity is impaired. To remove these negative effects in the spectrogram and make the frequency components of interest prominent, a spectrogram enhancement network is developed in SLNet.

Hereafter, we refer to the 2-D output of STFT as *spectrogram* and the 1-D output of FFT as *spectrum*.

2.2 Complex-Valued Neural Network

The neural network acts as one of the most powerful tools in solving various cognitive problems, such as image classification [24], speech enhancement [16], and text translation [31]. A neural network consists of layers of neurons that generate responses according to their inputs. As shown in Fig. 2a, a neuron calculates the sum of the input $\mathbf{x}$ weighted with parameters $\mathbf{w}$ and the bias and applies a nonlinear activation function σ (e.g., tanh) to generate the output x',

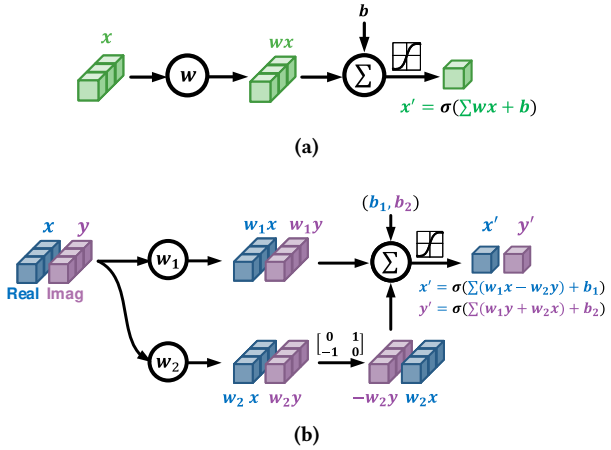


Figure 2: Comparison between (a) real-valued and (b) complex-valued neurons.

i.e., $x' = \sigma(\sum \mathbf{w}\mathbf{x} + b)$. Recently, neural networks have been used for applications of wireless sensing, such as activity classification [22], gait identification [91], and gesture recognition [90]. However, the phase information of CSI is less exploited or even abandoned in the existing approaches. According to Eq. 1, the CSI phase also encodes important information related to the person's activity, which, once exploited, can benefit the recognition process. Thus, instead of using a real-valued neural network, SLNet devises a complex-valued neural network, whose neuron processes complex values and fits the complex-valued spectrogram of CSI. As shown in Fig. 2b, a complex-valued neuron consists of two real-valued neurons, which process the real and imaginary parts of the input, respectively. Specifically, suppose the input is $\mathbf{z} = \mathbf{x} + i\mathbf{y}$, the weight is $\mathbf{w} = \mathbf{w}_1 + i\mathbf{w}_2$, and the bias is $\mathbf{b} = \mathbf{b}_1 + i\mathbf{b}_2$, then the output of the complex neuron is $z' = x' + iy'$, where $x' = \sigma(\mathrm{Re}(\sum \mathbf{w}\mathbf{z} + b))$ and $y' = \sigma(\mathrm{Im}(\sum \mathbf{w}\mathbf{z} + b))$.

3 SLNET ARCHITECTURE

SLNet is designed as a customized spectrogram learning framework assisted with deep learning for RF data applications. It identifies the limitations of the standard signal processing methods for wireless signals and employs specifically designed deep learning modules to overcome them. Fig. 3 shows the workflow of SLNet, which consists of four parts. First, the *spectrogram enhancement network (SEN)* takes as input a spectrogram transformed from wireless signals via STFT, removes the spectral leakage in the spectrogram, and recovers the underlying actual frequency components. Second, the *Fusion* module combines SEN-enhanced spectrograms with various temporal and frequency resolutions to form a hologram of spectrograms. To coherently combine all spectrograms, SLNet modulates them with linear phases,

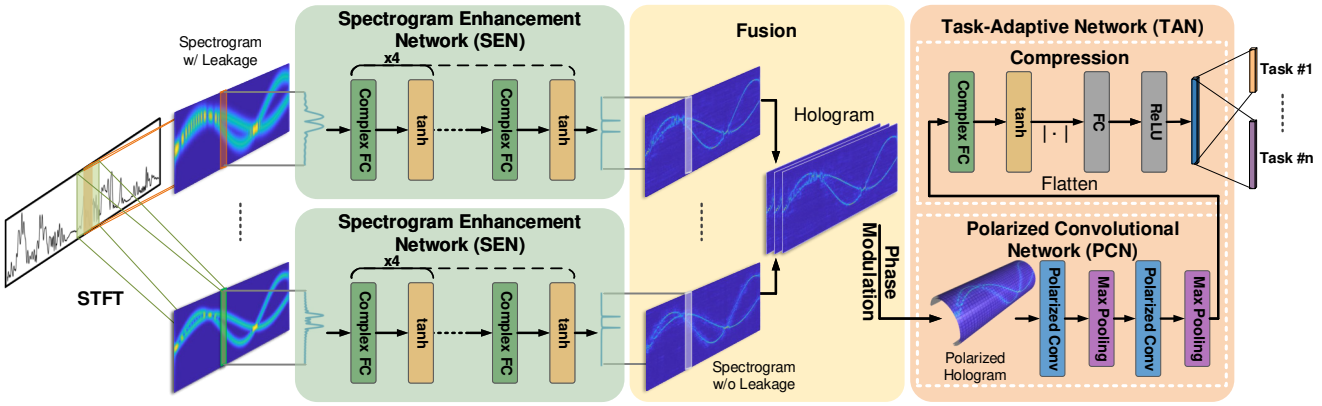


Figure 3: Overview of SLNet. The temporal CSI signal is transformed into spectrograms via a bank of STFT operators with different temporal and frequency resolutions. Each spectrogram is fed into the SEN to remove spectral leakage. Then, a hologram of spectrograms is generated by stacking all enhanced spectrograms and modulating them with linear phases. Next, the hologram is processed with the PCN to generate feature maps, and the compression networks to generate abstract features for specific learning tasks.

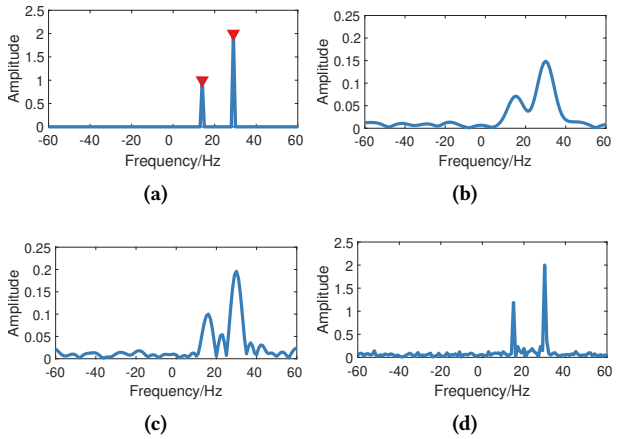


Figure 4: Illustration of spectral leakage. (a) The ideal frequency spectrum with discrete frequency components. (b) The measured frequency spectrum obtained via FFT. (c) The frequency components recovered via least mean square regression. (d) The frequency components recovered from SLNet's SEN.

and the result is termed a polarized hologram. Third, the *polarized convolutional network (PCN)* module processes the hologram to obtain feature maps with general representativity. Finally, we adopt a *compression* network for feature deduction and build a *task-adaptive network (TAN)*, which can be flexibly adapted for different sensing tasks.

3.1 Spectrogram Enhancement Network

Standard signal processing transforms temporal CSI signals to a time-frequency spectrogram via STFT. A certain STFT operator truncates the time series of signals using a sliding window with a fixed length. However, the truncation results in the windowing effect, which convolves the ideal frequency spectrum with a sinc function and creates spectral leakage in the frequency domain. Some classical windowing functions (like Hamming or Gaussian window [13]) can be multiplied with the truncated signal to mitigate the spectral leakage, but none of them completely removes the leakage. Fig. 4a illustrates an ideal frequency spectrum with two frequency components at 15 and 30 Hz. As shown in Fig. 4b, the estimated frequency spectrum obtained via STFT and Gaussian window has significant spectral leakage and additive Gaussian noise. Formally, suppose the ideal and estimated frequency spectrums are $\mathbf{s}$ and $\hat{\mathbf{s}}$ respectively. We have:

$$\hat{\mathbf{s}} = \mathbf{A}\mathbf{s} + \mathbf{n}, \tag{3}$$

where $\mathbf{n}$ represents the additive Gaussian noise vector and $\mathbf{A}$ is the convolution matrix of the windowing function in the frequency domain. Based on Eq. 2, the i-th column of $\mathbf{A}$ is:

$$\mathbf{A}_{(:,i)} = \mathrm{FFT}(\omega) * \delta(i). \tag{4}$$

The spectral leakage significantly distorts the frequency spectrum, producing unwanted side lobes and inaccurate frequencies and amplitudes. For example, when two frequency components are close to each other, their spectral leakage interacts, and the weaker component becomes less prominent, as shown in Fig. 4b. Such spectral leakage is caused by the truncation of the STFT operation and is not relevant to the sensing targets, and is essential to be removed before applying the frequency spectrum to sensing tasks.

Given the relation between the ideal and estimated spectrum as in Eq. 3, it is straightforward to recover the ideal spectrum via the least mean square (LMS) regression:

$$\mathbf{s} = \mathrm{argmin}_{\mathbf{s}} ||\hat{\mathbf{s}} - \mathbf{A}\mathbf{s}||_2. \tag{5}$$

However, the LMS regression tends to output suboptimal

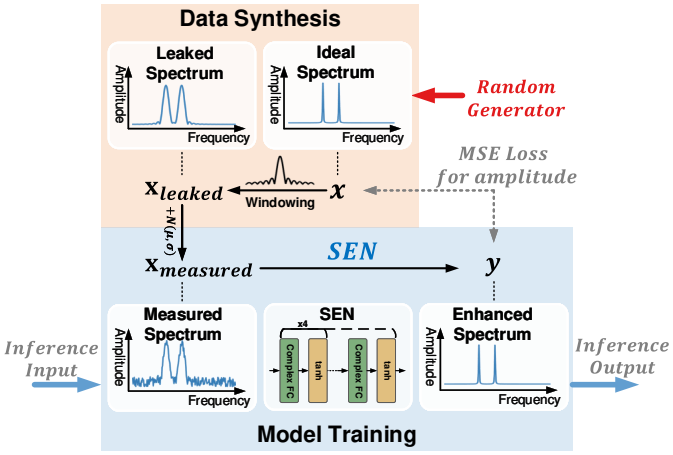


Figure 5: Data synthesis and training process of SEN.

solutions with inaccurate side peaks, as shown in Fig. 4c, due to the existence of Gaussian noises. In contrast, the ideal frequency spectrum of CSI signals tends to be sparse, due to the sparsity of moving objects exposed in the wireless channel [90]. By adding the l_0-norm regularization, the recovered spectrum is closer to the ideal spectrum. However, the l_0-norm regularization makes the computational complexity of the problem exponential to the dimension of the frequency spectrum $\mathbf{s}$ [6], which is thus intractable. Besides, the sparse Fourier transform that aims to solve this issue also suffers from high complexity [14].

To efficiently recover the ideal spectrum, we resort to the neural network. Compared with the optimization methods, the learning-based method offloads the computation efforts to the training phase and enables efficient linear computation in the testing phase. In addition, the neural network can regress arbitrary functions and is resistant to noises thanks to its continuity in the hidden space. To achieve it, we develop the dedicated network SEN. As shown in Fig. 5, the SEN takes as input the measured complex-valued frequency spectrum and outputs the recovered spectrum. The SEN consists of four complex-valued fully connected layers with the hyperbolic tangent activation function.

To train the SEN, the training dataset has to embrace the complexity of the frequency spectrum, which is extremely high due to the wide amplitude and phase range of wireless signals and random channel noises. For example, the frequency of interest for human-centered sensing is within $[-60, 60]$ Hz and usually, at most five frequency components can be observed for major reflections from the human body [55]. That is, after normalizing the signal amplitude, a spectrogram can consist of 1 to 5 frequency components, whose amplitudes, phases, and frequencies are in $[0, 1]$, $[0, 2\pi]$, $[-60, 60]$ Hz, respectively. As collecting data with labeled ground truth from real scenarios is challenging, we instead synthesize the training data, which turns out to be sufficiently effective. As shown in the upper part of Fig. 5, we randomly generate ideal spectrums with 1 to 5 frequency components, whose amplitudes, phases, and frequencies are uniformly drawn from their ranges of interest.

Then, the ideal spectrum is converted to the leaked spectrum following the process in Eq. 3 to simulate the windowing effect and random complex noises. The amplitude of the noise follows a Gaussian distribution, and its phase follows a uniform distribution in $[0, 2\pi]$. The SEN takes the leaked spectrum as input and outputs the enhanced spectrum close to the ideal one. Thus, we minimize the L_2 loss $L_{\text{SEN}} = ||\text{SEN}(\hat{\mathbf{s}}) - \mathbf{s}||_2$ during training. During inference, the spectrums measured from real-world scenarios are normalized to $[0, 1]$ and fed into the SEN to obtain the enhanced spectrum for further processing. Fig. 6 shows an example of the spectrogram when a person performs a gesture. As is shown, the frequency components caused by the pushing and pulling gesture are clearly recovered by the SEN.

3.2 Multi-Resolution Spectrogram Fusion

The SEN refines the frequency spectrum of the CSI signals by recovering its underlying frequency components. Thus, it assumes that the frequency components remain quasi-static during the sliding window where the frequency spectrum is generated. However, the Doppler frequency shifts induced by human activities keep changing, which may violate this assumption. For example, in the fall detection scenario, the speed of the human body changes between 0 m/s and 5 m/s, creating significant variations in signal frequencies. To illustrate its impact on SEN, we use an example of two components with changing frequencies, as shown in Fig. 7a. The measured spectrogram with a sliding window of 251 ms is shown in Fig. 7b and the refined spectrogram from SEN in Fig. 7c. While SEN correctly distinguishes the two components in the first half of the spectrogram, it fails to recover them clearly in the second half, due to the rapidly changing frequencies of the components.

One straightforward solution is to use a shorter sliding window, during which the rapidly changing frequencies can be approximately viewed as quasi-static. However, using a shorter sliding window reduces the frequency resolution, which limits the ability of the SEN to remove the spectral leakage of the two close frequency components. Fig. 7d shows the output of the SEN using a sliding window of 125 ms. With a shorter sliding window, the SEN recovers the second half of the spectrogram with fast-changing frequencies. However, it does not separate well the close frequency components in the first half of the spectrogram, due to the coarse frequency resolution of the short sliding window.

To overcome the limitation of the temporal and frequency

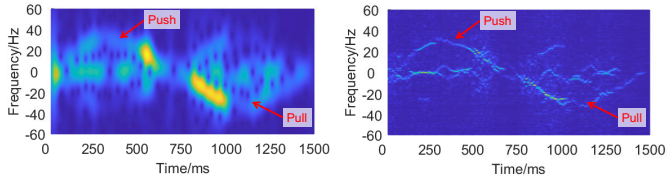


Figure 6: Illustration of the spectrogram of a pushing and pulling gesture. (a) The measured spectrogram and (b) the enhanced spectrogram from SEN.

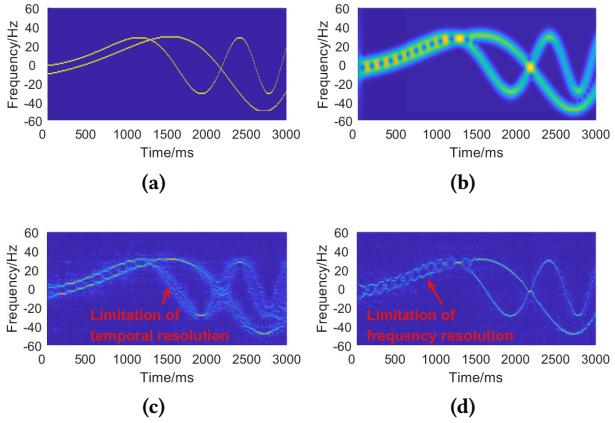


Figure 7: Illustrations of SEN-enhanced spectrograms. (a) The ideal spectrogram with two frequency components. (b) The measured spectrogram from STFT with a sliding window size of 251 ms. (c) The enhanced spectrogram with a window size of 251ms. (d) The enhanced spectrogram with a window size of 125 ms.

resolutions of the spectrogram, instead of using a fixed sliding window, SLNet employs a bank of sliding windows with different lengths (similar to [73]). For each sliding window, the corresponding spectrogram is processed via the SEN that is pre-trained with the synthesis spectrograms with the same window length. All SEN-enhanced spectrograms are then concatenated as multiple channels to form a hologram of spectrograms. As each spectrogram encodes useful information for a certain range of temporal and frequency resolutions, we further resort to the neural network to adaptively combine all the spectrograms.

3.3 Task-Adaptive Network

SLNet employs the Task-Adaptive Network to further adapt the hologram of enhanced spectrograms to various sensing tasks, such as gesture recognition, gait identification, and fall detection. As shown in Fig. 3, the TAN consists of two modules, the PCN module that captures high-level feature maps of the hologram and the compression module that reduces feature dimension for specific tasks.

Polarized Convolutional Network. A hologram can be treated as an image where each spectrogram spanning in 2-D time and frequency dimension is as one of its "color" channels, and, by doing so, a CNN [15, 24] can be applied to extract the underlying features of the hologram. However, the solution is not optimal, as explained below.

Each neuron in CNN only takes a local field of the input to generate the output. All the neurons in each layer share the same weights to ensure that the local features are preserved irrespective of their global locations. As a result, CNN is particularly tailored for visual data since it focuses on local dependencies and is invariant to global shifts of objects in images. This shift-invariant property makes CNN inappropriate for spectrogram processing, as the global locations, i.e., frequencies, of the frequency components are correlated with the physical properties of the person's activities. In another word, a shift along the frequency dimension means a change in the moving status of the person, which is highly possible due to a different activity. Besides, the local patterns of the spectrogram capture the instant motion status of the target, which is still needed for sensing tasks. Hence, it is necessary to develop a new model that simultaneously preserves local dependency and global discrimination.

We propose to modulate the spectrograms with phase information, which is discarded in existing wireless sensing models, to explicitly encode global locations in the spectrogram while retaining its local correlations. Specifically, it is expected that the adjacent frequency components have similar phases while the distant ones have discriminative phases. Thus, we modulate the spectrogram with phases that vary linearly along the frequency dimension, i.e., the modulated phase of the i-th frequency bin is:

$$\phi_i = i\frac{\phi_h - \phi_l}{M} + \phi_l, \tag{6}$$

where ϕ_h and ϕ_l are the phases modulated to the lowest and highest frequency bins, and M is the number of total frequency bins. As a result, global discrimination is introduced along the frequency dimension while the shift-invariant property is preserved along the time dimension. Note that we apply the proposed polarized phase modulation, rather than incorporating the original phase information because the raw phase contains significant errors due to carrier frequency offsets and timing offsets, *etc.* [37, 57].

The frequency components modulated with phases can be viewed as polarized in a 2-D complex plane. To process the polarized spectrograms, we propose the PCN network. PCN consists of two pairs of convolutional layers and max-pooling layers, which are responsible for higher-level feature extraction and dimension reduction, respectively. As shown in Fig. 8a, the polarized hologram is applied with a convolutional layer to extract local features followed by

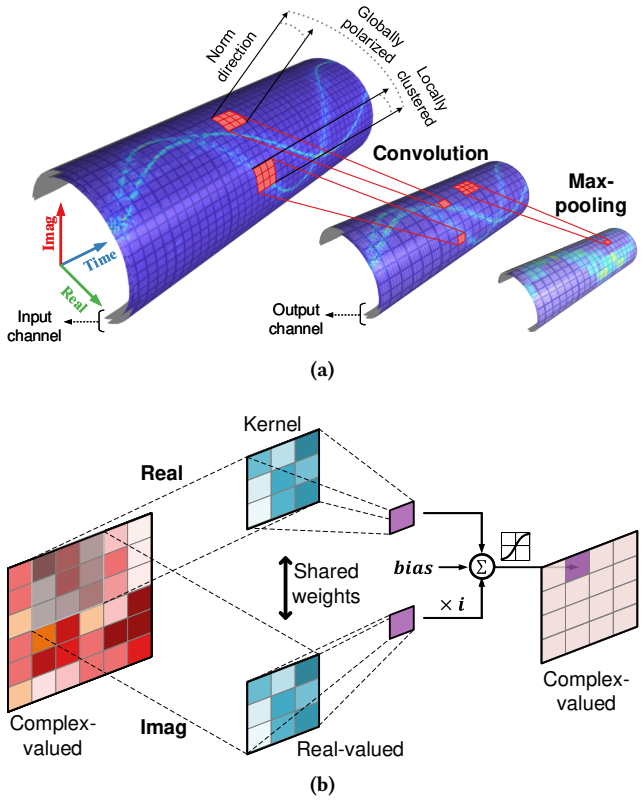


Figure 8: The (a) structure and (b) convolutional operations of the Polarized Convolutional Network.

a max-pooling layer to reduce the feature dimension. The elements within each kernel are locally clustered, while the elements within different kernels are globally polarized to have better global discrimination benefiting from the linear phase introduced. In practice, several convolutional layers can be cascaded to obtain higher-level features of the input spectrograms. The input channel represents the number of fused spectrograms in the hologram or the number of output channels in the previous convolutional layer. For each convolutional layer, Fig. 8b illustrates the convolutional operations in complex domain. The real-valued kernels are convolved with the real and imaginary parts of the spectrograms separately and combined with a bias to get the final complex-valued output. The max-pooling layer downsamples the features with the maximum amplitude and outputs the complex-valued features.

Feature map compression. SLNet further adopts a compression network to reduce the high dimensions of the feature maps generated by the PCN and obtains a condensed representation of features for specific tasks. The compression network consists of a complex-valued fully connected (FC) layer with the *tanh* activation function and a real-valued FC layer with the *ReLU* activation function. To connect two FC layers, SLNet calculates the absolute value of the output from the complex-valued FC layer and inputs it to the real-valued FC layer. The output features can be further fed into additional FC layers customized for different tasks. For example, an FC layer with N output units followed by a softmax activation function can be used for a gesture classification task with N gestures, while an FC layer with 1 output unit followed by a sigmoid layer can be used to predict the likelihood of human fall for the fall detection task.

We employ the pre-trained SENs to obtain the enhanced spectrograms and feed them into the TAN for training. The L_2 loss between the output of the TAN and the ground-truth label is minimized, and the RMSprop optimizer is used. During inference, SLNet takes as input the measured CSI spectrograms and outputs the prediction result.

4 IMPLEMENTATION & EXPERIMENTS

4.1 Implementation

Hardware. SLNet collects CSI measurements from commodity Intel 5300 WiFi Network Interface Cards (NICs) equipped on off-the-shelf mini-computers. The three antennas of the NIC are separated apart by half of the signal wavelength, i.e., 2.85 cm. The operating system of the mini-computer is Ubuntu 10.04 with Linux CSI Tool [12] installed to log CSI readings. The NIC is set to operate on channel 165 with a center frequency of 5.825 GHz. We set all the receivers to work on monitoring mode and inject the transmitter to broadcast at a rate of 1,000 packets per second. All the devices are connected to a router and remotely controlled. We employ a workstation equipped with an NVIDIA GeForce 2080Ti GPU to host the DNN model.

Software. We implement SLNet mainly for benchmark analysis. The data[2] is collected with a Linux shell script, and CSI measurements are preprocessed [36] with Matlab. The dataset [70] and code [69] is available to public. Instructions to use this dataset can be found in our released tutorial [68]. The PyTorch [34] library is adopted to implement the custom complex-valued neurons. Raw CSI is preprocessed in a similar way as in [90]. The SEN module is trained offline with randomly generated spectrums. A total of 5,000 epochs is used to train the SEN models, each of which contains 100 batches × 128 instances of generated spectrums. We pre-train an SEN for each resolution of the spectrogram. Three resolutions are used with window sizes of 125 ms, 251 ms, and 501 ms, respectively. The TAN is trained by the data measured from real deployment scenarios and enhanced by the pre-trained SEN. All the models are trained with Adam optimizer at a learning rate of 0.001. Batch normalization and dropout techniques are applied during training. In practice, the model can be trained offline, except for the use case of

[2] All experiments that involve humans satisfy our IRB requirements.

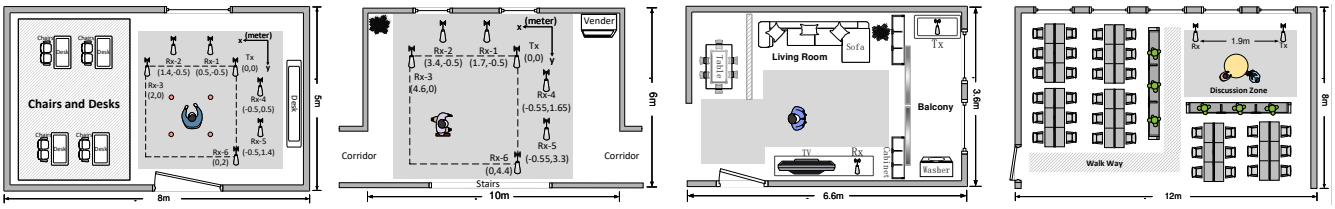


Figure 9: Experimental settings established in SLNet. (a) Classroom for gesture recognition. (b) Hall for gait identification. (c) Apartment for fall detection. (d) Office for breath estimation.

gait recognition where a user needs to register himself first.

4.2 Experiments

We evaluate SLNet in four WiFi sensing applications, including gesture recognition, gait identification, fall detection, and breath estimation. We mainly focus on WiFi CSI sensing in this paper and leave it for the future to explore SLNet's potential for other modalities like acoustic sensing.

Gesture recognition. Device-free gesture recognition [2, 90] is one of the core enablers for human-computer interaction. To evaluate the performance of SLNet for gesture recognition, we conduct experiments in a classroom (sketched in Fig. 9a). One WiFi transmitter and six receivers are placed at a height of 110 cm to capture the motion of human arms. The users are asked to stand at the five marked positions and face the second, third, or fourth quadrant. Eight users (6 males and 2 females) participate in this experiment, with heights varying from 155 cm to 185 cm and ages from 22 to 28. They perform 16 gestures, including 6 sign gestures (push and pull, sweep, clap, slide, draw a circle, and draw zigzag), and 10 input gestures (draw digits 0 to 9). We collect a total of 6,000 data samples (8 people × 6 gestures × 125 instances) for the sign gestures and 5,000 samples (2 people × 10 gestures × 250 instances) for the input gestures. The sign gestures are used in §5.1, and the input gestures are used in the other evaluations. We use the ratio between the number of correctly recognized gestures and the number of all samples as metrics.

Gait identification. Gait has been exploited [18, 64] for human identification. To evaluate the performance of SLNet for gait identification, we conduct experiments in a hall (sketched in Fig. 9b). The devices are placed similarly to that in the gesture experiment. The users are asked to walk freely across the center of the area with eight directions separating 45 degrees apart. Eleven users (7 males and 4 females) participate in this experiment, and their heights vary from 155 cm to 183 cm, and their ages vary from 20 to 26. We collect a total of 3,600 data samples, among which 2,800 samples (7 people × 8 directions × 50 instances) are from 7 users, and 800 samples (4 people × 8 directions × 25 instances) are from the other 4 users. We use the ratio between the number of correctly identified gait samples and the number of all samples as metrics.

Motion	Types	Sub variations
Fall	sit-then-fall, lose-balance, kneel-then-fall, trip walk-then-fall, slip	forward, backward, lateral, on-position
Normal	walk, sit-down/stand-up, run, bend-and-pickup, squat, dance, open/close door, open/close fanner	

Table 1: Fall and normal activities in SLNet

Fall detection. Fall is a major cause of impairment among senior citizens, and some works [33, 52] have tried to detect falls with wireless signals. To evaluate the performance of SLNet for fall detection, we conduct experiments in an apartment (sketched in Fig. 9c). A pair of WiFi devices are placed at the height of 135 cm and 40 cm, respectively, in the living room and the balcony at a distance of 4.5 m. We recruit a voluntary family with 5 members (two males and three females with heights varying from 160 cm to 181 cm and ages varying from 20 to 50). The observed fall and normal activities are described in Tab. 1. When collecting fall data, we require the users to wear protective gear, and the floor is covered with foam. Additionally, we augment the dataset by leveraging a manikin [25] to fall. In total, we collect 2,000 normal instances and 556 fall instances, among which 300 falls are from the manikin, and 256 are from the five users. We use precision and recall as metrics [45].

Breath estimation. Breath rate [67, 78, 83] is an important vital sign that can indicate the condition of physical health. To evaluate the performance of SLNet for breath estimation, we conduct experiments in an office (sketched in Fig. 9d). A pair of WiFi devices are placed at a height of 1 meter to capture the signal reflection from seated participants in the discussion zone. We recruit three participants with heights varying from 172 cm to 185 cm and ages varying from 22 to 26. For each experiment, two of the participants sit in the chairs and breathe naturally. We collect a total of 19 groups of data with a duration of approximately 44 minutes. We use the Breath Per Minute (BPM) error between the estimated

respiration rate and the ground truth as metrics.

We first conduct experiments on all tasks to demonstrate the generality for multiple tasks. Then we carry out ablation and parameter study with gesture recognition as the example task due to the space limit. Unless otherwise stated, the results below are obtained on a 10-fold validation basis where we randomly split the datasets into training and testing parts.

5 EVALUATION

5.1 Comparison Study

5.1.1 Comparison between learning models. To validate the effectiveness of the whole SLNet for wireless sensing, we compare it with 12 typical neural models used in different modalities, including WiFi sensing, FMCW-radar sensing, acoustic sensing, computer vision, and other tasks that leverage Complex-Valued Neural Networks (CVNNs). It is worth noting that the implementations of these networks differ slightly from those in the cited works. As the networks presented in those citations are designed for tasks beyond WiFi sensing, we borrow the backbone architectures and customize them for our own tasks and datasets. The comparison is performed with multiple metrics in terms of model complexity and recognition performance. For model complexity, we evaluate the number of parameters of the models, which is perceived as an effective estimate of the memory requirements and training overhead. For recognition performance, the metrics are discussed in §4.2.

WiFi (4 baseline models): We compare a hybrid CNN-RNN model similar to [23, 90] with three convolutional layers, a GRU layer, and four FC layers; an adversarial model [8, 22] with three convolutional layers as the feature extractor, two FC layers as the activity recognizer, and two FC layers as domain discriminator; an encoder-decoder model with ten FC layers as in [39, 79]; and the time-frequency feature learning model introduced in STFNet [73].

FMCW (2 baseline models): Similar to [87], we design an adversarial model with three convolutional layers and a GRU layer as the feature extractor, two FC layers as the activity recognizer, and two FC layers as the domain discriminator. We further compare a CNN model with three convolutional layers and four FC layers as in [84, 86]. The model is applied to the real and imaginary parts of complex-valued features separately.

Acoustic (1 baseline model): We compare an RNN model with a GRU layer and six FC layers as in [30].

Vision (3 baseline models): We compare three baseline models including VGG-11 [40], resnet-18 [15], and densenet [20]. The input and output layers are reshaped to accommodate our tasks.

CVNN (2 baseline models): We compare two **CVNN** networks that are not originally designed for wireless sensing tasks. Similar to [17, 32], we implement an encoder-decoder model with five complex-valued FC layers and three real-valued FC layers. Similar to [46], we evaluate a CNN model with two complex-valued convolutional layers, two complex-valued FC layers, and two real-valued FC layers.

Modality	Ref.	Gesture	Gait	Fall[1]	Para[2]
WiFi	[23, 90]	90.6%	95.1%	92.8%, 96.3%	1.07M
	[8, 22]	89.0%	96.6%	96.4%, 84.3%	2.72M
	[39, 79]	84.3%	83.3%	96.8%, 93.8%	5.77M
	[73][3]	78.9%	70.9%	95.5%, 96.8%	0.06M
FMCW	[87]	88.0%	95.4%	96.0%, 96.0%	1.06M
	[84, 86]	91.6%	96.4%	99.7%, 95.7%	2.76M
Acoustic	[30]	89.6%	95.4%	90.6%, 98.3%	6.08M
Vision	[40]	88.3%	90.1%	95.3%, 95.3%	128.8M
	[15]	91.9%	96.6%	97.0%, 95.6%	11.18M
	[20]	91.0%	97.7%	99.8%, 96.3%	6.96M
CVNN	[17, 32]	72.3%	96.0%	95.2%, 93.7%	115.6M
	[46]	92.0%	96.3%	98.4%, 93.8%	2.94M
WiFi	**SLNet**	**96.6%**	**98.9%**	**99.8%, 97.2%**	**1.48M**

Table 2: Comparison against 12 baseline models. [1] The two metrics are precision and recall. [2] Number of parameters in Million. [3] Trained with 10,000 epochs to converge.

The input of the baseline model [73] is CSI amplitude with a size of $(T, 30, C)$, where T represents the time snapshots of data samples, *30* represents the number of subcarriers, and C represents the number of WiFi antennas. The input of the other baseline models is raw DFS with a size of $(121, T, C)$, where 121 represents the frequency bins within $[-60, 60]$ Hz. For the signals collected by multiple antennas, we perform PCA analysis on the subcarriers of all three antennas and use the principle components for spectrogram1 analysis.

Tab. 2 presents the performance of the baseline models and SLNet. Three key observations can be derived from the results. First, the models used in computer vision tasks achieve better performance than most of the other baseline models. This is because the vision models are heavily parameterized, which endows them with strong representation capabilities. However, for wireless sensing tasks, a less parameterized neural network is preferable due to the cumbersome data collection and the lack of the wide availability of public datasets. SLNet is designed for this purpose. Second, the models that work in complex domain [17, 32, 46, 84, 86] achieve better performance than those real-valued models. This verifies our assumption that the phase of wireless signals embodies valuable information. SLNet strives to exploit this information with its custom neurons. Third, the advantage of SLNet over baseline models is more significant for the gesture and gait tasks than the fall detection task. This is because fall detection is a binary classification problem that is simpler than the other tasks. SLNet is advantageous in complicated

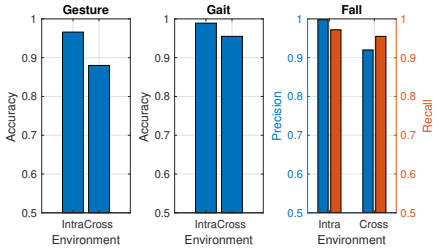


Figure 10: Performance across environments.

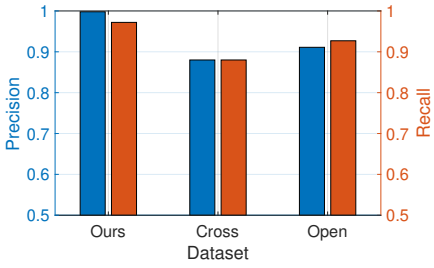


Figure 11: Performance on open dataset [33].

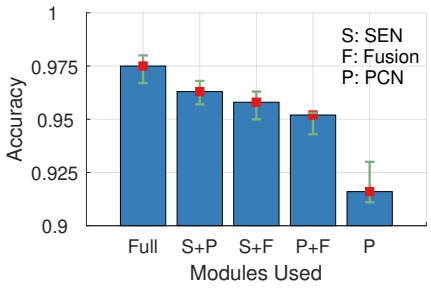


Figure 12: Ablation study on the modules of SLNet.

human sensing scenarios. The most relevant work to SLNet is STFNet [73], which designs customized neural operations to process sensor and RF data. However, while the performance on fall detection is comparable with the other models, it is not desirable for gesture and gait identification tasks. This is because STFNet is built upon the traditional STFT that suffers from spectral leakage. This leakage, when appearing in the spectrogram with clustered frequency components (typical for gesture and gait motion), will distort and even mislead the learning process when searching for the underlying signal structures. The lightly parameterized model in STFNet makes this problem even more challenging. On the contrary, SLNet alleviates this leakage with SEN before learning the hidden features with task-specific networks, ensuring high-fidelity motion representations.

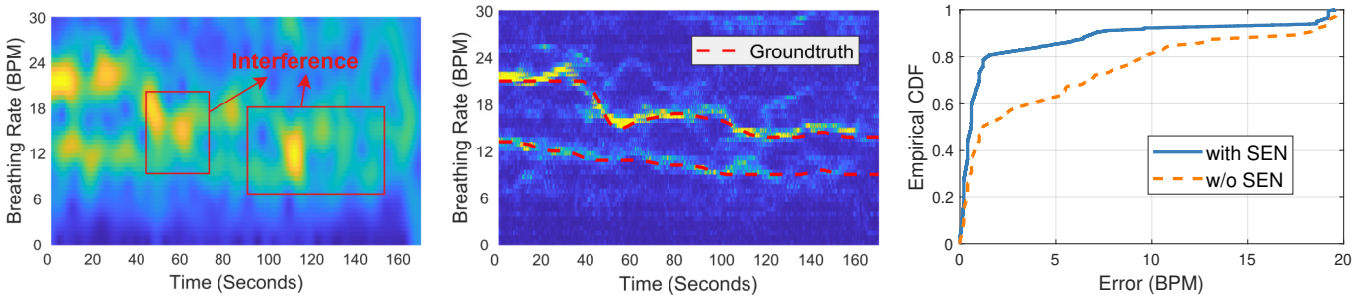
5.1.2 Performance for unseen environments. Wireless sensing systems are prone to environmental changes when deployed in various environments. In this experiment, we evaluate the performance of SLNet when it is applied in *unseen* environments/users after training. Specifically, for the gesture recognition task, we set up another system in an office, which has different layouts and sizes with the classroom illustrated in Fig. 9a. Four volunteers participate in the experiments, and we collect a total of 3,000 gesture samples. For the gait identification task, we collect 600 instances of walking samples from three volunteers in a discussion room, which has a smaller size and more furniture compared with the hall illustrated in Fig. 9b. For the fall detection task, we collect 500 walking samples and 200 falling samples in an office room with different layouts and sizes from the apartment in Fig. 9c. For each task, we train SLNet from scratch with the data collected from one site and test it with the data collected from another.

Fig. 10 demonstrates the performance. As is shown, when deployed in unseen environments without any model adaptation, the performance of SLNet slightly decreases but is still encouraging. SLNet is based on the spectrograms of wireless signals, making it more robust to the surrounding static objects. However, this also makes it prone to changes in relative locations and orientations between users and devices. This is because the frequency components in RF spectrograms are induced by the Doppler shifts, which depend on the moving direction and locations. SLNet is not particularly designed to resolve this problem, yet we believe it can be further addressed by domain adaptation mechanisms [10, 22, 90].

5.1.3 Performance on open datasets. We further evaluate SLNet on a publicly available open datasets. We mainly study fall detection using the dataset released in [33], since it is nearly impossible to find open datasets that have the same types of gestures for gesture recognition or have the same users for gait recognition. This dataset has a total of 181 clearly annotated fall samples and 297 samples of normal activities collected from five rooms of a typical apartment. We compare three settings: 1) We only use our own dataset and split it into non-overlapped train and test parts; 2) We train the model with our dataset and test it with the open dataset; 3) We only use the open dataset and split it into separate train and test parts. The results in Fig. 11 show that SLNet achieves close to 90% precision and recall when trained on our dataset and tested on the open dataset. Despite being degraded, we believe the performance is still encouraging, as the testing data are from completely different and unseen settings with different users, environments, devices, types of falls, etc. Compared with the performance in [33], the precision of our system increases by around 5%, demonstrating the effectiveness of the proposed spectrograms learning pipeline. Considering the promissing performance of SLNet in both intra-domain and cross-domain scenarios, We believe SLNet points a valuable direction to applying wireless sensing systems for real-world applications.

5.2 Ablation Study

SLNet consists of three key modules, i.e., SEN, Fusion, and PCN. To validate their effectiveness, we perform an ablation study for these modules. To do so, we remove it from SLNet while adapting the two modules to the input and output



(a) (b) (c)

Figure 13: Accuracy for breath estimation. (a) The raw spectrogram from traditional FFT. Spectral leakage causes severe interference for the close frequency components, making it hard to differentiate the breath rates of different people. (b) The enhanced spectrogram with SEN. Frequency components can be clearly discriminated. (c) The accuracy of breath rate estimation with raw and enhanced spectrograms.

format. Specifically, for the SEN module, the originally leaked spectrograms are used as the input of the Fusion and PCN modules. For the Fusion module, only the spectrograms with a window size of 251 ms are enhanced by the SEN and used as the input of the PCN. For the PCN module, the output of the Fusion module is fed to two conventional CNN layers and one max-pooling layer, followed by one FC layer as the output layer. In addition, we further remove both the SEN and the Fusion modules to evaluate their joint performance.

As shown in Fig. 12, the accuracy decreases from 97.5% to 96.2%, 95.6% and 95% when the Fusion, PCN and SEN are removed respectively. The accuracy further decreases to 91.2% when only the PCN is used. The result of the ablation study demonstrates the effectiveness of the three modules of SLNet. It is also worth noting that the benefit brought by the SEN module is more significant than others, meaning that the spectral leakage problem cannot be neglected in wireless sensing tasks, and SLNet successfully resolve the problem.

Multi-Person Breathing Rate Estimation Performance. Even though SLNet is designed for motion recognition tasks that attempt to leverage deep learning schemes, its components are valuable beyond that scope. For example, the SEN module can be used to mitigate the spectral leakage induced by the Fourier transform, which could potentially boost the performance of sensing systems that involve spectral analysis. To validate the effectiveness of SEN, we design a human breath estimation experiment in this part.

Breath estimation plays an important role in healthcare, and some recent works [51, 78] exploit the feasibility of using WiFi signals to estimate the respiration rate. A typical way to do this job is to convert the time domain CSI measurements into a frequency domain spectrogram and characterize respiration rates with the prominent frequency components. However, when multiple people breathe concurrently, the spectral leakage problem will severely blur the spectrogram components and make it difficult to discriminate the respiration of different people. With the SEN module, we envision that the leakage will be mitigated or even eliminated, contributing to improved respiration rate estimation accuracy.

In this experiment, two participants sit in chairs and breathe naturally. One of them has just finished some exercise. To obtain ground truth, each of the participants has a smartphone tied to his chest to measure the acceleration of the body induced by breath movements. We apply STFT with a window width of 6,000 (60 seconds) on the acceleration measurements and detect peaks in the spectrogram to represent the respiration rate of each participant. We downsample CSI to 10 Hz and apply STFT with a window width of 251 (25.1 seconds) to get the spectrograms of CSI measurements. We then apply SEN to the spectrograms and pinpoint the two most prominent peaks therein to characterize the respiration rates of the two participants.

Fig. 13a and Fig. 13b demonstrate the raw and enhanced spectrograms of WiFi. As can be seen, the raw spectrogram is severely distorted by the leakage effect, and the frequency components corresponding to the two participates are blurred. By applying SEN, two distinct frequency components can be observed and approximate ground truth very well. Fig. 13c presents the empirical CDF of the respiration rate estimation error. With SEN applied, the average error is 2.4 BPM and the 80%-tile error is 1.4 BPM. Without SEN, the performance deteriorates to 5.0 BPM for average error and 9.5 BPM for 80%-tile error. This experiment demonstrates SEN's strong capability of removing spectral leakage. This merit makes it especially suitable for human-centered sensing tasks, where the human-induced frequency components are tightly clustered and demand to be discriminated.

5.3 Parameter Study

5.3.1 Impact of training epochs of the SEN. In practice,

Figure 14: Impact of SEN training epochs.

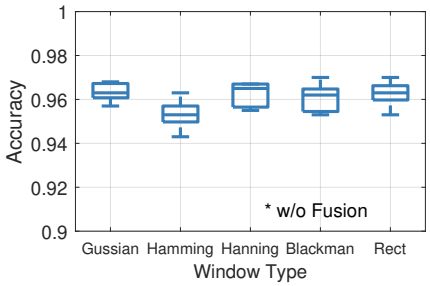


Figure 15: Impact of window type.

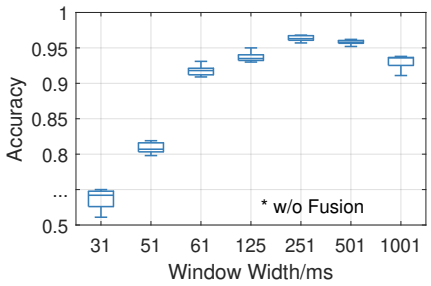


Figure 16: Impact of window width.

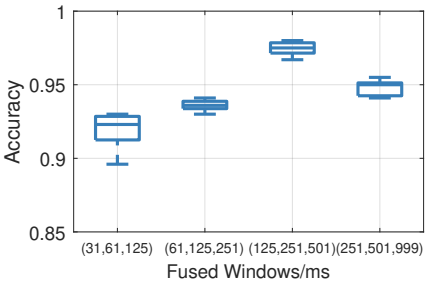


Figure 17: Impact of fused window width.

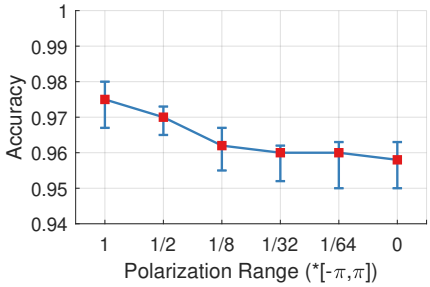


Figure 18: Impact of phase modulation range.

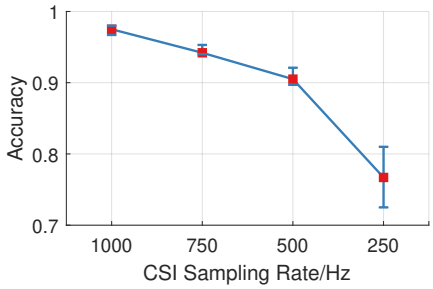


Figure 19: Impact of CSI sampling rate.

we randomly generate data samples during each training epoch. With more training epochs, the SEN should capture the underlying structure of spectrograms better and improve the system recognition accuracy. To reveal the relationship between training epochs and performance, we train the SEN with different numbers of training epochs from 40 to 2500 and integrate them into SLNet. For each SEN, the TAN module is retrained from scratch. We record the validation loss of the SEN and the recognition accuracy of the overall SLNet. As shown in Fig. 14, the validation loss for SEN training tumbles and the overall accuracy proliferates when the epochs increase from 40 to 2,000. The performance becomes stable when the SEN is trained with more epochs.

5.3.2 Impact of window type. Some window functions [13] have been proposed to suppress spectral leakages, such as Hamming, Hanning, and Blackman windows. In this experiment, we evaluate the system performance with regard to these windows. For each window function, we train an independent SEN module with spectrograms calculated with it. The TAN module is then trained from scratch with different SEN modules. We record the overall recognition accuracy concerning different window functions. As shown in Fig. 15, different window functions have little impact on the performance of SLNet, demonstrating the effectiveness of SEN for the removal of spectral leakage.

5.3.3 Impact of window width. In this experiment, we evaluate the impact of the window width on the system performance. Specifically, we train an SEN module for each window width. The Fusion module of SLNet is removed to evaluate each window width. The TAN modules are retrained accordingly, and the overall recognition accuracy is recorded. As shown in Fig. 16, when window width increases from 31 ms to 251 ms, the overall accuracy proliferates from 74% to 96%, but tumbles to 92.5% when window width further increases to 1001 ms. It is because a very small window leads to a coarse-grained frequency resolution, while a very large window cannot capture the rapid change of frequency components in the time domain. The result reveals that a width of 251 ms is the best for the gesture recognition task. However, it is noted that a single-resolution spectrogram for wireless sensing tasks is not the optimal solution, as verified by the ablation study.

5.3.4 Impact of combinations of different window widths. In this experiment, we evaluate the impact of different combinations of window widths. For each combination, we use the corresponding SEN modules to enhance the spectrograms and combine them as holograms. The TAN module is retrained for each combination. The overall recognition accuracy with different combinations is reported. As shown in Fig. 17, the best combination of window widths is $(125, 251, 501)$ ms and the worst is $(31, 61, 125)$ ms. The combination of three windows outperforms either one of these windows independently. The performance could be

further improved with more window widths fused at the cost of increased computational complexity, which is limited in edge devices in practice. For SLNet, the combination of (125, 251, 501) ms is adopted to achieve a trade-off between performance and complexity.

5.3.5 Impact of polarization range of PCN. The PCN module of SLNet is designed to extract both local and global features simultaneously from the holograms. In this experiment, we evaluate the impact of the range of the linear phase modulated to spectrograms in SEN. Specifically, the phases are set to be within $k * [-\pi, \pi]$, where k changes from 0 to 1. For each phase range, we retrain the TAN model from scratch. The overall recognition accuracy concerning different phase ranges is reported. As shown in Fig. 18, the accuracy decreases from 97.5% to 96.0% when the polarization range decreases from $[-\pi, \pi]$ to $[0, 0]$, which reveals that the PCN with phase modulation is effective in spectrogram-based sensing tasks.

5.3.6 Impact of CSI sampling rate. The impact of the CSI sampling rate is evaluated in this part. Specifically, we downsample the original CSI streams (1,000 Hz) before spectral analysis and input the corresponding spectrograms in SLNet. Both SEN and TAN are retrained for each downsampling rate. As shown in Fig. 19, when the CSI sampling rate decreases from 1,000 Hz to 500 Hz, the accuracy gradually decrease from 97.5% to 90% and further tumbles to 76% with a sampling rate of 250 Hz. It is because, with a lower sampling rate, the CSI signals have a poorer temporal resolution and cannot capture the rapidly changing frequency components. In addition, the signal-to-noise ratio decreases with the reduced number of samples for spectral analysis. These factors together deteriorate the recognition performance of SLNet.

6 RELATED WORK

Model-based wireless sensing. Model-based wireless sensing works [2, 4, 38, 49, 52, 60, 62, 74] try to establish quantitative relations between wireless signals and human activities via non-learning based approaches. Many applications have been explored and enabled, including gestures [2, 35, 44, 47, 75], walking [56, 57, 64, 76], falls [19, 21, 33, 45], and respiration [1, 27, 58, 61, 78], and tracking [3, 37, 63, 66], etc. These approaches have the benefit of being interpretable and usually efficient. For example, WiGest [2] empirically builds a link between received signal strength (RSSI) and hand-moving patterns. SMARS [78] exploits breathing estimation by periodicity finding. However, these approaches are constrained by the coarse-grained signal and motion models and are approaching performance limits in real environments. More works are seeking learning-based schemes for better performance, and SLNet is one among them.

Learning-based wireless sensing. Early works mainly rely on signal processing and employ traditional machine learning [48, 56, 57, 59, 64, 76, 77]. With the impressive achievements in computer vision using deep neural networks, more effort [9, 10, 22, 45, 54, 71, 72, 79, 80, 82, 85, 86, 88, 90] has been put into applying deep learning models in wireless sensing tasks. Among them, Widar3.0 [81, 90] leverages CNN and RNN networks to learn from its novel motion feature BVP. RFPose [84], RFPose3D [86], and RFAvatar [85] use CNN models to capture human skeleton and mesh of body. Many works [10, 22, 49, 79, 90] employ sophisticated network architectures like adversarial learning, transfer learning, and meta-learning to solve the environment-dependency problem of wireless sensing, while others aim to reduce cumbersome data collection for training [7, 11, 42, 53, 65]. Some works *e.g.*, [84–86] on FMCW sensing, have further considered customized models for the unique properties of RF data. Existing works either learn from the time series of raw CSI, with both amplitude and phase, or convert them into the frequency-domain representation or other feature space. Despite some time-domain approaches for speech separation [29, 43], recent works like STFNets [73], which extends DeepSense [72], and UniTS [26] both pursue and demonstrate superior performance of temporal-spatial learning with STFT operators. Noticing phase encodes essential spatial information, complex-valued neural networks [5] have been explored in the DL community [17, 32, 46] and exploited especially for radar sensing [89], acoustic sensing and speech processing [28, 50].

7 CONCLUSION

This paper presents SLNet, a spectrogram analysis-deep learning co-design for deep wireless sensing. We demonstrate SLNet's remarkable performance in gesture recognition, gait recognition, fall detection, and breath estimation, showing the highest accuracy and lowest computation compared to the state-of-the-art models. We believe SLNet is a unique deep-learning framework for WiFi sensing. At the same time, the techniques can be used, jointly or separately, to augment the spectrogram quality and enhance learning performance for many applications in signal estimation, frequency analysis, sensing with acoustic/millimeter-wave signals, etc.

ACKNOWLEDGMENTS

We thank our shepherd Dr. Srikanth Kandula and the anonymous reviewers for their feedback, which greatly improved the paper. This work is supported in part by the NSFC under grants No. 61832010, No. 62202262, and No. 62222216, and Hong Kong RGC ECS under grant 27204522.

REFERENCES

[1] Heba Abdelnasser, Khaled A Harras, and Moustafa Youssef. 2015. UbiBreathe: A ubiquitous non-invasive WiFi-based breathing estimator. In *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing*. 277–286.

[2] Heba Abdelnasser, Moustafa Youssef, and Khaled A Harras. 2015. Wigest: A Ubiquitous Wifi-based Gesture Recognition System. In *Proceedings of IEEE INFOCOM*.

[3] Fadel Adib, Zachary Kabelac, and Dina Katabi. 2015. Multi-Person Localization via RF Body Reflections. In *Proceedings of USENIX NSDI*.

[4] Kamran Ali, Alex X Liu, Wei Wang, and Muhammad Shahzad. 2017. Recognizing keystrokes using WiFi devices. *IEEE Journal on Selected Areas in Communications* 35, 5 (May 2017), 1175–1190.

[5] Joshua Bassey, Lijun Qian, and Xianfang Li. 2021. A survey of complex-valued neural networks. *arXiv preprint arXiv:2101.12249* (2021).

[6] Holger Boche, Robert Calderbank, Gitta Kutyniok, Jan Vybíral, et al. 2015. *Compressed sensing and its applications*. Springer.

[7] Hong Cai, Belal Korany, Chitra R Karanam, and Yasamin Mostofi. 2020. Teaching rf to sense without rf training measurements. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 4 (2020), 1–22.

[8] Xi Chen, Hang Li, Chenyi Zhou, Xue Liu, Di Wu, and Gregory Dudek. 2020. FiDo: Ubiquitous Fine-Grained WiFi-Based Localization for Unlabelled Users via Domain Adaptation. In *Proceedings of ACM WWW*.

[9] Zhe Chen, Tianyue Zheng, Chao Cai, and Jun Luo. 2021. MoVi-Fi: Motion-robust vital signs waveform recovery via deep interpreted RF sensing. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 392–405.

[10] Shuya Ding, Zhe Chen, Tianyue Zheng, and Jun Luo. 2020. RF-Net: A Unified Meta-Learning Framework for RF-Enabled One-Shot Human Activity Recognition. In *Proceedings of ACM SenSys*.

[11] Yu Gu, Huan Yan, Mianxiong Dong, Meng Wang, Xiang Zhang, Zhi Liu, and Fuji Ren. 2021. Wione: One-shot learning for environment-robust device-free user authentication via commodity wi-fi in man–machine system. *IEEE Transactions on Computational Social Systems* 8, 3 (2021), 630–642.

[12] Daniel Halperin, Wenjun Hu, Anmol Sheth, and David Wetherall. 2011. Tool Release: Gathering 802.11n Traces with Channel State Information. *SIGCOMM Comput. Commun. Rev.* 41, 1 (2011), 53.

[13] Fredric J. Harris. 1978. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proc. IEEE* 66, 1 (1978), 51–83.

[14] Haitham Hassanieh. 2018. *The Sparse Fourier Transform: Theory and Practice*. Association for Computing Machinery and Morgan & Claypool.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015).

[16] Geoffrey Hinton, li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Phuongtrang Nguyen, Tara Sainath, and Brian Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.

[17] Akira Hirose and Shotaro Yoshida. 2012. Generalization Characteristics of Complex-Valued Feedforward Neural Networks in Relation to Signal Coherence. *IEEE Transactions on Neural Networks and Learning Systems* 23, 4 (2012), 541–551.

[18] Chen-Yu Hsu, Yuchen Liu, Zachary Kabelac, Rumen Hristov, Dina Katabi, and Christine Liu. 2017. Extracting Gait Velocity and Stride Length from Surrounding Radio Signals. In *Proceedings ACM CHI*.

[19] Yuqian Hu, Feng Zhang, Chenshu Wu, Beibei Wang, and K. J. Ray Liu. 2020. A WiFi-based Passive Fall Detection System. In *Proceedings of IEEE ICASSP*.

[20] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. 2016. Densely Connected Convolutional Networks. *CoRR* (2016).

[21] Sijie Ji, Yaxiong Xie, and Mo Li. 2022. SiFall: Practical Online Fall Detection with RF Sensing. In *Proceedings of the Twentieth ACM Conference on Embedded Networked Sensor Systems*. 563–577.

[22] Wenjun Jiang, Chenglin Miao, Fenglong Ma, Shuochao Yao, Yaqing Wang, Ye Yuan, Hongfei Xue, Chen Song, Xin Ma, Dimitrios Koutsonikolas, Wenyao Xu, and Lu Su. 2018. Towards Environment Independent Device Free Human Activity Recognition. In *Proceedings of ACM MobiCom*.

[23] Wenjun Jiang, Hongfei Xue, Chenglin Miao, Shiyang Wang, Sen Lin, Chong Tian, Srinivasan Murali, Haochen Hu, Zhi Sun, and Lu Su. 2020. Towards 3D Human Pose Construction Using Wifi. In *Proceedings of ACM MobiCom*.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of NIPS*.

[25] Laerdal. Accessed 2021. Resusci Anne QCPR Manikin. https://laerdal.com/us/products/simulation-training/resuscitation-training/resusci-anne-qcpr/. (Accessed 2021).

[26] Shuheng Li, Ranak Roy Chowdhury, Jingbo Shang, Rajesh K Gupta, and Dezhi Hong. 2021. UniTS: Short-Time Fourier Inspired Neural Networks for Sensory Time Series Classification. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 234–247.

[27] Jian Liu, Yan Wang, Yingying Chen, Jie Yang, Xu Chen, and Jerry Cheng. 2015. Tracking vital signs during sleep leveraging off-the-shelf wifi. In *Proceedings of the 16th ACM international symposium on mobile ad hoc networking and computing*. 267–276.

[28] Yi Luo, Zhuo Chen, Nima Mesgarani, and Takuya Yoshioka. 2020. End-to-end microphone permutation and number invariant multi-channel speech separation. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 6394–6398.

[29] Yi Luo and Nima Mesgarani. 2019. Conv-tasnet: Surpassing ideal time–frequency magnitude masking for speech separation. *IEEE/ACM transactions on audio, speech, and language processing* 27, 8 (2019), 1256–1266.

[30] Wenguang Mao, Mei Wang, Wei Sun, Lili Qiu, Swadhin Pradhan, and Yi-Chao Chen. 2019. RNN-Based Room Scale Hand Motion Tracking. In *Proceedings of ACM MobiCom*.

[31] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of ICLR*.

[32] Nils Moenning and Suresh Manandhar. 2018. Complex- and Real-Valued Neural Network Architectures. In *International Conference on Learning Representations (openreview)*. https://openreview.net/forum?id=HkCy2uqQM

[33] Sameera Palipana, David Rojas, Piyush Agrawal, and Dirk Pesch. 2019. FallDeFi: Ubiquitous Fall Detection Using Commodity Wi-Fi Devices. In *Proceedings of ACM IMWUT*.

[34] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[35] Qifan Pu, Sidhant Gupta, Shyamnath Gollakota, and Shwetak Patel. 2013. Whole-home gesture recognition using wireless signals. In *Proceedings of the 19th annual international conference on Mobile computing & networking*. 27–38.

[36] Kun Qian, Chenshu Wu, Zheng Yang, Yunhao Liu, and Kyle Jamieson. 2017. Widar: Decimeter-level passive tracking via velocity monitoring with commodity Wi-Fi. In *Proceedings of ACM MobiHoc*.

[37] Kun Qian, Chenshu Wu, Yi Zhang, Guidong Zhang, Zheng Yang, and Yunhao Liu. 2018. Widar2.0: Passive Human Tracking with a Single

Wi-Fi Link. In *Proceedings of ACM MobiSys*.

[38] Kun Qian, Chenshu Wu, Zimu Zhou, Yue Zheng, Zheng Yang, and Yunhao Liu. 2017. Inferring Motion Direction Using Commodity Wi-Fi for Interactive Exergames. In *Proceedings of ACM CHI*.

[39] Cong Shi, Jian Liu, Hongbo Liu, and Yingying Chen. 2017. Smart User Authentication through Actuation of Daily Activities Leveraging WiFi-Enabled IoT. In *Proceedings of ACM MobiHoc*.

[40] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[41] SLNet. 2022. https://github.com/SLNetRelease/SLNetCode. (2022).

[42] Ruiyuan Song, Dongheng Zhang, Zhi Wu, Cong Yu, Chunyang Xie, Shuai Yang, Yang Hu, and Yan Chen. 2022. RF-URL: unsupervised representation learning for RF sensing. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. 282–295.

[43] Daniel Stoller, Sebastian Ewert, and Simon Dixon. 2018. Wave-u-net: A multi-scale neural network for end-to-end audio source separation. *arXiv preprint arXiv:1806.03185* (2018).

[44] Sheng Tan and Jie Yang. 2016. WiFinger: Leveraging commodity WiFi for fine-grained finger gesture recognition. In *Proceedings of the 17th ACM international symposium on mobile ad hoc networking and computing*. 201–210.

[45] Yonglong Tian, Guang-He Lee, Hao He, Chen-Yu Hsu, and Dina Katabi. 2018. RF-Based Fall Monitoring Using Convolutional Neural Networks. *Proceedings of ACM IMWUT* (2018).

[46] Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio, and Christopher J Pal. 2018. Deep Complex Networks. (2018). arXiv:1705.09792

[47] Raghav H Venkatnarayan, Griffin Page, and Muhammad Shahzad. 2018. Multi-user gesture recognition using WiFi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 401–413.

[48] Raghav H. Venkatnarayan, Griffin Page, and Muhammad Shahzad. 2018. Multi-User Gesture Recognition Using WiFi. In *Proceedings of ACM MobiSys*.

[49] Aditya Virmani and Muhammad Shahzad. 2017. Position and Orientation Agnostic Gesture Recognition Using WiFi. In *Proceedings of ACM MobiSys*.

[50] Anran Wang, Maruchi Kim, Hao Zhang, and Shyamnath Gollakota. 2022. Hybrid neural networks for on-device directional hearing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11421–11430.

[51] Fengyu Wang, Feng Zhang, Chenshu Wu, Beibei Wang, and K. J. Ray Liu. 2020. Respiration Tracking for People Counting and Recognition. *IEEE Internet of Things Journal* 7, 6 (2020), 5233–5245.

[52] Hao Wang, Daqing Zhang, Yasha Wang, Junyi Ma, Yuxiang Wang, and Shengjie Li. 2017. RT-Fall: A Real-Time and Contactless Fall Detection System with Commodity WiFi Devices. *IEEE Transactions on Mobile Computing* 16, 2 (2017), 511–526.

[53] Jie Wang, Qinhua Gao, Xiaorui Ma, Yunong Zhao, and Yuguang Fang. 2020. Learning to sense: Deep learning for wireless sensing with less training efforts. *IEEE Wireless Communications* 27, 3 (2020), 156–162.

[54] Mei Wang, Wei Sun, and Lili Qiu. 2021. MAVL: Multiresolution Analysis of Voice Localization. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 845–858.

[55] Wei Wang, Alex X. Liu, Shahzad Muhammad, Kang Ling, and Sanglu Lu. 2017. Device-Free Human Activity Recognition Using Commercial WiFi Devices. *IEEE Journal on Selected Areas in Communications* 35, 5 (2017), 1118–1131.

[56] Wei Wang, Alex X Liu, and Muhammad Shahzad. 2016. Gait Recognition Using WiFi Signals. In *Proceedings of ACM UbiComp*.

[57] Wei Wang, Alex X. Liu, Muhammad Shahzad, Kang Ling, and Sanglu Lu. 2015. Understanding and Modeling of WiFi Signal Based Human Activity Recognition. In *Proceedings of ACM MobiCom*.

[58] Xuyu Wang, Chao Yang, and Shiwen Mao. 2017. TensorBeat: Tensor decomposition for monitoring multiperson breathing beats with commodity WiFi. *ACM Transactions on Intelligent Systems and Technology (TIST)* 9, 1 (2017), 1–27.

[59] Yan Wang, Jian Liu, Yingying Chen, Marco Gruteser, Jie Yang, and Hongbo Liu. 2014. E-eyes: Device-free Location-oriented Activity Identification Using Fine-grained WiFi Signatures. In *Proceedings of ACM MobiCom*.

[60] Yuxi Wang, Kaishun Wu, and Lionel M. Ni. 2017. WiFall: Device-Free Fall Detection by Wireless Networks. *IEEE Transactions on Mobile Computing* 16, 2 (2017), 581–594.

[61] Chenshu Wu, Zheng Yang, Zimu Zhou, Xuefeng Liu, Yunhao Liu, and Jiannong Cao. 2015. Non-invasive detection of moving and stationary human with WiFi. *IEEE Journal on Selected Areas in Communications* 33, 11 (2015), 2329–2342.

[62] Chenshu Wu, Zheng Yang, Zimu Zhou, Kun Qian, Yunhao Liu, and Mingyan Liu. 2015. PhaseU: Real-time LOS identification with WiFi. In *Proceedings of IEEE INFOCOM*.

[63] Chenshu Wu, Feng Zhang, Yusen Fan, and KJ Ray Liu. 2019. RF-based inertial measurement. In *Proceedings of the ACM Special Interest Group on Data Communication*. 117–129.

[64] Chenshu Wu, Feng Zhang, Yuqian Hu, and K. J. Ray Liu. 2020. GaitWay: Monitoring and Recognizing Gait Speed Through the Walls. *IEEE Transactions on Mobile Computing* (2020).

[65] Rui Xiao, Jianwei Liu, Jinsong Han, and Kui Ren. 2021. OneFi: One-Shot Recognition for Unseen Gesture via COTS WiFi. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 206–219.

[66] Yaxiong Xie, Jie Xiong, Mo Li, and Kyle Jamieson. 2019. MD-Track: Leveraging Multi-Dimensionality for Passive Indoor Wi-Fi Tracking. In *Proceedings of ACM MobiCom*.

[67] Xiangyu Xu, Jiadi Yu, Yingying Chen, Yanmin Zhu, Linghe Kong, and Minglu Li. 2019. BreathListener: Fine-Grained Breathing Monitoring in Driving Environments Utilizing Acoustic Signals. In *Proceedings ACM MobiSys*.

[68] Zheng Yang, Yi Zhang, Guoxuan Chi, and Guidong Zhang. 2022. Hands-on Wireless Sensing with Wi-Fi: A Tutorial. (2022). https://arxiv.org/abs/2206.09532

[69] Zheng Yang, Yi Zhang, Kun Qian, and Chenshu Wu. 2023. SLNet Release Code. https://github.com/SLNetRelease/SLNetCode. (2023).

[70] Zheng Yang, Yi Zhang, Guidong Zhang, and Yue Zheng. 2020. Widar 3.0: WiFi-based Activity Recognition Dataset. (2020). https://doi.org/10.21227/7znf-qp86

[71] Zheng Yang, Yi Zhang, and Qian Zhang. 2022. Rethinking Fall Detection With Wi-Fi. *IEEE Transactions on Mobile Computing* (2022).

[72] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. 2017. DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing. In *Proceedings of ACM WWW*.

[73] Shuochao Yao, Ailing Piao, Wenjun Jiang, Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Jinyang Li, Tianshi Wang, Shaohan Hu, Lu Su, Jiawei Han, and Tarek Abdelzaher. 2019. STFNets: Learning Sensing Signals from the Time-Frequency Perspective with Short-Time Fourier Neural Networks. In *Proceedings of ACM WWW*.

[74] Nan Yu, Wei Wang, Alex X. Liu, and Lingtao Kong. 2018. QGesture: Quantifying Gesture Distance and Direction with WiFi Signals. *Proceedings of ACM IMWUT* 2, 1 (2018), 23.

[75] Nan Yu, Wei Wang, Alex X Liu, and Lingtao Kong. 2018. QGesture:

Quantifying gesture distance and direction with WiFi signals. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 1 (2018), 1–23.

[76] Yunze Zeng, Parth H Pathak, and Prasant Mohapatra. 2016. WiWho: WiFi-Based Person Identification in Smart Spaces. In *Proceedings of ACM/IEEE IPSN*.

[77] Shuangjiao Zhai, Zhanyong Tang, Petteri Nurmi, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2021. RISE: Robust wireless sensing using probabilistic and statistical assessments. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 309–322.

[78] Feng Zhang, Chenshu Wu, Beibei Wang, Min Wu, Daniel Bugos, Hangfang Zhang, and KJ Ray Liu. 2019. SMARS: Sleep monitoring via ambient radio signals. *IEEE Transactions on Mobile Computing* 20, 1 (2019), 217–231.

[79] Jie Zhang, Zhanyong Tang, Meng Li, Dingyi Fang, Petteri Tapio Nurmi, and Zheng Wang. 2018. CrossSense: Towards Cross-Site and Large-Scale WiFi Sensing. In *Proceedings of ACM MobiCom*.

[80] Yi Zhang, Zheng Yang, Guidong Zhang, Chenshu Wu, and Li Zhang. 2021. XGest: Enabling Cross-Label gesture recognition with RF signals. *ACM Transactions on Sensor Networks (TOSN)* 17, 4 (2021), 1–23.

[81] Yi Zhang, Yue Zheng, Kun Qian, Guidong Zhang, Yunhao Liu, Chenshu Wu, and Zheng Yang. 2021. Widar3. 0: Zero-effort cross-domain gesture recognition with wi-fi. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[82] Yi Zhang, Yue Zheng, Guidong Zhang, Kun Qian, Chen Qian, and Zheng Yang. 2021. GaitSense: towards ubiquitous gait-based human identification with Wi-Fi. *ACM Transactions on Sensor Networks (TOSN)* 18, 1 (2021), 1–24.

[83] Mingmin Zhao, Fadel Adib, and Dina Katabi. 2016. Emotion Recognition Using Wireless Signals. In *Proceedings of ACM MobiCom*.

[84] Mingmin Zhao, Tianhong Li, Mohammad Abu Alsheikh, Yonglong Tian, Hang Zhao, Antonio Torralba, and Dina Katabi. 2018. Through-Wall Human Pose Estimation Using Radio Signals. In *Proceedings of IEEE/CVF CVPR*.

[85] Mingmin Zhao, Yingcheng Liu, Aniruddh Raghu, Hang Zhao, Tianhong Li, Antonio Torralba, and Dina Katabi. 2019. Through-Wall Human Mesh Recovery Using Radio Signals. In *Proceedings of IEEE/CVF ICCV*.

[86] Mingmin Zhao, Yonglong Tian, Hang Zhao, Mohammad Abu Alsheikh, Tianhong Li, Rumen Hristov, Zachary Kabelac, Dina Katabi, and Antonio Torralba. 2018. RF-Based 3D Skeletons. In *Proceedings of ACM SIGCOMM*.

[87] Mingmin Zhao, Shichao Yue, Dina Katabi, Tommi S. Jaakkola, and Matt T. Bianchi. 2017. Learning Sleep Stages from Radio Signals: A Conditional Adversarial Architecture. In *Proceedings of ACM ICML*.

[88] Tianyue Zheng, Zhe Chen, Shuya Ding, and Jun Luo. 2021. Enhancing RF sensing with deep learning: A layered approach. *IEEE Communications Magazine* 59, 2 (2021), 70–76.

[89] Tianyue Zheng, Zhe Chen, Shujie Zhang, Chao Cai, and Jun Luo. 2021. MoRe-Fi: Motion-robust and Fine-grained Respiration Monitoring via Deep-Learning UWB Radar. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 111–124.

[90] Yue Zheng, Yi Zhang, Kun Qian, Guidong Zhang, Yunhao Liu, Chenshu Wu, and Zheng Yang. 2019. Zero-effort cross-domain gesture recognition with Wi-Fi. In *Proceedings of ACM Mobisys*.

[91] Han Zou, Yuxun Zhou, Jianfei Yang, Weixi Gu, L. Xie, and C. Spanos. 2018. WiFi-Based Human Identification via Convex Tensor Shapelet Learning. In *Proceedings of AAAI*.

A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches

Mariano Scazzariello[1,2], Tommaso Caiazzi[1,2], Hamid Ghasemirahni[1], Tom Barbette[3], Dejan Kostić[1], and Marco Chiesa[1]

[1]KTH Royal Institute of Technology
[2]Roma Tre University
[3]UCLouvain

Abstract

High-speed ASIC switches hold great promise for offloading complex packet processing pipelines directly in the high-speed data-plane. Yet, a large variety of today's packet processing pipelines, including stateful network functions and packet schedulers, require *storing* some (or all the) packets for short amount of times in a programmatic manner. Such a programmable buffer feature is missing on today's high-speed ASIC switches.

In this work, we present RIBOSOME, a system that extends programmable switches with external memory (to store packets) and external general-purpose packet processing devices such as CPUs or FPGAs (to perform stateful operations). As today's packet processing devices are bottlenecked by their network interface speeds, RIBOSOME carefully transmits only the relevant bits to these devices. RIBOSOME leverages spare bandwidth from any directly connected servers to store the incoming payloads through RDMA. Our evaluation shows that RIBOSOME can process 300G of traffic through a stateful packet processing pipeline (*e.g.*, firewall, load balancer, packet scheduler) by running the pipeline logic on a *single* server equipped with one 100G interface.

1 Introduction

Network Function Virtualization is an essential architectural paradigm of today's networks [32]. Operators create and manage complex packet processing pipelines by combining together Network Functions (NFs), which are then deployed on the infrastructure. Network functions that require simple computations are generally deployed onto *cost-effective* ASIC-based switches, whereas more complex packet processing computations must be deployed on *expensive* general-purpose CPUs or FPGAs due to the inherent difficulty and cost of designing complex ASIC circuits [4]. Unfortunately, the networking stack of general-purpose servers and FPGAs is significantly slower in processing packets than dedicated ASIC hardware counterparts, ultimately increasing the energy footprint and cost of operating a large network.

Deploying network functions that have to manage large amounts of frequently changing *stateful per-flow* information in a cost-effective manner (*i.e.*, entirely on an ASIC switch) has been an elusive goal.

To understand the requirements posed by multi-terabit per second stateful packet processing, we analyze a set of real-world CAIDA traces in the 2013–2019 period [6]. Through a linear regression, we observe that *i)* the number of active flow connections traversing a switch is 120 K for every gigabit of forwarded traffic and *ii)* there are 4 K new flow connections for every gigabit of forwarded traffic. This translates to 385 M active flows and 12.8 M new flow-table insertions per second on a 3.2 Tbps forwarding pipe. With a 17 B flow-state entry (*i.e.*, a 5-tuple + action), as in a Layer-4 load balancer, the memory requirement becomes 6.5 GB, which go beyond the stateful memory that is available on today's ASIC chips.

In this work, we aim at designing a stateful per-flow packet processor system that satisfies the following requirements:

- **Expressiveness**, by supporting a variety of complex stateful logic (*e.g.*, load balancers, packet schedulers).
- **High Throughput**, by achieving superior performance compared to existing expressive designs.
- **Dynamicity**, by supporting very frequent modifications to its stateful data structures.
- **Cost Effectiveness**, by reducing the costs and power consumption for operating this system.

Building a system that supports the above requirements is highly challenging. The expressiveness requirement requires a system to rely (at least to some degree) on general-purpose CPUs or FPGAs. We distinguish between two types of systems that require external resources:

- Systems that use **dedicated** external devices to realize complex NFs. An example within the first category is Tiara [46], a clever load balancer system that reroutes packets from a switch to 16 ports that are connected to FPGAs performing per-packet load balancer calculations. While such solutions are expressive, they are not cost-effective. Half of the devices connected to a 32-port programmable switch are used exclusively to perform stateful packet processing operations.

This reasoning motivates the next type of approaches.

- Systems that rely on **shared** external devices whose resources are primarily used for running customers' applications. Such systems embrace emerging disaggregation paradigms in which applications runs on resources that are combined together on demand. As an example, TEA [21] is the first system to efficiently enlarge the memory of the switch by cleverly crafting RDMA messages to access remote memory on shared CPU-based servers. TEA exploits the well-known large amount of spare of bandwidth and memory resources in datacenters [23]. This design allows operators to make better utilization of the resources available on an external general-purpose server: customers' applications run on general-purpose servers and any spare bandwidth and memory resources are used by the switch to store all the per-flow connection states that are required to process terabits of traffic. Unfortunately, as we show in our motivation section, TEA cannot support expressive network functions, as only the state is stored on external servers while the forwarding rules are applied on the ASIC switch, which does not support advanced logic.

In this work, we present RIBOSOME, a system that is expressive, flexible, cost effective, and achieves high-throughput packets processing. RIBOSOME relies on two fundamental observations. First, in a large fraction of stateful per-flow network functions the bottleneck is the bandwidth between the switch and the packet processor. For example, a load balancer saturates a 100Gbps interface with just 3 CPU cores on a real-world trace [12]. Second, many network functions do not need to analyze the entire packet but only the relatively small portion of a packet containing its headers.[1] To put things into perspective, a load balancer that operates on a 5-tuple (13 bytes) field, would only require receiving 13 B per packet instead of potentially 1.5 KB full size packets.

RIBOSOME relies on dedicated external packet processors to process packet headers while storing packet payloads on shared general-purpose servers without any CPU intervention (*i.e.*, using RDMA technology). More specifically, we leverage the advanced capabilities of emerging high-speed programmable switches to receive packets, split them into headers and payloads, and reconstruct them after the NF processor has updated their headers or re-schedule their transmission. By only processing packet headers, we overcome the bandwidth bottleneck at the dedicated devices, which allows us to process significantly higher numbers of packets on the same dedicated machine. As all data structures are handled by CPUs, we support high numbers of modifications to these data structures.

We motivate this design approach with the following observation: storing & fetching payloads are two operations that only require simple support for writing & reading on a memory. These memory operations are general, making it attractive to offload the storage & fetching of payloads on shared memory resources (*e.g.*, RDMA). Using shared resources to store payloads allow operators to make more efficient utilization of the memory resources existing in a network (such as a datacenter). We then rely on dedicated resources for processing packet headers. In this case, the rationale is that the performance achievable by a stateful network functions highly depends on temporal and spatial factors (*e.g.*, high cache-locality), and is therefore less suitable to be executed on shared resources. Finally, RIBOSOME brings benefits when an NF only needs to inspect a small part of a packet, *e.g.*, a load balancer. RIBOSOME does not bring benefits when the NF requires access to the entire packet (*e.g.*, a deep packet inspection function).

We implement RIBOSOME on an ASIC programmable switch, with FastClick [1] as the NF packet processor, and RDMA to store payloads on other servers. We evaluate RIBOSOME using an empirically-derived multi-100G traffic trace. Our micro-benchmarks show that a general-purpose server processes 70 Mpps on a single server, which would correspond to 560 Gbps of traffic with 1KB average packet size. Based on this estimate, we observe that one would need only three 100G ports on the programmable switch to process 1.6 Tbps of traffic (whereas systems like Tiara would need 16 ports).

We also evaluate the entire system using 4 RDMA servers and a single 100-Gbps dedicated server to process 300 Gbps of traffic. Our results show that the bandwidth requirements at the dedicated server are merely 20 Gbps.

To summarize, our contributions are:

- We propose a new disaggregation-based architecture to circumvent the inherent constraints of high-speed ASIC switches both in terms of logic and memory. We design a system to perform stateful packet processing that carefully splits operations between dedicated and shared resources, where headers are processed by dedicated servers while payloads are stored on shared resources.
- We present the first programmable buffer abstraction that is suitable for Tbps NF packet processing.
- We make the observation that today's deployment of NFs onto general-purpose CPUs is severely bottlenecked by the server bandwidth, thus motivating the splitting of the packets into headers and payloads.
- We demonstrate a single server processes up to 70 M small-size packets per second and the bottleneck moves onto the PCIe. With 3 servers, one could process 210 M packets per second, which is equivalent to roughly 1.6 Tbps of 1KB-size packets. We discuss future optimizations to overcome this limit with future-available hardware.
- We demonstrate in a small testbed that RIBOSOME processes 300 Gbps using a single dedicated NF processor, whose bandwidth requirement is just 20 Gbps.
- We demonstrate a 2.2× speedup over the state of the art for

[1] We do not claim novelty for this observation but rather for the novel trade-off achieved by the design of our packet processing architecture and our fine-grained evaluation.

running complex packet schedulers [11], on similar server hardware. By doing so, we show that we have fundamentally pushed the performance barrier achievable with a combination of a programmable switch and commodity servers.
- We release all our P4 and FastClick code for running RIBOSOME including a high-speed implementation of RDMA on the Tofino programmable switch [40].

2 Motivation

We start this section by quantifying the memory and flow-table update requirements of general ASIC switches. Based on an analysis of real-world traffic traces, we posit that today's (but also next-generation) ASIC switches do not have enough memory to support stateful per-flow NFs. We then quantify and discuss the limitations of the state-of-the-art systems using dedicated resources (*i.e.*, PayloadPark [13] and Tiara) as well as shared resources (*i.e.*, TEA [21]).

ASIC switches have constrained memory. Several existing approaches, such as SilkRoad [31], Cheetah [3] and SwiSh [47], propose to store the entire state required to operate a specific NF entirely on the memory available on the chip of an ASIC switch. However, the amount of memory available to store per-flow state on existing high-speed ASIC chips is a renowned constrained resource that may not be sufficient for NF applications that handle very large amounts of flows. We run some back-of-the-envelope calculations to upper bound the amount of state that could potentially be stored on an ASIC using SRAM technology. Assuming one could use the entire chip area for SRAM memory (*i.e.*, no I/O, no buffers), an 826 mm^2 chip using 7nm technology would only store at most 5GB of flow state in SRAM [7]. We show in this section that this amount of memory would suffice to only store ~10% of the state required on a multi-terabit per second switch. In practice, the amount of memory is below this estimate as typically I/O and buffers occupy roughly 50% of the chip area and some memory is used to implement the packet processing logic. For example, a 16-nanometer high-speed ASIC switch contains 1.5-15.4MB per terabit of forwardable traffic (*i.e.*, 10-100MB of SRAM on a 6.5 Tbps ASIC chip) [14, 29, 31].

To understand the implications of potential future trends on the feasibility of storing per-flow state on ASIC chips, we analyze CAIDA traces in the 2013 – 2019 period for the NYC and CHI monitored links, for which there are publicly available statistics [6].[2] The throughput for these traces ranges from 2 Gbps to 6.5 Gbps. Each trace contains the number of IPv4 and IPv6 forwarded packets, their mean packet size, the trace duration and the flows per second. The flow per second field represents the number of distinct flows in the trace divided by the duration of the trace.[3] We define a flow to be *active* if that flow has sent a packet in the last 30 seconds.[4] If a flow is active, it means the stateful processor must keep state for that flow. In the following analysis, we assume that one needs to deploy a basic load balancer that stores 17 B for each single flow (*e.g.*, 13 B for the 5-tuple and 4 B to store the private IP destination address). We define the *mean number of active flows* in a trace as the number of active flows divided by the duration in seconds of the trace multiplied by 30 (*i.e.*, the threshold to determine if a flow is active). In Fig. 1, we report the mean number of active flows (y-axis in millions, blue crosses) with respect to the trace throughput (x-axis) for each single trace in the studied period. We first observe that 3 out of 53 traces already require more memory resources than those available on a real-world pipe of a high-speed ASIC, *i.e.*, the traces require above 20 MB of memory as they contains more than 1.1 M active flows.[5] This means that storing state for a real-world trace recorded at roughly 10 Gbps would require roughly 20% of the existing memory on a 16-nm ASIC switch chip (which is in the 10-100 MB range).

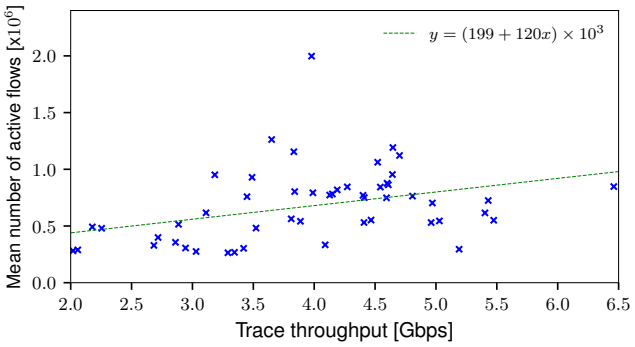


Figure 1: Number of active flows in historical CAIDA traces from 2013 – 2019 from Chicago and NYC.

We also plot a linear regression (green dashed line) that we use to estimate the memory requirements for a switch transporting terabits of traffic (such as recent 25.6 Tbps switches [5, 18]).[6] The steepness of the regression line is 120 K active flows per Gbps of traffic. We estimate the mean number of active flows on a 25.6 Tbps switch to be 3 072 M, which would require 52 GB of memory to store the corresponding state for a load balancer (*i.e.*, 17B per flow). This memory requirement is 300× higher than what is available on 7nm high-speed ASIC switches today [18] and roughly 10x the maximum amount of SRAM memory realizable on a

[2] We select this type of traces as we do not have access to datacenter traces at per-packet granularity (i.e., no sampling).

[3] We verified it by taking the Caida-nyc 2018-03-15 trace, counting the number of flows, and dividing by the trace duration, obtaining the same number.

[4] We use a 30-second threshold based on real-world timeouts used in the Facebook Katran load balancer [9].

[5] This is a lower bound since we take the mean of the active flows but peaks with higher number of active flows are likely to arise in the traces.

[6] The assumption may not be perfectly accurate but we do not have access to a datacenter trace at terabits per second speed and *per-packet* granularity (*i.e.*, no sampling).

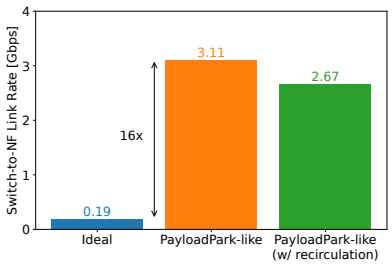


Figure 2: Ideal vs PayloadPark-like on CAIDA trace.

$825mm^2$ ASIC chip. Finally, we also note the memory on the pipe should be used for implementing other functionalities.

ASIC switches cannot support frequent per-flow state insertions from the control plane. Today's ASIC chips support a specific amount of modifications to their flow-tables through the control plane [3,31]. For example, a 16-nm 3.2 Tbps ASIC switch supports ~100 K flow-table entry modifications per second only [46]. To the best of our knowledge, this is the highest (publicly available) table update frequency achieved by a 16-nm ASIC switch through the control plane and we are not aware of public measurement studies showing such high-frequency insertions entirely in the faster data-plane of a multi-Tbps ASIC switch. Based on our analysis of the aforementioned CAIDA packet trace, we observe that the number of rule modifications is lower bounded by the amount of flows per second. Based on a similar regression, we obtain that the number of flow insertions per second grows by 4 K per Gbps of traffic. When we use this linear regression to estimate the amount of flow insertions on a 3.2Tbps ASIC switch, we obtain 12.8 M per-flow insertions per second, which is roughly 100× higher than the aforementioned 100 K flow-table modifications using an ASIC switch.

Now that we have quantified the amount of memory resources required to store the per-flow state of an NF, we discuss advantages and limitations of the three main state-of-the-art approaches to implement NF processors on top of programmable network hardware. We focus on the work that closely relates to our system and we defer the reader to Sect. 8 for a broad discussion of the existing related work.

NIC-based approaches that split packets are bottlenecked by the NIC speed. The *nicmem* system [35] is an NF accelerator that resides completely on a general-purpose server and does not involve any programmable switch. A packet arriving at the Network Interface Card (NIC) of a nicmem-equipped server is split into a header that is sent to the CPU cores and a payload that is stored on the small NIC memory. This approach comes with several benefits: higher hit cache ratio as payloads do not pollute CPU caches and 80% higher packet processing throughput. The inherent limitation of this work is that the throughput of the NF server is limited by the NIC speed (*e.g.*, a 100 G NIC can only process 10 M packets with size 1.25 KB). To process 800 Gbps of traffic, nicmem must connect to 8×100G ports on the switch. In this paper, we argue that an NF server should only receive the relevant bits (*e.g.*, the packet headers). This approach reduces bandwidth overheads toward external NF servers, reducing the number of ports on the switch that must be connected to dedicated resources. The remaining ports can be used for connecting the switch to other shared resources (where the payloads could be buffered) or devices. Moreover, traditional server- or NIC-based approaches (including nicmem) force the storage of the payload to be performed on the same machine that processes the header. Conversely, RIBOSOME decouples these two operations and it leverages spare memory resources in the network for performing the simpler payload storage.

Storing payloads at the switch does not mitigate bandwidth overheads. The *PayloadPark* [13] system also splits headers from payload but performs this operation directly on a programmable switch instead of the general-purpose server. The benefits of splitting a packet at the programmable switch instead of doing that at the server (as in nicmem) is that one can forward just the headers to the servers and store the payloads on the programmable switch. To implement a load balancer, a server could receive just 13 B from each packet (*i.e.*, the 5-tuple), thus potentially processing almost one billion packet headers per second through a 100 G interface. Unfortunately, PayloadPark suffers from an inherent constraint of high-speed ASIC devices. First, it is not possible to store the entire payload of a packet into the switch memory in a programmable manner.[7] Consequently, PayloadPark only stores the first 160 (352) bytes of payload for each packet without (with) recirculating it, thus letting most of the payloads still going to the external server.

To quantify the reduced performance gains, we have analyzed again the aforementioned CAIDA trace for three scenarios: *i)* when only a 54-byte header is sent to an external NF server, which we call *Ideal*, *ii)* when only 160 bytes of payloads are removed from the packet sent to the external NF server, which we call *PayloadPark-like*, and *iii)* a PayloadPark-like system that recirculates packets to store 352 bytes. Fig. 2 shows the link rate in Gbps between the switch and the external NF server (y-axis) for the three aforementioned systems. The ideal system requires 16× and 14× less bandwidth than the PayloadPark-like system with and without packet recirculation. We note that producing ASICs that would store larger parts of the packet in a programmatic manner would become significantly more complex and expensive [4] and are therefore neither available today nor in the near future.

Desiderata: large external memories shared with other applications. The goal of our work is to overcome the inherent limitations of ASIC switches and find an alternative design that keeps bandwidth requirements as close as possible to the ideal bar in Fig. 2. Since the memory on a switch cannot be used to store payloads in a programmatic manner, we fo-

[7] The payload is temporarily stored by the switch while the headers are being processed.

cus our attention to leveraging external *unused* resources that are shared with customers' applications. As storing payloads only requires storing information into memory (without any complex logic), we focus our attention onto RDMA technology to store and retrieve payloads between a programmable switch and a set of external servers that are deployed to run customers' applications. Leveraging RDMA from a Tofino switch is not a new idea per-se as it has been already explored in TEA [21], a network function accelerator, and Dart [24], a monitoring system. We now discuss existing limitations of TEA, which will motivate our design. We note that some limitations of TEA are due to its design while some others are related to the functionalities that are today available on ASIC switches.

TEA cannot run complex NF logic such as packet schedulers. TEA [21] is the first framework to implement NFs using a programmable switch and leveraging additional RDMA-accessible memory to store per-flow state. In TEA, a packet is forwarded from the switch to the RDMA server that stores the rule used to process the packet. Both the packet and the rule are forwarded back to the switch, where the rule is applied to the packet. Unfortunately, this design does not support more complex per-flow network functions (*e.g.*, advanced load balancers, batch-based NF processing, etc.) since the only logic that can be performed in TEA is the one that is supported by the switch. For example, TEA cannot support advanced per-flow packet scheduler such as Reframer [11], where packets arriving at an NF are buffered for a few tens of microseconds and are then reordered to increase their per-flow spatial locality (*i.e.*, placing packets belonging to the same flow close to each other). The reason why TEA cannot support such NFs is that TEA can only "buffer" a single packet while reading its rule but it cannot buffer arbitrary sets of packets in a programmatic manner. We are not aware of any existing ASIC switch supporting such programmable buffers for packets.

TEA cannot handle per-flow rule insertions at high speed. When a new packet of a flow arrives at the switch, TEA states that "since it takes some time to complete an insertion operation, new entries are first inserted in to an SRAM stash". However, the insertions into the Stash are performed through the control-plane, which is renown to take up to 1ms to perform insertions [31].[8] In any case, the limit of flow-table insertions per second derived in Tiara [46] also applies to any table modification on TEA, which severely undermines the ability of TEA to perform a large number of flow-table insertions per second.

In the remaining sections, we address the following question:

> *"Can we design an NF packet processor that retains the high-throughput of an ASIC switch while supporting dynamic per-flow stateful network functions in a cost-effective manner?"*

[8] We do not have access to the original P4 code of TEA.

3 System Design

We now present an overview of RIBOSOME, a NF accelerator for stateful per-flow packet processing that relies on a novel design to overcome the limitations of existing architectures based on programmable switches and external devices.

Design space. We first divide the design space into *i)* systems built entirely *within a switch* and *ii)* systems using *external devices*. In the first category, realizing stateful packet processing entirely using ASIC-based switches is out of reach because of both memory limitations and limited modifications per second to the stateful data structures. In the second category (*i.e.*, systems with external devices), we further divide into two categories: *a)* systems that only use external *dedicated resources* and *b)* systems that also rely on external *shared resources*. In the following, we discuss these two types of systems and we refer the reader to Table 1 for a summary of the architectural and communication overhead differences

The table covers three types of operations (*i.e.*, the processing of the header, the storage of the packet, and the splitting and merging of the packet with the header (if any)) as well as the communication overheads in terms of bits and number of packets transmitted to the NF and the shared servers for each incoming packet at the switch.

Delegating all stateful packet processing functionalities to *dedicated* external FPGAs or CPUs (*e.g.*, Tiara [46], nicmem [35]) results in a high utilization of the switch ports to interconnect the external dedicated devices (*i.e.*, to process 800 Gbps of traffic, 8x100G ports on a switch must be connected to dedicated devices). PayloadPark [13] reduces bandwidth requirements toward externally dedicated devices. However, it only saves 1280 bits of bandwidth per transmitted packet, which only slightly reduces the number of ports on the switch that are connected to dedicated devices when the average packet size of a trace is in the 1 KB range.

Leveraging *shared* resources mitigates these overheads as ports on a switch can be connected to devices running other types of computations. Some recent work (*e.g.*, TEA [21]) delegates the storage of payloads on shared memory while relying on the switch to run the stateful packet processing logic. However, the logic implementable on an ASIC switch is limited (*e.g.*, no batch-based stateful processing as in packet schedulers or rate limiters). Moreover, it is difficult to use CPU-bypass technologies like RDMA to insert per-flow state inside the external server memory because RDMA only supports basic primitives (*e.g.*, Read, Write) and cannot be easily used to perform insertions at high-frequency [37]. Striking the correct balance in the usage of dedicated and shared resources and the architectural choices is the main goal of this section.

Our design principles. In this work, we explore a trade-off in the design space between the usage of dedicated and shared resources to accelerate stateful packet processing. Our observation from Sect. 2 is that any stateful packet processing should support *i)* high-speed insertions into per-flow state data

	Operations			Communication overhead (per packet)	
	Header processing	**Payload store**	**Split & merge**	**NF server [bits, # pkts]**	**Shared server [bits, # pkts]**
Traditional	NF server	NF server	-	pkt.size, 1	-
nicmem [35]	NF server	NIC	NIC	pkt.size, 1	-
Tiara [46]	NF server (fast path on FPGA on NIC)	FPGA and server CPU	-	pkt.size, 1	-
PayloadPark [13]	NF server	switch	switch	pkt.size - 1280 b, 1	-
TEA [21]	switch	RDMA server	-	-	pkt.size, 2
RIBOSOME	NF server	RDMA server	switch	pkt.header, 1	pkt.payload, 2

Table 1: Qualitative comparison among existing systems in the design space and RIBOSOME.

structures (in the order of tens of millions per second) and *ii)* more complex stateful logic (*e.g.*, batch-based processing) when deployed on a multi terabits per second switch. Our design is inspired by the following principles:

- **Offload complex logic to dedicated devices.** As ASIC switches support a limited number of flow-table updates per second and provide limited memory space, we argue that non-trivial network functions, whether for inserting high volumes of per-flow entries into the per-flow data structures or processing packets in a batch (*e.g.*, for scheduling), should be realized on dedicated general-purpose servers.
- **Process only relevant bits.** Our design targets network functions (*e.g.*, load balancers, NATs, rate limiters, packet schedulers) that do not require inspecting the entire packet, but rather just a few bytes such as a flow identifier. We therefore propose to only send the relevant bits to the dedicated general-purpose servers and store the payloads on shared servers while the headers are being processed. Splitting headers is not a new idea per-se (see [13, 35]), however we leverage it in such a way that the large gains materialize in practice, as shown in our evaluation section. Notice that our design also provides the possibility to disable the packet splitting for specific traffic classes. This allows the coexistence between RIBOSOME and NFs that require fully inspecting packets.
- **A programmable buffer on shared resources.** ASIC switches (including programmable ones) do not provide an interface for buffering packets in a programmatic manner. Packets are stored either while their headers are processed through the pipeline or in port queues. We argue that a network function system should be able to buffer packets in a programmatic manner, operate on batches of packets and schedule their transmission (to a certain degree of granularity, see Sect. 4). We rely on RDMA to bypass CPU and avoid wasting CPU cycles on shared machines. Note that our approach does not rule out the possibility of accessing other types of memory for storing payloads. We embrace disaggregation paradigms where the storage of payloads is performed on any shared memory resources in the network. As an example, switches could potentially support a programmable interface to store and fetch packets in an internal DRAM or HBM.

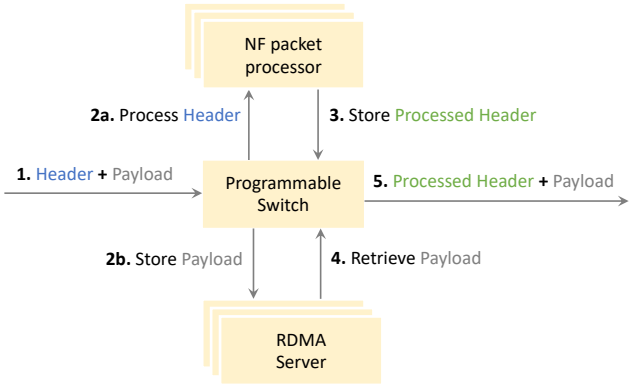


Figure 3: RIBOSOME overview.

To summarize, the main benefits of RIBOSOME are that it relies on dedicated devices only for realizing the NF processing logic and delegates the storage of the payload on external RDMA servers. RIBOSOME does not use any CPU cores on these RDMA servers. It only shares memory and NIC bandwidth with applications running on these servers. The benefits of RIBOSOME come with a cost: doubling the number of packets in a network since each packet will be split into a header and a payload packet.

System overview. RIBOSOME consists of a high-speed programmable switch, a set of *dedicated* external NF packet processors (*e.g.*, CPUs, FPGAs) and a set of *shared* servers. We leverage recent advancements in high-speed ASIC programmable switches [19], CPU-bypass memory storage (*i.e.*, RDMA [17]), and NF-specific CPU compiler optimiza-

tions [10] to design a system where dedicated packet processors only process the *relevant* portions of a packet while their payloads are stored on RDMA servers. We show a diagram of the high-level RIBOSOME architecture in Fig. 3. The programmable switch receives incoming packets (step 1) and splits each packet whose size is above a predefined threshold into a small header and a larger payload chunks. The programmable switch assigns an ID to both the header and the payload chunks. The switch assigns increasing IDs to each received packet within a predefined range (in a modulo manner). The switch forwards the header of the packet to one of the external NF packet processors (step 2a) and the payload to one of the shared servers (chosen hashing the flow 5-tuple) using RDMA (step 2b). The NF packet processors store the per-flow state needed to process any incoming packets. The NF uses this state to transform each incoming header into a new *processed header*, which is sent back to the programmable switch where it is stored on its small memory using the header ID as an index into an array in the switch SRAM memory (step 3). After storing a packet header, the programmable switch retrieves the corresponding payload from the RDMA servers (step 4). The programmable switch *combines* then the payload with the stored header using the ID and outputs the transformed packet (step 5).

We now discuss the different relevant operations and components of RIBOSOME, focusing on the main design challenges and our proposed solutions.

3.1 Splitting and Merging Packets

Deciding *i)* how many bits of a packet should be sent to the NF processor, *ii)* when a packets should be split into a header and payload, and *iii)* how to store the headers before the payloads are recollected are all questions that affect the overall performance of the system.

Challenges. Splitting a packet into a header and payload bring several benefits: it reduces both bandwidth overheads and cache pollution on the dedicated resources. However, splitting a packet also comes with some overheads: when we split a packet, *i)* we need to process a higher number of packets on the switch and *ii)* we need to use the switch memory to store the headers before recombining them with their corresponding payloads. More specifically, a single incoming packet arriving at a switch requires two packet processing if the packet is not split (*i.e.*, forwarding the packet to the NF and forwarding the modified packet from the NF to the output port) whereas a packet that is split results in 4 packet processing operations (*i.e.*, forwarding the header to the NF, forwarding the payload to the RDMA server, forwarding the NF response to the RDMA server to retrieve the payload, forwarding the recombined payload on the output port). Moreover, RDMA comes with limits on the number of operations per second that it can perform, which means transmitting small payloads may overload the server NICs without bringing any meaningful performance improvement.

Our approach. We devise a mechanism in RIBOSOME that splits a packet based on a threshold. There are two key thresholds in RIBOSOME: one threshold to specify when a packet should be split and one threshold to specify how many bits should be sent to the NF. We split packets at 72 bytes, considering the minimum Ethernet frame size of 64 bytes plus 8 bytes of additional custom RIBOSOME headers. Notice that the split threshold is configurable depending on the use case.[9]

RIBOSOME does not need to store any information on the switch when the packet has been split. Our system stores an header received from the NF on an array in the switch SRAM memory. Every time we split a packet, the switch increases the array index by one (modulo size of the array). This information is carried over in the header and the payload. When the header comes back to the NF, it is stored in the switch memory, and it also issues an RDMA Read Request to retrieve the corresponding payload.

The headers are stored until either *i)* the payload comes back to the RDMA server or *ii)* a new packet header is stored at the same array index, *i.e.*, the index pointer looped over the array size. We micro-benchmark the RDMA Read Request time (see Appendix F), finding that the maximum latency is at 4μs (with a 4096B payload). At 1.6 Tbps with an average packet size of 1 KB, this means we only need to store less than 800 headers in the array without risking to overwrite any header and combining it with the wrong payload. We configure the array with a size of 2K entries, which is large enough to guarantee that when the payloads come back, the header has not been replaced by a different header. By storing only 72 B of the packet header, RIBOSOME requires less than 60 KB of SRAM to store all the headers.

Our approach brings a significant advantage compared to alternative switch-based approaches like PayloadPark [13]. RIBOSOME only stores headers while retrieving payloads using RDMA, while PayloadPark must store payloads while waiting for the NF to process the headers. We observe that RDMA has more deterministic and lower response time than arbitrary NF processors. For instance, Batchy [25] reports processing times ranging from 100s of μs to few milliseconds to process a packet even for NFs that only look at packet headers (no heavy intrusion detection systems). At 1.6 Tbps with an average packet size of 1 KB and a response time of 1 ms, PayloadPark would need to store 185 MB of payloads, almost 10× the available memory on a 3.2 Tbps pipe.

Avoiding RDMA memory collisions. Packet payloads are stored contiguously in the external memory and are retrieved with the information contained in custom headers, hence there is no chance that two packets close in time read the same memory chunk. RIBOSOME conceptually uses the RDMA

[9] Ideally, the switch would only extract the bits relevant for a specific NF and batch the bits extracted from different packets into a single packet that is sent to the NF. Unfortunately, creating such batches is not supported by today's ASIC switches. We leave such optimizations as future work.

memory as a circular ring buffer, with a size provisioned large enough to prevent collisions.

Exploiting spare bandwidth on the RDMA servers. RIBOSOME leverages shared servers as remote buffers and it is therefore critical to use only the spare bandwidth of the server links without affecting the hosted services. Thus, RIBOSOME includes a control-plane mechanism that monitors the link bandwidth. When a link carries above a user-configured *back-off RDMA threshold*, the system stops sending payloads to the overloaded server.

Packets-per-second overhead. As RIBOSOME split packets, it also doubles the number of packets-per-second to be processed on both the switch (where we split the packets) and across the NF and RDMA-enabled devices when compared to a traditional approach. This is an inherent cost of splitting packets that is part of the RIBOSOME architecture. Table 2 compares per-packet processing overheads on each single component of three different NF approaches: *i)* Traditional NF processing where a switch sends the entire packet to an external NF processor, *ii)* a Payload-on-Switch (PoS) approach in which the payload is entirely stored on the switch (*e.g.*, PayloadPark), and *iii)* RIBOSOME. As it can be seen, RIBOSOME has the highest overhead. We first discuss the overhead on the switch. We note that several ASIC switches today support line-rate forwarding for small packets (*e.g.*, 300-Byte packets on general switches [18]). This means that RIBOSOME would support line-rate for packets with doubled size (*e.g.*, 600-Byte packets). This seems a reasonable trade-off in RIBOSOME as splitting a packet brings substantial benefits only when the packet is large-sized. As for the dedicated NF and shared RDMA-enabled servers, the number of PPS handled by the set of all these servers is twice as in a traditional approach. More specifically, the number of PPS handled by the NF servers is equal to that of the shared RDMA-enabled servers.

	Trad.	PoS	RIBOSOME
Switch	2/2	2/2	3/4
NF Server	1/1	1/1	1/1
RDMA Server	-	-	1/1

Table 2: Number of RX/TX packets for each approach (traditional, store payload on the switch, and RIBOSOME) in each component for each processed input packet.

3.2 High-Speed Reliable RDMA

To obtain a high-speed reliable RDMA implementation from the programmable switch, RIBOSOME has to overcome two main technical challenges. To support both RDMA Write and Read operations, Queue-Pairs (QPs) (virtual queues always composed of a *send* and a *receive* queue used to manage connections) must use the Reliable Connection transport mode. In this mode, the QP sends an acknowledgement for each packet received correctly, or a Nak in case of transmission problems. Detecting and recovering an RDMA Nak from a programmable switch is complex. For instance, receiving a *PSN Error Nak* would require transmitting the Nak-ed packet, and it is infeasible to store pending requests on the switch memory (especially RDMA Write operations that could contain a payload up to 4096 B). It is even more complex to recover from an *Invalid Request Nak*, triggered when the maximum number of outstanding RDMA Read Requests limit is reached. In fact, Infiniband specifications [42] limit the maximum number of outstanding Read Requests targeting a responder QP at any one time to a fixed amount (that is 16 with Nvidia Mellanox ConnectX-5 NICs [33]). To recover from this error, the QP state must be entirely reset.

We present a mechanism for recovering from the aforementioned failures with a minimal drop of packets in Appendix C. Moreover, we show in Sect. 5 how it is possible to handle multiple QPs, incrementing the maximum number of outstanding Read Requests, using a lower level API available within InfiniBand verbs [28].

4 Supporting Advanced Network Functions

Several common per-flow stateful operations require either batching a set of incoming packets (*e.g.*, [11, 25]), or keeping track of highly frequently arriving connections (*e.g.*, load balancers, NATs). In the following, we discuss three use cases for RIBOSOME that leverage advanced NFs whose logic would be difficult to realize on today's ASIC switches with large amounts of flows.

Stateful Load Balancers and NATs. Stateless load balancers suffer from high load imbalance [3] while software load balancers such as Maglev [8] must rely on many servers to remember which servers are taking care of every new selection. In RIBOSOME, we support stateful load balancers by storing the per-flow state on the NF processors and sending the headers of all packets to these processors. The main challenge is to support both high-throughput with millions of connections (which may not fit in the CPU caches) and forwarding rule insertions in the order of millions per second. In RIBOSOME, we use per-core Cuckoo++ [41] hash-tables, which have demonstrated superior performance [12]. The state management is using the recent FastClick's flow system, which finds a minimal per-flow state layout and handle state allocations and releases [2]. We support NATs similarly to load balancers.

Advanced per-packet software telemetry. Network telemetry is an indispensable component of today's networks, for both traffic optimization and security. Traditional monitoring tools such as NetFlow [16] rely on packet sampling and aggregation to perform off-path monitoring of the traffic, where "off-path" refers to the fact that network events are not detected on the data-path. Per-packet software monitoring has been proposed in *Flow [44], which however is also off-path,

thus cannot support advanced NFs. Emerging data-plane approaches such as ElasticSketch [45] detects network events directly in the ASIC data-plane using approximate data structures. However, ASIC switches have constrained memory for storing information about all flows. Through RIBOSOME, an operator leverages the large server memory to monitor in software and *on-path* all the packets processed by other advanced network functions, including load balancers and packets schedulers, which we discuss next.

Packet Schedulers. Packet schedulers are an example of network functions that determines the rate at which a flow or a traffic class should transmit traffic on a port or a destination server. Most importantly, they only need to inspect the packet headers and *buffer* packets for a limited amount of time (*e.g.*, on RDMA servers), which fits well within the RIBOSOME design. Realizing a per-flow packet scheduler on a hardware switch is hard for a number of reasons. Switches typically offer basic packet scheduler policies that scale to few traffic classes (up to 32 in general [43]). Realizing packet schedulers at the per-flow granularity or for more than 32 traffic classes is therefore hard to realize entirely in hardware.

RIBOSOME supports packet schedulers at the per-flow granularity, for instance, it support a per-flow leaky bucket rate limiter and the advanced Reframer [11] scheduler. RIBOSOME guarantees that packets belonging to the same traffic class leave the switch in the desired order. The main challenge in building a packet scheduler on RIBOSOME is that even if the NF processor reorders packets, there is no guarantee those packets will be output in the correct order from the RDMA servers. In fact, an RDMA Queue-Pair guarantees RDMA Read Request are served sequentially but Read Requests spread over different Queue-Pairs are not. Our key intuition is to guarantee that payloads of packets belonging to the same traffic class are read from the same RDMA Queue-Pair. The maximum throughput at which RIBOSOME guarantees an ordering of packets in a traffic class is limited by the hardware throughput of an RDMA Queue-Pair.

Example with a Reframer packet scheduler. We show an example of our approach implementing the Reframer packet scheduler [11]. Reframer is a NF that buffers packets for tens of microseconds and reorders them so that packets belonging to the same flow are transmitted back-to-back. We show that this functionality can be implemented on top of RIBOSOME.

Consider Fig. 4, where the RIBOSOME switch receives 8 packets from four ports belonging to four flows called *A* (black squares), *B* (yellow squares), *C* (blue squares), and *D* (green squares). Assume flows *A* and *B* are mapped to core 1 of the NF and they should be forwarded on port 1 of the switch while flows *C* and *D* are mapped to core 2 and they will end up on port 2. The switch is connected to one RDMA server which has 2 active Queue-Pairs; hence, all the payload data will be written on the same RDMA server. We use dashed orange (green solid) arrows to denote packets moving from the switch to external entities (from external entities to the

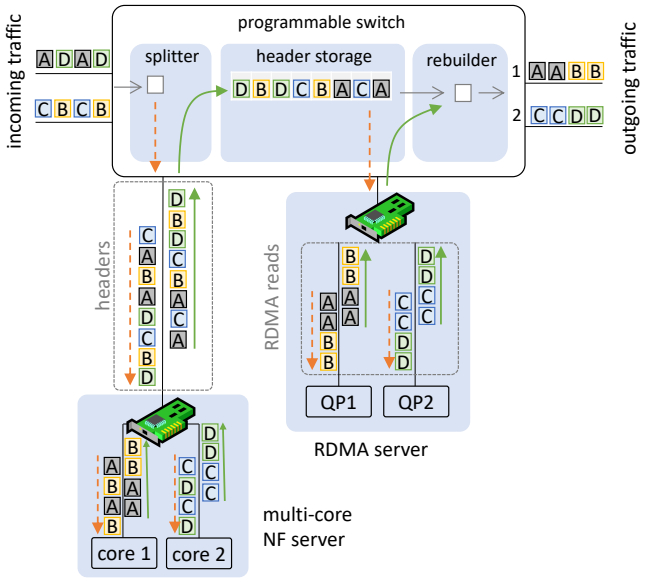


Figure 4: Supporting packet schedulers with RIBOSOME.

switch). The packets are initially split into a header that is forwarded to the NF server and a payload that is stored on the RDMA server (this RDMA Write operation is not shown in the figure). The headers will traverse the NF link in any extension of the partial order in which they arrived at the 4 ports. The switch tags headers with information about which RDMA server is used to store packets content (in this case, we have only one RDMA server and all packets have the same tag) [10]. On the NF side, core 1 (core 2) receives a sequence of packets $< B,A,B,A >$ ($< D,C,D,C >$) and reorders it into $< B,B,A,A >$ ($< D,D,C,C >$). To avoid Queue-Pairs overloading and to preserve packets ordering, the NF is enabled to add a new tag to packets determining the preferred Queue-Pair index for reading the content from the RDMA server. In this scenario, core 1 tags packets with QP1 and core 2 tags packets with QP2. Finally, the NIC forwards these headers back to the switch, possibly interleaving packets from the two cores. When packets arrive at the switch, the headers are stored in the order in which they are received and the corresponding RDMA Read Request are generated according to tags added by the NF cores. As a result, the RDMA server receives the Read Requests in two Queue-Pairs exactly in the order in which the corresponding NF cores have generated them. The payloads also return in the same order of the Read Requests.

5 Implementation

We implemented RIBOSOME's data plane in P4_16 language and compiled it to a Intel Tofino ASIC [19]. The server pro-

[10] We encode this tag in the MAC destination address which gives the flexibility to users to dispatch packets among cores based on corresponding RDMA servers if it is needed.

cess that manages RDMA connections and remote buffers is written in C++, using Infiniband Verbs [28].

Server Connection Establishment. For initializing a connection with the switch, the server agent takes as input the Infiniband interface name and the Server ID (an incremental index that starts from 0). It allocates a memory buffer enabled for remote write/read access. The process sends to the switch all the information to identify the connection using several different custom Ethernet frames (described in Appendix E).

The switch saves this information in different registers (`base_addr`, `rkey`, `mac`, and `ip`), using the Server ID as index.

After this initial setup, the process creates a user-defined number N_{qp} of Queue-Pairs. Each QP q_i ($i = 0, \ldots, N_{qp}$) is created using the Reliable Connection transport mode. A unique local Queue-Pair ID L_{q_i} is assigned by the NIC. Instead of pairing q_i with a remote QP, RIBOSOME fakes the connection to a remote endpoint, avoiding the need of a second Infiniband-capable device. A fake remote Queue-Pair ID R_{q_i} is computed using the formula $R_{q_i} = i + (Server_ID * N_{qp})$. The remote and local initial PSNs are set to 0. At this point, the server sends the Queue-Pair information to the switch with a Server Queue-Pair Info frame, containing L_{q_i} and R_{q_i}. The switch stores L_{q_i} in the `qp` register at index R_{q_i}, it writes the `enabled_qp` register at index R_{q_i}, enabling the QP, and it also sets the register `psn` to 0 at the same index R_{q_i}.

RIBOSOME Headers. RIBOSOME uses two custom headers, that contain information needed to retrieve a payload and merge it with its correct headers. We provide a description of the headers and their fields in Appendix E.

Splitter. When a packet is received on a RIBOSOME enabled port, the switch checks that the length of the packet is higher than a definable threshold. If it is under the threshold, the packet is not split and sent to the NF. Otherwise, the switch applies a hash function (on a 4-tuple composed of *SrcIP, DstIP, SrcPort, DstPort*) to index a Match-Action Table to select the server and the index i of the QP q_i that will store the relative payload. This ensures that payloads of the same flow will be managed by the same server and QP, and avoids reordering packets belonging to the same flow. The switch reads the `enabled_qp` register: if the QP is not enabled (equal to 0), the packet is sent to the NF without splitting it. Else, the switch retrieves (using i) the server data from `mac`, `ip`, `base_addr` and `rkey` registers. The packet is then transformed into a RoCEv2 RDMA Write. The `PSN` field of the BTH header is set by reading and incrementing the register `psn` at index i. The switch also appends the Header Info, selecting an index h where the header will be stored after being processed by the NF. It also appends the exact padding bytes to align the payload to a 4-byte boundary. The packet is mirrored to the Egress pipeline, where it is truncated to the header size. The switch appends both Payload Info and Header Info, that will be used for reconstructing the packet after the NF processing.

We show a high level overview of the flow in Appendix D.

Rebuilder. When the switch receives a processed header from the NF, if it contains the Payload Split header, it means that the packet must be reconstructed, else the packet is normally routed. Before reconstructing the packet, the switch saves the processed header into several registers `hdrs` at the index h specified by the Header IDX field in Header Info. The packet is then transformed into a RoCEv2 RDMA Read Request. The switch reads the Payload Info header and retrieves the index i to read the information of the server that contains the payload (`mac`, `ip`, and `rkey` registers). The `PSN` field of the BTH header is set by reading and incrementing the register `psn` at index i. The switch fills RETH header with the Payload Address and Payload Length fields stored in Payload Info.

The RDMA Read Request is sent to the corresponding server, which answers with an RDMA Read Response containing Header Info and the payload. The switch parses the response, reads h from the Header IDX field of Header Info and uses it to load the right header from registers `hdrs`. It removes the additional padding and prepends the processed header, reconstructing the entire packet that is normally routed. The reconstructed packet is 4 bytes longer than the original one as the switch cannot remove the ICRC appended by RoCEv2.

We show a high level overview of the flow in Appendix D.

Spare Bandwidth Exploitation. To ensure that RIBOSOME uses only the spare bandwidth of the shared servers without affecting hosted services, we implement a control plane mechanism that monitors the actual usage of the links. If the per-port bandwidth usage is under a configured *back-off RDMA threshold*, RIBOSOME uses the link for storing payloads in the remote memory of the server. Instead, if the port usage is above the threshold, the switch stops using that link for payloads, preserving the bandwidth for services. In this case, RIBOSOME remaps the Match-Action Table that selects QPs and servers, equally redistributing the entries of the overloaded server among the others. When the port bandwidth usage goes below the threshold, RIBOSOME restores the original mapping of the table, re-enabling the server.

6 Evaluation

RIBOSOME is the first programmable buffer abstraction that is suitable for Tbps advanced NF packet processing. It performs stateful packet processing, carefully splitting operations between dedicated and shared resources, dedicated servers process headers and servers hosting customers' services store payloads without CPU interference. In this section, we demonstrate the performance gains achievable by RIBOSOME. All scripts, including documentation for full reproducibility, are available [39]. We aim to answer five main questions:

- *"How much* RIBOSOME *improves the per-packet throughput and latency gain on the NF server?"*
- *"How does the packet size impact the throughput gains?"*
- *"Can we build advanced NFs on top of* RIBOSOME*?"*
- *"What are the overheads on the RDMA servers?"*
- *"How many ASIC resources does* RIBOSOME *require?"*

RIBOSOME's data plane is deployed on a 64 × 100 Gbps Stordis BF6064X with Intel Tofino ASIC [19]. Four of its ports are connected to a 32 × 100 Gbps Edgecore Wedge100BF-32X with Intel Tofino ASIC. Four commodity servers run the server agent and are equipped with Intel®Xeon®Gold 6140 CPU @ 2.30GHz and Nvidia Mellanox ConnectX-5 NICs [33]. All CPUs are set at their nominal frequency. The testbed is wired with 100Gbps links. The experimental setup is depicted in Appendix A.

Workload generation. To generate different loads, we use an additional server equipped with Intel®Xeon®Gold 6140 CPU @ 2.30GHz, and Nvidia Mellanox ConnectX-5 NICs [33], connected to the 32-port switch. The switch multicasts the incoming traffic to three ports unless stated differently. We inject both synthetic and real-world traffic traces using FastClick [1]. Another server with the same hardware runs different NFs, also implemented in FastClick. All the experiments are repeated 3 times.

6.1 Throughput and Latency Gains

RIBOSOME enables multi-100G packet processing. Fig. 5 shows the throughput of the NF (in pps) and the output throughput of the system (in Gbps) for three systems: a baseline where the NF receives the entire packet (dashed-dotten orange line), a PayloadPark-like system that removes 160 bytes of a payload when transmitting a packet to the NF server (dashed light blue line), and RIBOSOME (solid dark blue line). The average packet size is 1KB. The x-axis is the packet rate injected by the traffic generator (in Mpps). The baseline rapidly saturates the available 100 Gbps link bandwidth, consequently capping the NF throughput to this rate. The PayloadPark-like performance shows that only a limited increase in throughput can be achieved as the switch can store just a small part of the payload. RIBOSOME achieves higher throughput by sending only the headers to the NF, showing the system can keep up the processing of the 300 Gbps input traffic. We note that only ~75 Gbps of payloads can be handled by the RDMA NICs in our testbed due to RDMA overheads. Therefore, the 300 Gbps of generated traffic is limited by the 4 RDMA servers. So in order to process 1.6 Tbps of traffic, RIBOSOME would need to be connected to 22 shared RDMA-enabled servers. The gains for this simple forwarding NF could potentially be even higher by deploying more RDMA servers, which we could not do in our testbed.

RIBOSOME improves latency. While one would expect delaying packets to recover the payload through RDMA takes time, the advantage of reducing the queue sizes and the transmission rate at the NF compensates. In Fig. 6, we show the median latency (y-axis) with respect to the input rate (x-axis). Even at a medium input rate, the latency of the baseline system (which does not split packets) increases, while the latency of RIBOSOME is kept constant, achieving a 4× gain. We also verified that tail latency follows a similar trend: the baseline

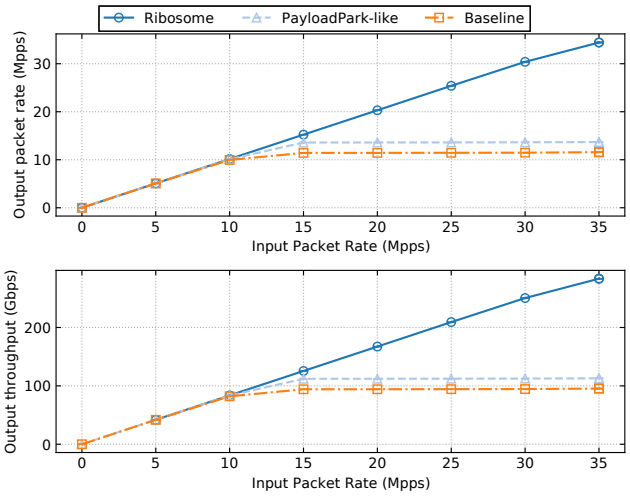


Figure 5: Bandwidth advantage of sending only headers to a forwarding NF, 1024 B packets.

reaches up to ~500 µs tail latency because of the queuing happening on the NF server, while Ribosome maintains a constant ~60 µs latency.[11]

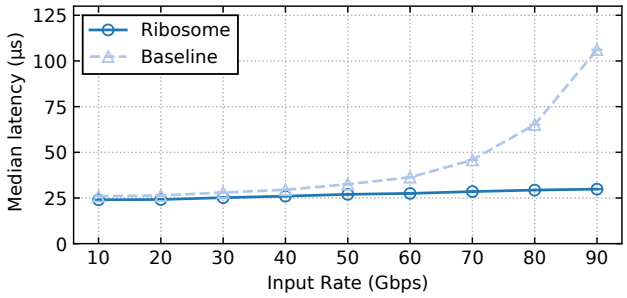


Figure 6: Median RTT latency for 1024 B packets sent by the generator to a forwarding NF under varying input rate.

RIBOSOME retains high performance regardless of the input packet size. Fig. 7 shows the output throughput (y-axis) of our baseline forwarding NF that does not split packets (dashed line) and RIBOSOME (blue line). The x-axis is the packet length in bytes. The split threshold is set to 64 bytes. Varying the packet size does not affect the overall throughput as the NF is still capable of processing 300 Gbps. Moreover, this also demonstrates that RIBOSOME is highly effective for the relevant real-world scenarios, where the average packet size ranges between 500 and 1K bytes [6]. We reach 300 Gbps of throughput already with 400 B packets (*i.e.*, 93Mpps). This graph demonstrates that the bottleneck with 1KB packets is not the CPU but rather the limited number of RDMA servers. We hypothesize that RIBOSOME could potentially operate

[11] Fig. 11 in Appendix B shows 99th percentile tail latency.

above 600Gbps (75Mpps) with a simple NF forwarder.

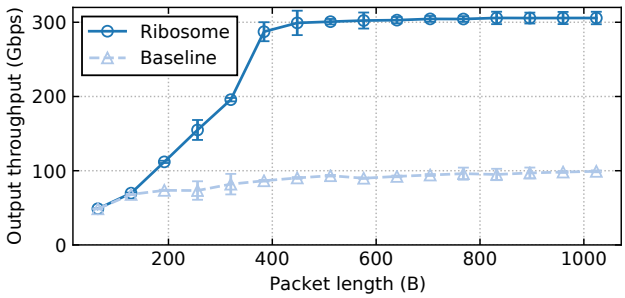


Figure 7: Bandwidth gain according to packet length.

6.2 Advanced Network Functions

We have successfully integrated and evaluated the three applications presented in Sect. 4 within RIBOSOME. Fig. 8 shows the RIBOSOME output throughput (in Gbps, y-axis) for the Reframer [11] advanced packet scheduler (blue solid line), a per-flow rate limiter (light blue dashed line), and a Layer-4 load balancer (orange dashed-dotted line). We vary the number of CPU cores used on the NF (x-axis). We replay a CAIDA trace that runs for 3.7s and contains 13.4 M flows.

RIBOSOME supports the Reframer packet scheduler at about 150 Gbps with 4 CPU cores and about 220 Gbps with 8 CPU cores (hyperthreading enabled). To put things into perspective, this level of throughput is almost 2.2× higher than the one achieved in the original paper on a single server (*i.e.*, 100 Gbps) [11]. We observe that a single Queue-Pair has a throughput limitations of 2.5 Gbps. This means that currently RIBOSOME, combined with Reframer, can only guarantee packet ordering for traffic classes whose throughput is at most 2.5 Gbps. We expect such limitations to ease in future generations of RDMA.

Similarly, Fig. 8 shows the performance of the per-flow rate limiter that independently tracks the rate of each individual micro-flow going through the switch and limits them using a per-flow token bucket. The rate-limiter NF achieves close to 300 Gbps similarly to the load balancer function. Both these NFs require keeping track of individual Layer-4 connections. In both cases, the NF server handles 3 M new flows per second, whereas an ASIC switch can only support ~100 K flow-table entry modifications [46].

RIBOSOME preserves the order of packets. As discussed in Sect. 4, there is a possibility that packets from different Queue-Pairs got interleaved during RDMA Reads. We investigate the amount of unordered packets by measuring the average Spatial Locality Factor (SLF) of the traffic on the RIBOSOME output and comparing it with the SLF value right after Reframer instances on the NF. It is the parameter used in [11] to measure the ordering level of the traffic. We observe that using

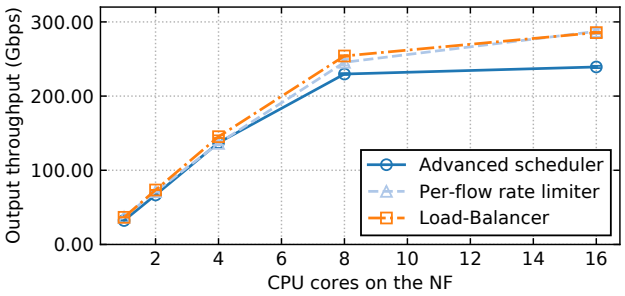


Figure 8: Throughput of three advanced Network Functions.

our trace file, the SLF value on the NF server is 1.31 while it only drops slightly to 1.29 on the output of RIBOSOME, which demonstrates its ability to primarily preserve the order of packets according to the advanced scheduler.

6.3 RDMA Interference Analysis

Storing payloads on RDMA servers may impact applications running on those servers, and specifically, their available network and memory bandwidth.

Little impact of RDMA on memory bandwidth and CPU. We generate RDMA operations that fill the NIC capacity (nearly 100 Gbps) and verify that the CPU load is 0% as RDMA traffic bypasses the CPU. We then use STREAM [30] to benchmark the CPU-to-memory bandwidth. We see that the available CPU-to-memory bandwidth decreases by ~25% when using 100% of the NIC for RDMA operations. Despite such bandwidth decrease, RIBOSOME leaves plenty of memory spare bandwidth on the RDMA servers.

RIBOSOME reactively releases network bandwidth resources from RDMA servers. We craft a synthetic trace where RIBOSOME does not split a specific traffic class and forwards it directly to the RDMA Server 1. This "unsplit" traffic gradually increases over time simulating an increased bandwidth demand on Server 1. We run this experiment without 3× multicasting enabled and we set the back-off RDMA threshold at 40 Gbps. Fig. 9 shows the input throughput of the "unsplit" traffic (violet line) and that of the four RDMA servers. We see that the RDMA throughput on each RDMA servers is around 7 Gbps at time 15s. When the "unsplit" traffic reaches roughly 33Gbps (at time 16s), this event triggers RIBOSOME's control mechanism, which stops sending payloads to Server 1 and redistributes the load on the other three servers. In future work, we will design an adaptive algorithm (instead of an on/off control mechanism) to share the NIC bandwidth in a fine-grained manner. This will also decrease the CPU memory controller utilization of RIBOSOME when the server is used for a network workload.

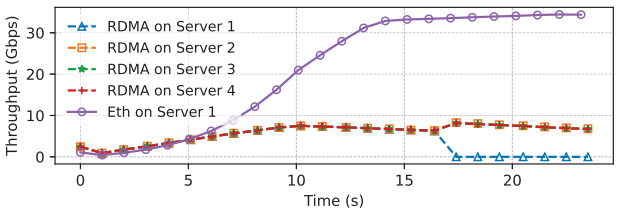


Figure 9: RIBOSOME stops sending payloads to Server 1 when it becomes overloaded.

6.4 ASIC Resource Usage

Table 3 shows the additional ASIC resources consumed by RIBOSOME based on the Tofino compiler's output. Overall, RIBOSOME consumes a similar amount of VLIW Instructions and Match Crossbar than TEA [21] and occupies a negligible amount of TCAM. For SRAM memory usage, we allocate $1.5K \times 15 \times 4B$ register entries to store headers while fetching the payloads with RDMA, which suffice to sustain the $4\mu s$ RDMA maximum response time. Moreover, RIBOSOME also relies on several registers to store RDMA connections data, which justifies the additional memory usage.

Resource	Additional Usage
SRAM	7.84%
TCAM	1.14%
VLIW Instruction	13.82%
Exact Match Crossbar	13.92%
Ternary Match Crossbar	0.69%

Table 3: Additional ASIC resources used by RIBOSOME.

7 Discussion

How many NF-dedicated ports to achieve full-throughput on a high-speed ASIC switch? Similarly to Tiara, RIBOSOME only achieves half of the switch throughput as half of the ports store payloads. One RIBOSOME server processes up to 80 Mpps for a forwarding NF. We need to use three ports of the switch exclusively for NF processing to process the equivalent of 1.6 Tbps of 1 KB packets. With advanced NFs, we must reserve 6-7 ports on the switch whereas Tiara requires 8 ports to connect its FPGAs. Replacing our NF servers with FPGAs may lower the number of dedicated ports to our lower bound of 3 ports, which is future work.

Can RIBOSOME offload heavy-hitter entries to the switch after an insertion? Yes, this is doable (whenever the NF function is realizable on a programmable switch) and similar to what TEA or CRAB [22] do. We believe this optimization is orthogonal to our approach and we leave it as future work.

8 Related Work

We discuss related work that we have not already mentioned.

Dedicated external devices. Several systems require sending the entire packet to the NF processor [8, 15, 20, 26, 34–36, 46, 49]. Gallium [48] enables offloading a part of the NF processing on the switch, but complex processing still needs to be executed on the NFs servers. In contrast, we minimize the amount of dedicated resources needed to run complex NFs by relying on shared resources to store and retrieve payloads, thus minimizing the number of ports on the switch connected to dedicated devices. Moreover, we present a novel programmable buffer that supports packet schedulers or batch-based NFs.

RDMA on a high-speed ASIC switch. TEA [21], SwitchML [38], and Dart [24] all propose to use RDMA directly on the Intel Tofino switch. TEA (SwitchML) implements reliable (unreliable) RDMA transport. TEA code is not publicly available. Unreliable RDMA does not support RDMA Reads. Dart sketches an implementation without providing code. We implement reliable RDMA, evaluate bottlenecks, and make all our code public.

9 Conclusion

We presented RIBOSOME, a high-speed stateful packet processor that reduces the amount of dedicated NF processors by carefully sending only headers to the external NFs. We showed in our testbed that RIBOSOME scales throughput by up to a factor of 3× with a single NF processor (potentially up to 10× with additional RDMA servers for storing payloads). We believe that RIBOSOME aligns with the current trends towards disaggregating architecture in which resources are shared for different purposes. We leave as future work the fundamental problem of further improving the performance of CPU-based NF processors given the observed high cost of retrieving the flow state from memory (especially when it resides outside the cache). Also, we will further investigate optimizations for reducing packet-per-second overheads introduced when splitting packets.

Acknowledgements

We would like to thank our shepherd Jeongkeun Lee, the anonymous reviewers for their insightful comments and suggestions on this paper. This work has been partially supported by the Swedish Research Council (agreement No. 2021-04212) and KTH Digital Futures. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889). Tom Barbette has been funded by an FSR Post-doc Fellowship from UCLouvain.

References

[1] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16, 2015.

[2] Tom Barbette, Cyril Soldani, and Laurent Mathy. Combined stateful classification and session splicing for high-speed NFV service chaining. *IEEE/ACM Transactions on Networking*, 29(6):2560–2573, 2021.

[3] Tom Barbette, Erfan Wu, Dejan Kostić, Gerald Q. Maguire, Panagiotis Papadimitratos, and Marco Chiesa. Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency. *IEEE/ACM Transactions on Networking*, pages 1–14, 2021.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[5] Broadcom. Tomahawk4, 2022. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series.

[6] CAIDA . Trace Statistics for CAIDA Passive OC48 and OC192 Traces, 2019. https://www.caida.org/catalog/datasets/trace_stats/.

[7] Don Draper. TSMC's 5nm 0.021um2 SRAM Cell Using EUV and High Mobility Channel with Write Assist at ISSCC2020, 2020. https://semiwiki.com/semiconductor-manufacturers/tsmc/283487-tsmcs-5nm-0-021um2-sram-cell-using-euv-and-high-mobility-channel-with-write-assist-at-isscc2020/.

[8] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 523–535, USA, 2016. USENIX Association.

[9] Facebook. Katran Load Balancer, 2021. https://github.com/facebookincubator/katran/blob/3fadb1eaaff719980a3cc9dc8870f88d442a40e1/katran/lib/bpf/balancer_consts.h#L100.

[10] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-Core 100-Gbps Networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Maguire, and Dejan Kostic. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2021.

[12] Massimo Girondi, Marco Chiesa, and Tom Barbette. High-speed Connection Tracking in Modern Servers. In *22nd IEEE International Conference on High Performance Switching and Routing, HPSR 2021, Paris, France, June 7-10, 2021*, pages 1–8. IEEE, 2021.

[13] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. *Parking Packet Payload with P4*, page 274–281. Association for Computing Machinery, New York, NY, USA, 2020.

[14] Vladimir Gurevich and Andy Fingerhut. P4_16 Programming for Intel Tofino using Intel P4 Studio, 2021. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf.

[15] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-Accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 195–206, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys Tutorials*, 16(4):2037–2064, 2014.

[17] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification - Annex A17: RoCEv2, 2014. https://cw.infinibandta.org/document/dl/7781.

[18] Intel. Intel Tofino 3 Intelligent Fabric Processor Brief, 2022. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html.

[19] Intel. Intel Tofino Series, 2022. https://www.intel.com/content/www/us/en/products/

network-io/programmable-ethernet-switch/tofino-series.html.

[20] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, Renton, WA, April 2018. USENIX Association.

[21] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 90–106, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the Load Balancer without Regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 193–207, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[24] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. *Zero-CPU Collection with Direct Telemetry Access*, page 108–115. Association for Computing Machinery, New York, NY, USA, 2021.

[25] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor Retvari. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 633–649, Santa Clara, CA, February 2020. USENIX Association.

[26] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Linux. perftest, 2022. https://github.com/linux-rdma/perftest.

[28] Linux. RDMA Core, 2022. https://github.com/linux-rdma/rdma-core.

[29] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3829–3846. USENIX Association, August 2021.

[30] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, 2016.

[33] NVIDIA Networking. NVIDIA Mellanox ConnectX-5 adapters, 2021. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/.

[34] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. *SIGCOMM Comput. Commun. Rev.*, 43(4):207–218, aug 2013.

[35] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The Benefits of General-Purpose On-NIC Memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, pages 1–18. Association for Computing Machinery, New York, NY, USA, February 2022.

[36] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano.

FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

[37] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association.

[38] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[39] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. Ribosome Experiments Github Repository, 2022. https://github.com/Ribosome-Packet-Processor/Ribosome-experiments.

[40] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. Ribosome Github Repository, 2022. https://github.com/Ribosome-Packet-Processor/Ribosome.

[41] Nicolas Le Scouarnec. Cuckoo++ hash tables: High-performance hash tables for networking applications. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 41–54, 2018.

[42] Tom Shanley. *Infiniband Network Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[43] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, April 2018. USENIX Association.

[44] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 823–835, Boston, MA, July 2018. USENIX Association.

[45] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 561–575, New York, NY, USA, 2018. Association for Computing Machinery.

[46] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, apr 2022. USENIX Association.

[47] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, Renton, WA, April 2022. USENIX Association.

[48] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 283–295, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020.

A Experimental Setup

Fig. 10 depicts the experimental setup. The Traffic Generator is connected to the Multicast Tofino with a 100-Gbps link. The Multicast Tofino has four of its 100-Gbps ports connected to the RIBOSOME Tofino, and it multiplexes the incoming traffic from the Generator. It modifies each packet to have two additional copies with different 4-tuple values before sending them to the RIBOSOME Tofino. RIBOSOME Tofino is attached to four commodity servers running the RDMA server, and one server that implements the NF, all with 100 Gbps links. When a packet is reconstructed after the NF processing, the RIBOSOME Tofino sends it back to the Multicast Tofino. If

the packet is an original one (with no flow modifications), it is sent back to the Traffic Generator.

B RIBOSOME Tail Latency Impacts

Fig.11 demonstrates the impact of RIBOSOME on packets 99th percentile tail latency when the NF server is running a simple forwarder.

C Recovering RDMA Queue-Pairs

We illustrate the implementation of the QP Recover mechanism discussed in Sect. 3.

For supporting both RDMA Write and Read operations, Queue-Pairs are created using the Reliable Connection transport mode. This mode expects that packets are received in the correct order, by checking their PSN. If a packet is out of order, the QP sends back a *PSN Error Nak*, requesting the retransmission. RIBOSOME's implementation does not store RDMA Write or Read Request packets in the switch (it only keeps the current PSN). Hence, it is not able to retransmit a Nak-ed packet. Additionally, Infiniband specifications limit the maximum number of outstanding RDMA Read Requests on each QP. If there are more of such requests, the QP transits into an invalid state, sending an *Invalid Request Nak*.

To overcome these limitations, RIBOSOME exploits several QPs on each server, and it also implements a QP recovery mechanism, that allows to reset a QP in case of errors.

When the programmable switch receives a Nak, it puts the corresponding Queue-Pair q_i in a disabled state writing the `enabled_qp` register to 0. The index i to access the register is the `DestQP` field of the BTH header, that is the value $R_{q_i} = i + (Server_ID * N_{qp})$. The `enabled_qp` register is periodically checked by a control plane script, that takes all the entries with a value of 0 and, for each index j, reads register `qp` at index j. If the Queue-Pair ID is set, the switch writes another register (called `qp_to_restore`) at index j with a value 1.

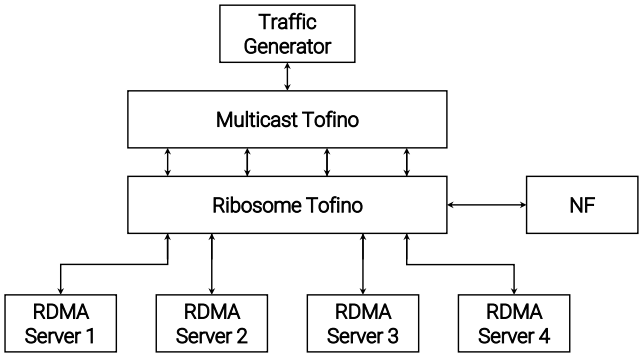


Figure 10: RIBOSOME testbed.

When receiving a packet in the Splitter component, if the Queue-Pair q_i is selected, and the value of `enabled_qp` register at index i is 0, and the value of `qp_to_restore` register at index i is 1, the packet is sent to the NF without splitting it. Also, the packet is mirrored to the Egress pipeline, and transformed into a simple Ethernet frame with `EtherType` = 0x4321, containing the index of the Queue-Pair to restore. The server agent has a raw L2 socket listening on the interface used to open the connection. When the aforementioned frame is received, the associated QP is reset and re-initialized. At this point, a Server Queue-Pair Info packet is sent to the switch, that re-enables the QP.

D Splitter and Rebuilder Components

We illustrate a high level overview of the two main RIBOSOME's components. Fig. 12 depicts the *Splitter* component, while Fig. 13 shows the *Rebuilder* component.

E Custom Ethernet Frames and Headers

RIBOSOME leverages on several custom Ethernet frames to identify a server connection, depicted in Fig. 14.

Also, the system uses two custom headers (showed in Fig. 15), that contain information needed to retrieve a payload and merge it with its correct headers:

a) **Payload Info**: appended to truncated headers when the packet is split. The Marker field is used by the switch to identify the header. The Payload Address indicates the starting address of the payload in the remote RDMA buffer. The Length field is the length of the payload in the buffer. The Index is used by the switch to request the payload on the same Queue-Pair used when sending the RDMA Write request.

b) **Header Info**: appended to the Payload Info when the packet is split. It is also prepended to the payload before sending it to the remote buffer. The Pad Count field stores the number of bytes of the Additional Padding field to align the payload to a 4-byte boundary as per Infiniband

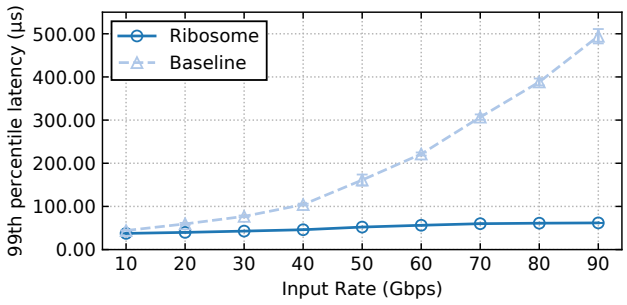


Figure 11: RIBOSOME 99th percentile latency of packets w/wo existence of Ribosome.

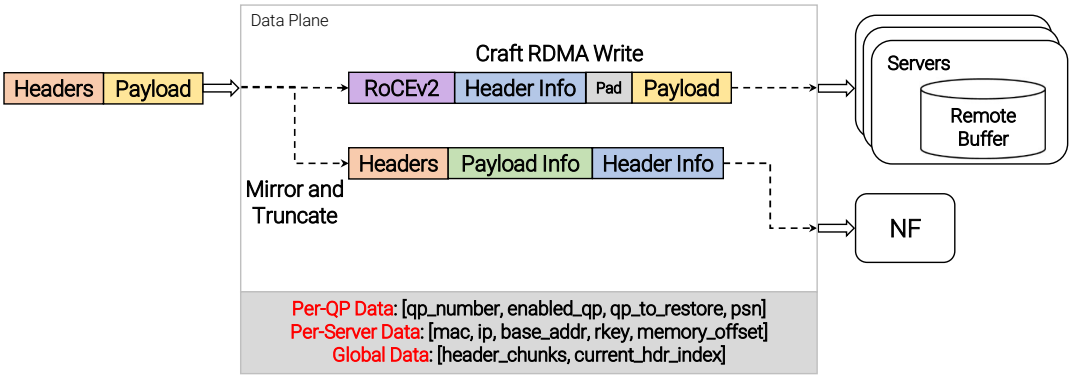


Figure 12: RIBOSOME's *Splitter* Component Overview.

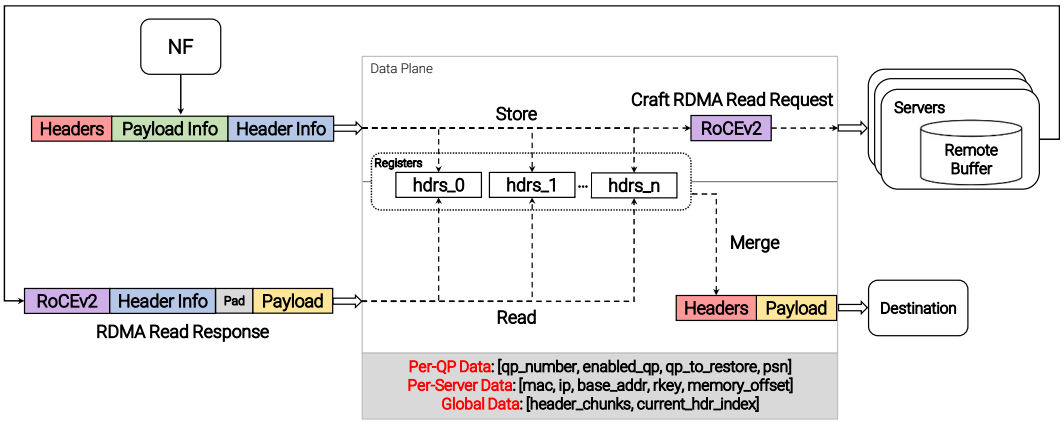


Figure 13: RIBOSOME's *Rebuilder* Component Overview.

specifications [42]. The Header IDX indicates the index of the register where the header processed by the NF is saved while the switch is fetching the payload from the remote buffer.

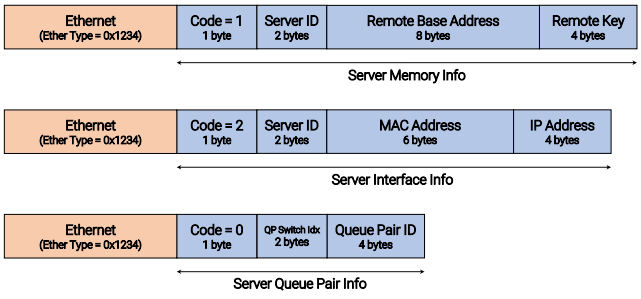


Figure 14: Server Info Ethernet Frames.

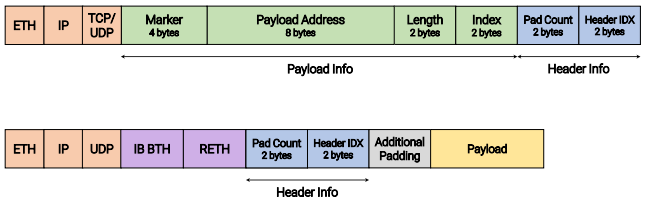


Figure 15: RIBOSOME Headers.

F RDMA Latency Microbenchmark

We performed a microbenchmark of RDMA operations using the Linux Infiniband `perftest` suite [27] on two servers, equipped with Intel®Xeon®Gold 6140 CPU @ 2.30GHz, and Nvidia Mellanox ConnectX-5 NICs. Fig. 16 shows the average latency of 1 K iterations (y-axis) of both RDMA Read and Write operations with different payload lengths (x-axis).

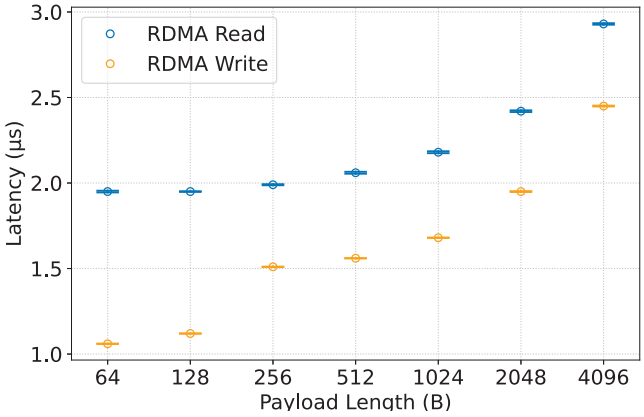


Figure 16: RDMA Operations Microbenchmark.

ExoPlane: An Operating System for On-Rack Switch Resource Augmentation

Daehyeok Kim[†‡] Vyas Sekar[§] Srinivasan Seshan[§]
[†]*Microsoft* [‡]*University of Texas at Austin* [§]*Carnegie Mellon University*

Abstract

The promise of in-network computing continues to be unrealized in realistic deployments (e.g., clouds and ISPs) as serving concurrent stateful applications on a programmable switch is challenging today due to limited switch's on-chip resources. In this paper, we argue that an on-rack switch resource augmentation architecture that augments a programmable switch with other programmable network hardware, such as smart NICs, on the same rack can be a pragmatic and incrementally scalable solution. To realize this vision, we design and implement ExoPlane, an operating system for on-rack switch resource augmentation to support multiple concurrent applications. In designing ExoPlane, we propose a practical runtime operating model and state abstraction to address challenges in managing application states correctly across multiple devices with minimal performance and resource overheads. Our evaluation with various P4 applications shows that ExoPlane can provide applications with low latency, scalable throughput, and fast failover while achieving these with small resource overheads and no or little modifications on applications.

1 Introduction

While recent efforts have demonstrated the feasibility of using programmable switches to implement network functions, such as NATs, firewalls, and load balancers (e.g., [4, 38, 46]) and to accelerate distributed systems (e.g., [43, 44, 52, 55]), there is still significant apprehension from practitioners whether in-network computing is ready for prime time. In many ways, this apprehension is justified as serving concurrent stateful applications in production-scale clouds and cellular networks is not possible today or in the foreseeable future. The fundamental issue is that due to limited on-chip resources (e.g., 10s MB of SRAM), these switches cannot keep up with the increasing number of stateful applications [33, 39] which operators want to run on a switch and the demand to handle heavier workloads in terms of traffic volume and flows [8, 26].

Instead of arguing for beefing up the switch ASICs or creating hyper-optimized applications, we explore a pragmatic alternative and make a case for *on-rack switch resource augmentation* architecture. We envision a deployment that consists of a programmable switch, other data plane devices (e.g., smart NICs [7, 11, 14, 18], and software switches running on servers [19, 56]) connected to the switch on the same rack. These external devices offer more resources to offload stateful packet processing, albeit with some performance penalty. Perhaps more significantly, they offer a path to affordably and incrementally scale the effective capacity of a programmable network to handle future workload demands.

To effectively realize this vision of on-rack switch resource augmentation, we argue that we need an *operating system* (OS) to manage resources spread across multiple on-rack devices. To borrow from Anderson and Dahlin [21], we can draw a first-principles analogy to the three roles that any OS serves: (1) a "glue" to provide a set of common services that facilitate the sharing of resources among applications; (2) an "illusionist" to provide an abstraction of physical hardware to simplify application design; and (3) a "referee" for managing resources shared between multiple applications. While there is some recent work on mapping a single switch application to heterogeneous devices or to augment memory (e.g., [30,40,42, 54]), these fundamentally do not tackle multiple concurrent applications or provide these capabilities.

However, realizing such components in our context is uniquely challenging because of hardware and workload characteristics. More specifically, we observe that managing states correctly while minimizing the performance and resource overhead is difficult, especially under high packet processing speed and dynamically changing workloads. In our setting, application states can be placed, and workloads (i.e., packets) can be executed on multiple devices. Thus, state management becomes critical for application correctness (e.g., accessing incorrect state), performance (e.g., high packet processing latency due to inter-device communications), and resource overhead (e.g., additional switch resources).

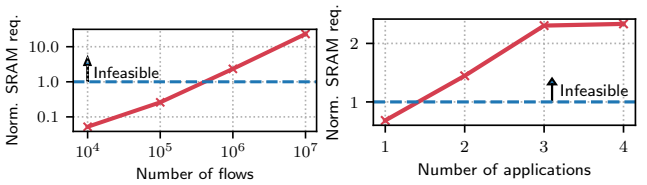
In designing ExoPlane,[1] an OS for switch resource augmentation, we address these challenges as follows:

- Runtime service (the glue): To avoid frequent inter-device communications during packet processing, we propose a *packet-pinning* operating model that guarantees that a packet is processed entirely on a single device.
- State abstraction (the illusionist): To enable correct stateful processing of packets even under dynamically changing workloads, we design a *two-phase state management* that places application states correctly on different devices as the workload changes. We also design appropriate levels of consistency for different types of stateful objects that appear in applications.

[1]The name denotes an external (exo-) data plane.

Applications	States
Per-tenant VPN gateway + Packet counter	Ext.-to-int. tunnel mapping and processed packet counter for each tenant.
Per-tenant NAT	Per-flow address mapping for each tenant. Per-flow address mapping for each tenant.
Per-tenant ACL + Filtered packet counter	Per-flow ACL and dropped packet counter for each tenant.
Sketch-based monitor	UnivMon [45] for remaining traffic classes.

Table 1: P4 applications deployed in a gateway switch of the data center in our motivating scenario.



(a) Four applications with varying numbers of flows (log-scaled). **(b)** Varying number of applications with 1 million flows.

Figure 1: SRAM requirements (normalized to the total amount of SRAM on a switch) with varying workload sizes and numbers of applications. If the requirement > 1, it is infeasible.

- Resource allocation (the referee): To achieve performance and policy goals specified by developers and operators, we formulate and solve an optimal resource allocation problem that accommodates heterogeneity across applications and data plane device capabilities.

ExoPlane consists of two key components: the planner and runtime environment. The planner takes multiple P4 [25] applications written for a switch with no or little modifications and optimally allocates resources to each application based on inputs from a network operator and developers. It requires developers to add application-specific logic using our APIs *only if* the program contains an object that can be updated in the data plane. Then, the ExoPlane runtime environment executes workloads across the switch and external devices by correctly managing state, balancing loads across devices, and handling device failures.

We implement the planner in C++, the data plane of the runtime environment in P4, and the control plane of the runtime environment in Python and C++. We evaluate it using various P4 programs in our testbed consisting of a Tofino-based programmable switch [9] and servers equipped with Netronome Agilio CX smart NICs [7]. Our evaluations show that ExoPlane achieves low latency (e.g., ≈300 ns at the switch and 5.5 µs at an external device in steady-state) and scalable throughput with more external devices (e.g., up to 394 Gbps, the maximum rate in our testbed). In case of an external device failure, ExoPlane can recover an end-to-end TCP throughput within 200 ms. ExoPlane achieves these with small control plane (a few tens MB) and switch ASIC resource overheads (less 4.5% of ASIC resources).

2 Motivation and related work

In this section, we motivate the need for supporting multiple concurrent stateful applications in the network, provide a primer on in-switch stateful applications, and discuss why prior work falls short.

2.1 Motivating example

We observe two key trends in in-network computing that increase demands on switch resources. First, the number of applications that need to run concurrently will likely increase [33, 39]. Second, the per-app workload size in terms of traffic volume and the number of flows keeps growing [8, 26].

As a concrete example, suppose a cloud or cellular operator wants to deploy four applications in Table 1 on the edge router (e.g., [47, 49]) processing traffic entering/leaving the network. Each application maintains per-flow states for each tenant to enable virtual private networks (*VPN gateway*), route traffic from tenants' on-premise networks to VMs running services (*NAT*), or control access to services running on tenants' VMs (*ACL*). The *sketch-based monitor* collects statistics for the remaining traffic classes using an UnivMon sketch [45]. To see if/how these applications can coexist, we implement them in P4 or use source code from the original authors, compose them into a single P4 program using our merger (described in §6) and compile the result using the Tofino P4 compiler.

Unfortunately, we find that enabling these applications concurrently in a switch is infeasible for typical workloads, which requires the support of at least 1M concurrent flows [28, 46, 47], as shown in Fig 1. We consider two scenarios: (a) running all four applications but varying numbers of concurrent flows per application and (b) fixing the number of flows to 1M but adding applications incrementally. Here, we use SRAM requirements from each application, normalized (due to vendor NDAs) to the total amount of SRAM on a switch, which is the bottleneck resource in our scenario. In Fig 1a, we see that as the workload increases, it becomes infeasible to run all the applications. Similarly, in Fig 1b, we see that the switch can support only a single application.[2]

2.2 Stateful switch applications

Before we discuss why prior work cannot tackle the above problem, we provide a brief primer on *stateful* in-switch applications, where a state on the switch determines how to process packets. A typical program (p) contains one or more *stateful objects* (o_i), each of which can be represented as a P4 construct [25] such as a match-action table and a register.[3] Each object contains *state data* in the form of key-value pairs ((K_{o_i}, V_{o_i})) and *actions*. For example, a register in P4 consists

[2] In Fig 1b, adding the 4th app does not increase the SRAM usage much because the sketch's SRAM usage is independent of the number of flows.

[3] While our focus of this paper is on P4, other programming languages for programmable switches such as NPL [15] provide similar constructs.

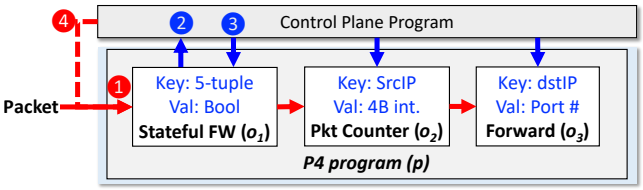


Figure 2: An abstract P4 application and runtime model. An application consists of multiple stateful objects (white boxes) and the control plane logic (blue arrows).

of a data array and actions that access the array. Fig 2 shows an example stateful P4 program (p) with three objects (o_1–o_3). Each object requires some amount of memory (e.g., SRAM) for state data and compute resources (e.g., stateful ALUs (SALUs)) for actions. The vendor-provided compiler (e.g., Tofino P4 compiler) allocates resources to each object using proprietary heuristics; if it cannot find a feasible allocation, the compilation will fail.

Once the program is successfully compiled and loaded to a pipeline, it can process incoming packets using its stateful objects; e.g., the firewall application in Fig 2 tracks active connections and drops unwanted packets from the Internet that do not belong to active connections. At runtime, the control plane logic can access the objects in the data plane (e.g., inserting a new entry to the stateful FW object). Note that in the current switch architecture, inserting and deleting entries from a match-action table can be done only via the control plane. From the data plane, a packet only can look up an entry from the table. Registers can be read and updated by both the data and control plane. For example, in Fig 2, when a packet from an internal network comes in and if a state miss occurs at the stateful FW (❶), it reports the packet to the control plane program (❷) that inserts new entries for the packet (or flow) (❸). Optionally, it sends the packet back to the data plane (❹) so that it can be processed with the inserted entries.

2.3 Prior work and limitations

At a high level, our work is related to prior efforts in switch program composition (e.g., [32, 53, 58, 59]), recent efforts to tackle switch resource constraints (e.g., [30, 40, 54]), and prior work in the software stateful NF literature (e.g., [31, 36, 37, 50, 57]). While these efforts are valuable, they do not tackle the problem our motivating scenario poses—*multiple concurrent switch applications with demanding workloads*.

Language and framework for application composition. Some prior works attempt to support multiple data plane programs or modules in a single device [32, 53, 58, 59]. For example, software-based virtualization approaches such as Hyper4 [32] and HyperV [58] allow composing multiple P4 programs with a constrained programming model. P4Visor [59] merges different versions of a program resource-efficiently. However, they fail to work when the amount of resources required by the composed program exceeds the available resources in the switch.

Leveraging external resources. TEA [40] provides a virtual table abstraction for a single switch application to access remote DRAM for a large lookup table. While TEA can be extended for an application with multiple tables, it requires multiple remote memory accesses, affecting application's performance. Flightplan [54] takes a single application written with custom annotations and disaggregates it to multiple devices. Developers need to manually partition the application so that each device runs only a particular portion of the application. Lyra [30] proposes a custom language for writing a single application split across multiple heterogeneous switches. None of these considers multiple applications.

Server-based network functions. In the context of server-based NFs, previous work augments servers' resources by leveraging remote compute and storage resources, especially to manage NF state [31, 36, 37, 50, 57]. However, they are not directly applicable in our setting due to the workload characteristics of switch applications and hardware constraints.

3 Overview

In this section, we make a case for on-rack switch resource augmentation and discuss the challenges in realizing it.

3.1 Case for on-rack augmentation

Given the above trends and limitations of prior work, one can consider several candidate approaches; e.g., optimizing applications to reduce resource footprint or adding more resources to switch ASIC. While these are valid, they have limitations; e.g., applications, even if optimized, may have high resource usage, especially with changing workloads, and extending switch hardware is expensive.

We explore a practical alternative and envision an *on-rack switch resource augmentation* architecture consisting of a programmable switch connected to a few other programmable *external* devices on the same rack. For example, we can allocate 2U of rack space, where a programmable switch is located, to install a server equipped with four 100 Gbps NPU, DPU, or FPGA-based smart NICs [6, 18, 34] connected to the switch. While these NICs provide a lower packet processing rate (up to a few 100s Gbps), compared to hardware switches (a few tens Tbps), they have more resources (e.g., a few GB of DRAM vs. a few 10s MB of SRAM) to support demanding workloads. This architecture provides a practical deployment solution as it takes up limited space and does not require changes to other parts of the network.

Deployment assumptions. In this context, we assume the following deployment capabilities: (1) A switch and external devices located on the same rack are programmable with the same set of P4-16 [10] constructs (e.g., tables and registers), and we have blackbox access to vendor P4 compilers; (2) External devices have enough memory (e.g., a few GB) to

store all states for multiple applications;[4] (3) Each application handles its own non-overlapping subset of traffic, called a traffic class, with no inter-app dependencies (i.e., a given packet is processed by only a single application). This simplifies our design for merging programs, but it is also a limitation of our current design;[5] (4) The number of switch pipeline stages is not a bottleneck resource; and (5) Stateful objects that can be updated in the data plane, which we call data plane-updatable objects, only maintain *mergeable* statistical data (e.g., packet counter) that do not impact the control flow.

3.2 ExoPlane architecture

While the vision of on-rack switch resource augmentation is promising, to realize it in practice, we need to effectively share resources available on multiple devices across multiple applications. Drawing an analogy from traditional computing [21], ideally, we need an OS to provide an *infinite switch resource abstraction*. That is, application developers and network operators can express their programs and requirements at a higher level of abstraction without worrying about the complexities of managing and multiplexing the resources on heterogeneous devices. While some early efforts have leveraged resources on heterogeneous devices for individual in-switch applications [30, 40, 54], they do not provide the OS-like capabilities and abstractions for multiple concurrent applications.

A classical OS multiplexes multiple applications on the limited CPU/memory by choosing when, and what processes, to swap in/out. Our workload is a set of incoming packets mapped to various in-switch applications. In this setting, state management becomes especially critical to system performance and resource overheads. To see why, let us consider two seemingly natural strawman solutions:

- In an *app-pinning* model, an application is pinned to a single device by placing the entire application states only on that device, and thus a packet is entirely processed on the device without requiring additional logic. In this model, there is no additional processing latency due to inter-device rerouting and resource overhead. However, since the application can only run on that particular device, its throughput and available resources are limited.
- Alternatively, we can consider a *full-disaggregation* model where an application can run on multiple devices, and a packet also can be processed on multiple devices. Since application states can be placed on any device, it has more available resources. However, depending on the availability of the state, a packet needs to be routed between the switch and the external device multiple times. Such frequent inter-device routing increases packet processing latency and makes it unpredictable. This approach incurs high resource overhead due to per-object inter-device processing logic to route packets to a particular device and resume processing at that object on the device. Furthermore, this approach consumes significant link and device bandwidth.

[4] We acknowledge that not every P4-programmable device supports all the features used by a switch application. According to our conversations with vendors, they plan to add such missing features, so this is not a fundamental limitation. Nonetheless, our design adapts such devices as well by considering app-to-device compatibility.

[5] One possible approach for this is to apply offline preprocessing steps to convert overlapping subsets into an equivalent non-overlapping set [48].

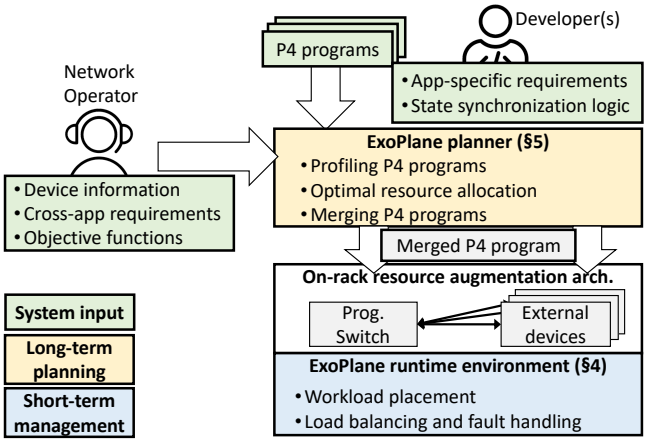


Figure 3: ExoPlane Overview: Green boxes represent inputs and yellow and blue boxes indicate key modules.

Building on the above insights, we adopt a *packet-pinning* model that pins a given packet to one device (i.e., the switch or an external device) where it is processed entirely while providing flexibility in placing an application and its flows on any device. First, it can avoid frequent per-packet inter-device routing with much lower complexity. Second, our observation from real network traces shows that only a small fraction of popular flows serve the majority of traffic for a given application (e.g., 6% of flows takes more than ≈80% of an Internet backbone traffic [20]). By placing these popular flows on the switch, we can process the majority of the packets at the switch, while the rest are processed at the external device.

ExoPlane implements the packet-pinning operating model via two key components (Fig 3):

- **The ExoPlane planner** takes inputs from developers and the network operator and allocates resources on the switch and external devices to each application.
- **The ExoPlane runtime environment** places workloads on devices, manages app states, and handles external device failures. In particular, at runtime, it tracks workload changes (i.e., new flows arrive or flow popularity changes) and updates the application's objects at the switch and external devices according to the changes.

As illustrated in Fig 3, to run applications on ExoPlane, developers provide P4 program codes and app-specific requirements (e.g., affinity to the switch). Note that ExoPlane requires application modifications only if it contains a data plane-updatable object whose copies can exist on multiple devices. The operator provides information on devices (e.g., resource types), cross-app workload (e.g., traffic distribution), and an objective function. The ExoPlane planner profiles the

applications to determine compatibility with each device and estimated resource usage, and performance. It then computes an optimal resource allocation and generates a merged P4 program. It compiles the merged program using vendor-provided P4 compilers (e.g., Tofino compiler) and loads the binaries to the switch and external devices. At runtime, the ExoPlane runtime environment executes the workload (i.e., packets) across the switch and external devices.

3.3 Design challenges

While the packet-pinning model for concurrent applications seems promising, managing resources and application states correctly in practical settings present three challenges:

C-1. Correctness under new flow arrivals and popularity changes. When the traffic workload changes, we need to update the application's objects at the switch. We find that this can lead to incorrect packet processing due to the slow control plane operations. Also, when there are multiple copies of a data plane-updatable object across devices, those copies can be updated simultaneously. Unfortunately, it is infeasible to adopt shared object synchronization schemes used in server-based systems [31, 50, 57] due to hardware constraints.

C-2. Handling multiple devices and device failures. While one can add more external devices to extend resources or processing capacity, we find that just adding more devices would not be effective due to possible access load imbalance across the external devices. Also, when an external device fails, we need to detect and react to the failure rapidly.

C-3. Meeting objectives across applications. Given multiple applications, we have to share resources among them properly while considering per-app and cross-app objectives provided by an operator and developers.

4 ExoPlane runtime environment

In this section, we discuss the design of the ExoPlane runtime environment. For clarity, we start with a few simplifying assumptions—a single instance of an external device, steady state traffic with no workload changes, no data plane-updatable state, no device failure, and a single application. We relax these assumptions subsequently.

4.1 Packet-pinning operating model

Recall from §3.2 that the packet-pinning model ensures that each packet is processed at a single device only (i.e., requires at most a single round-trip between the switch and an external device). Here, we load an application binary and all the state entries on an external device with a subset of entries loaded along with the application on the switch. As mentioned in §3.1, an external device has a few GB of DRAM, which is enough to store all the state entries (requiring up to a few hundred MB for a few million entries). If there is no entry for an incoming packet at the switch, the packet is routed to an external device as all the state entries needed to process the packet will be available.

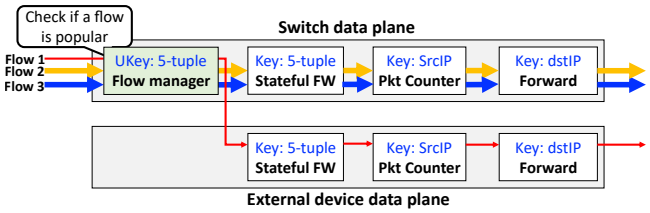


Figure 4: Our runtime environment processes most traffic at the switch and the rest at the external device. The green box is a per-app flow manager, and *UKey* indicates the app's union key.

However, naïvely implementing the packet-pinning model has two potential problems. First, if we do not carefully choose which entries to place on the switch, a high volume of traffic will be routed to the external device, and it becomes overloaded, limiting the throughput. Second, since an entry miss can happen for an arbitrary object, per-object inter-device processing logic is needed to handle such cases. Such additional logic incurs switch data plane resource overheads.

To tackle this, we propose a *union-key based* state management to process a majority of traffic for an application at the switch and the remaining at the external device (Fig 4). We define a union key type (UK) for an application as the union of key types of its constituent objects ($UK = \cup_i K_{o_i}$). A flow is a set of packets with the same union key value. For example, in the figure, an IP 5-tuple is the union key type, and packets with the same IP 5-tuple form a flow.

Having defined the union key, we can use traffic workload characteristics to enable the switch to serve the majority of traffic for the application. Specifically, we build on the observation that the distribution of flow keys (including the union key) is skewed in typical networks. For example, we measure the distribution of IP 5-tuple which is the union key of our example application, by analyzing packet traces collected from an Internet backbone [20] and a university data center [22] (Fig 15 in Appendix C illustrates the distributions). For both cases, we see that a small fraction of the keys contribute to the majority of the traffic; ≈6% of keys in the backbone and ≈10% of keys in the data center take more than ≈80% of traffic. The skew persists across measurement epochs (5 mins and 1 min for the backbone and data center, respectively). We also confirm the skew exists for other coarse-grained keys such as the source IP. This suggests that we can serve most of the traffic at the switch by placing a few popular union keys (e.g., 516 entries for 80% in the data center trace).

Based on this, we employ a per-app *flow manager* (the green box in Fig 4 and denoted as o_{FM}) at the switch, which maintains a list of popular union keys for an application and checks if the key of an incoming packet exists in the list when it arrives. If the key exists (i.e., the packet is from a popular flow), the packet is processed at the switch. Otherwise, it is routed and processed at the external device. This allows for low overhead by avoiding per-object inter-device process-

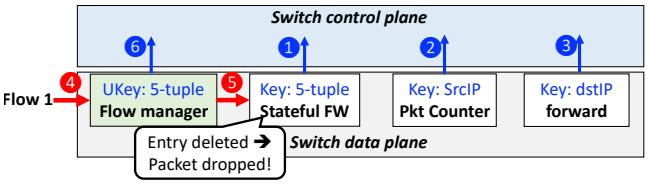


Figure 5: Incorrect state eviction: application's state has been removed while there is a packet being processed.

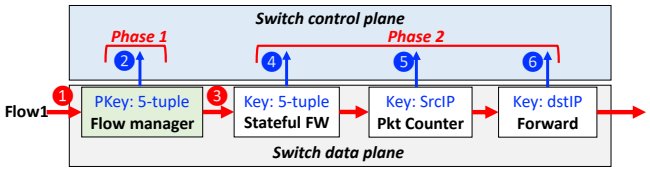


Figure 6: Correct two-phase state eviction.

ing. Taken together, our packet-pinning model and data plane design provide the following correctness property:

Invariant 1 (Packet-pinning model) *For each application, if the flow manager (o_{FM}) has the packet's union key ($UK(pkt)$), the constituent objects (o_i) must have entries ($K_{o_i}(pkt)$) for the packet. Formally,*

$$\forall pkt : UK(pkt) \in o_{FM} \implies \forall i : K_{o_i}(pkt) \in o_i.$$

4.2 Handling workload changes

So far we assumed a steady state—(1) no new flows and (2) no changes in flow popularity. Next, we discuss how we handle new flows and popularity churn.

Handling new flows. When a packet belonging to the new flow arrives at the switch, and if a miss occurs in the flow manager, it routes the packet to the external device. There are two cases for the miss: (1) the first packet of the new flow or (2) a packet of an existing flow for which the flow state is not at the switch. Since these two cases are indistinguishable from the view of the flow manager, it always routes packets with misses to the external device. When a packet of the new flow arrives at the external device, it must first be processed by the application's control logic for handling new flow arrivals. In our example, the stateful FW table reports the packet to the control logic that inserts entries for the flow to three objects. The control logic also asks the control logic running on the switch to initialize the flow state at the switch data plane, which can succeed only when there is a space on every object. Depending on the application logic, the packet can be sent back to the data plane and processed with the new entries. If the flow state has been initialized both at the switch and the external device, the switch will process subsequent packets in the flow. Otherwise, the external device will process them.

Promoting popular flows. In practice, the popularity of flows can change, and we need to promote and demote flow states as needed. Suppose we know which flow keys become popular (i.e., their entries are currently not on the switch) and unpopular (i.e., their entries are currently on the switch). We discuss how we track this in §6.

When promoting a new popular flow (i.e., installing state at the switch), there are two possibilities: (1) there is spare space in the flow manager and application's other objects for new entries vs. (2) there is no room in the objects. For (1), we can simply insert new entries to the objects. For (2), however, we need to evict some unpopular flow to make room.

Doing so correctly is challenging. Fig 5 illustrates why via a naïve mechanism can violate Invariant 1. Suppose that flow 2 becomes popular while flow 1 becomes unpopular, and there is no room for inserting new entries. Thus, the switch control plane tries to replace the entries for flow 1 with the flow 2's. It first evicts entries for flow 1 from application objects (FW, Counter, and Forward) as well as the flow manager (blue arrows in Fig 5). However, in the current switch architecture, a set of eviction operations (blue arrows) cannot be executed atomically. Thus, there could be cases where state entries have been removed while packets are being processed in the data plane (❺), violating Invariant 1. Even if eviction is correct, insertion can be incorrect. That is, during the time the switch control plane tries to insert entries for a flow, packets for the flow arrive and are looked up the flow manager. If the entry exists, the packet must be processed completely at the switch. However, since entries in other objects may not be available, the packet cannot be processed and will get dropped.

Two-phase state update. To address the issues, we adopt a *two-phase state update* mechanism, inspired by classical two-phase update or commit protocols [23, 51]. As illustrated in Fig 6, when evicting entries for flow 1, in the first phase, the switch control plane evicts an entry from the flow manager. Since there can be some packets being processed in the switch data plane, it waits for a certain period (T_{flush}) to flush out the packets. Then, in the second phase, it evicts entries from the application's objects. This mechanism ensures that all packets that arrive at the switch before the entry of the flow manager has been evicted are correctly processed in the switch. When it evicts entries from the application's objects, it ensures that entries for other non-victim flows will remain. The insertion works similarly. To insert entries for a flow, in the first phase, the switch control plane inserts entries to the objects, and then in the second phase, it inserts an entry to the flow manager.

4.3 Synchronizing shared stateful objects

The previous discussion considers scenarios with no cross-flow objects that can be updated at runtime, which meant there was no need for objects on an external device and the switch to be synchronized. In practice, applications may have such objects; e.g., per-SrcIP packet counter in our example is shared across flows. Next, we extend the basic ExoPlane protocol to handle such objects.

Consistency modes. P4 programs can have two types of stateful objects: (1) *control plane-updatable object* can be updated *only* from the control plane, such as a match-action

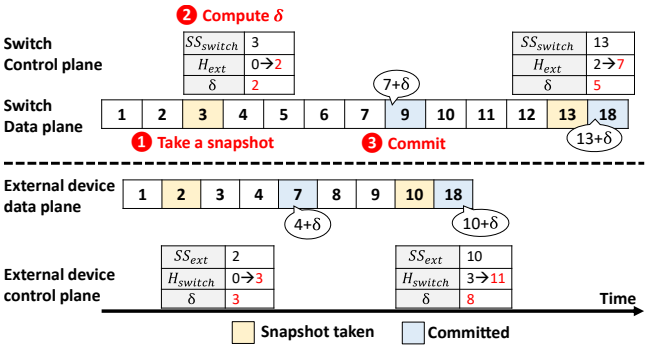


Figure 7: Our state synchronization protocol synchronizes two copies of an entry in the packet counter. The switch and external device's control plane maintain the history (H) of entry updates.

table and (2) *data plane-updatable object* can be updated from the data plane, such as a register. Correspondingly, ExoPlane provides two levels of consistency. Control plane-updatable objects are rarely updated (e.g., a stateful firewall table entry is inserted only for the first packet of each flow generated from an internal network) and an exact value is critical for correct behavior (e.g., allowing packets for an established TCP connection). Thus, for this type, we provide a *strong consistency*. In contrast, data plane-updatable objects can be updated more frequently (e.g., per-SrcIP packet counter is updated for every packet) in the data plane and typically do not require strong consistency since they maintain approximate or statistical information (e.g., packet counters and sketches). Thus, for data plane-updatable objects, we provide *bounded inconsistency* within a configurable time bound T_ε (e.g., 1 s in our prototype). As mentioned in §3.2, we consider bounded inconsistency only for mergeable statistical stateful objects.

Supporting strong consistency for control plane-updatable objects is straightforward; when the external device's control plane receives a request for updating (or inserting) an entry to an object with a key (e.g., a SrcIP), it updates (or inserts) all entries corresponding to the key existing at the external device and the switch.

Bounded inconsistency for data plane-updatable objects is more challenging. Consider the per-SrcIP packet counter implemented using a P4 register array. Suppose that for a given SrcIP, there are two copies placed on the switch and the external device that can be updated simultaneously. To achieve bounded inconsistency, the ExoPlane runtime needs to periodically *merge* values of the copies. Traditional techniques for state merging in server-based network functions (e.g., [31,50,57]) are impractical in our context since they rely on buffering incoming packets and pausing processing while synchronizing copies. This is expensive and even infeasible in the switch because packet rates are much higher, and we cannot buffer arbitrary packets during synchronization.

Our approach for bounded inconsistency. We devise a state synchronization protocol that achieves bounded inconsistency without needing packet buffering. We do so by combining the capabilities of both the switch and the external device's control and data plane. We use the control plane's memory to track the history of periodic synchronizations while executing the merge operation in the data plane.

Let us revisit our packet-counter example from Fig 7. The control plane of each device maintains per-entry metadata including the current snapshot (SS) and a history (H) of an entry value on the other side (i.e., the switch tracks the history of the external device and vice versa). When there are multiple objects that need to be synchronized, the control plane maintains metadata for each object. We discuss the overhead of maintaining the metadata in §7.5. For every T_ε seconds, the switch control plane initiates synchronization by sending its SS and the H, and the external device's control plane replies it with its snapshot and history; in Fig 7, the switch control plane takes the snapshot of the packet counter (❶) and sends $<SS$=3, H=0$>$ to the external device, and the external device sends $<SS$=2, H=0$>$ back. Then, each side computes the changes that have been made on the other side (δ) after the previous synchronization round by subtracting two history values from the received snapshot value (❷). This prevents a potential under or double-counting issue. Lastly, the control plane of both sides injects a special control packet containing δ to the data plane to merge the changes to the latest state value (❸). Note that our protocol synchronizes the copies of states correctly even when the external device fails and recovers. This is because the switch maintains the progress that the external device had made until the failure happened (H) and provides this information to the recovered device to resume the synchronization from the state when it failed.

Generally, our protocol supports mergeable key-value pair states for most stateful objects implemented using P4 registers. We provide a developer with an interface to specify object-specific *merge operators* consisting of an addition ($\circ^+$) that merges two values and an optional subtraction ($\circ^-$) operator that subtracts one value from the other, which are used by our protocol to compute δ and commit the update. For example, a Bloom filter [24] can be expressed as (Key: an integer, Value: $\{0, 1\}$) pairs with the binary OR as $\circ^+$ (no subtraction operator is needed). We provide a detailed pseudo-code in Appendix B.

4.4 Scaling to multiple devices

Thus far, we have assumed that there is a single external device. However, in practice, a single device instance may not provide enough processing capacity or resources. To use multiple devices, ExoPlane shards entries in objects across the devices based on the union key. When an entry miss occurs at the flow manager, it routes a packet based on the union key to a specific external device that has state for the key. However, the skewness in the union-key space (§4.1) could result in load imbalance across the devices (i.e., a subset of devices can be overloaded). Fortuitously, the small fraction of popular entries we already have at the switch is helpful for load balancing. Prior analysis in storage systems shows that caching at least $O(N \log N)$ popular entries where N is the number of backend

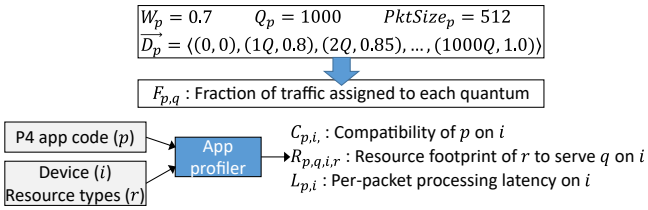


Figure 8: Example inputs for an application p.

servers (in our context, external devices), guarantees uniform load balancing across the servers regardless of the skew [29]. Thus, by placing $\geq O(N \log N)$ popular union keys at the switch, we can provide the cache effect for load balancing.

4.5 Handling external device failures

Application state loss due to failures can affect the performance or correctness of applications [41]. Specifically, we consider the failure model where an external device (or its hosting machine) fails or a network link between the switch and the device fails.[6] To deal with state loss due to such failures, the ExoPlane runtime environment replicates each flow state to at least one additional external device when initiating flow entries, and when the primary device fails, it falls back to a replica. It does so by managing the logical to physical external device ID mapping at the switch, where the primary and replicas share the same logical ID. However, even if there is a replica, if we cannot detect failures and route packets to the replica quickly, the application performance can be degraded (e.g., due to packet drops). To enable rapid failure detection and reaction, we repurpose the packet generation engine of the switch ASIC (which is typically used for diagnosis), similar to previous work [40]. We configure the engine to generate a packet when ports go down. By processing the generated packet, the runtime environment updates the external device ID table in the data plane. Using the table, it can route subsequent packets to the replica.

5 ExoPlane planner

Next, we tackle the issue of sharing resources across multiple applications while meeting the performance objectives given by developers and operators. The resulting ExoPlane planner consists of a resource allocator and an application merger. The resource allocator finds an optimal resource allocation using inputs from the developers and the network operator. The application merger generates a merged P4 program based on the optimal allocation decision. Fig 8 illustrates example inputs for an ensemble of applications.

Inputs. Developers provide a set of P4 programs (p), each of which consists of a set of stateful objects. For each object, developers specify the required size (e.g., the number of entries in a table or register). Optionally, they can also specify a high, medium or low affinity to the switch for each app. If the affinity of an application is set to high (or low), the application will run entirely at the switch (or at external devices). The network operator provides cross-app and per-app traffic information, which includes the fraction of all traffic served by each application out of the entire traffic arriving at the switch (W_p) and the cumulative traffic distribution (D_p) over the union key space. While using a fraction of traffic served by *each key* provides the most fine-grained information, we find that it could make the search space for resource allocation too large. Instead, we use the distribution discretized into a larger quantum size denoted as Q_p. Based on D_p, we compute the estimated fraction of traffic served by each quantum q ($F_{p,q}$). The operator also provides resource information (r) for devices (i). This includes SRAM, TCAM, hash units, and SALUs for a Tofino-based switch and compute units, SRAM, and DRAM for NPU-based NICs.[7]

Profiler. Based on the inputs, our profiler generates per-app profiles consisting of a resource footprint, per-packet processing latency, and compatibility matrix for each device type. The profiler estimates a resource footprint of r for p serving q on i denoted as $R_{p,q,i,r}$. Since blackbox compilers determine the resource usage using proprietary heuristics, our preprocessor compiles p to determine $R_{p,q,i,r}$. For each q, it updates the size of each object specified in the application code and compiles it using vendor-provided compilers. Then it extracts the resource usage from compiler outputs. If the compilation fails due to insufficient resources, it sets the resource usage to infinite. We use constants $Cap_{i,r}$ to represent the total amount of r available on i. The profiler also estimates a per-packet processing latency of p on i, $L_{p,i}$. Specifically, it instruments the switch to record two timestamps on a custom packet header field when a packet enters and leaves the rack. Then it injects $PktSize_p$-sized packets to the rack and estimates the latency based on the timestamps in returned packets.

Finally, some vendor-provided P4 compilers for external devices may not support certain features or P4 constructs[8] used by applications. Because of this, if an application uses a feature that is not supported by an external device, it cannot run the application. To consider the compatibility of the application on devices, our profiler generates a compatibility matrix ($C_{p,i}$) that indicates whether p can be run on device i based on a set of features supported by i and a set of features used by p. The first set can be typically obtained from vendor's compiler manual. For the second set, the profiler analyzes the application code to extract used features.

Resource Allocation. Given these inputs, we can formulate the problem of finding an optimal resource allocation satisfying per-app and cross-app requirements. In our formulation, we assume that the resource usage of multiple applications can be estimated by accumulating the resource usage of each app.

[6] We do not consider the failure of the switch itself since in that case, packets cannot be processed in our deployment model (§3.1) where the switch is the single entry point of the architecture.

[7] The operator can easily extend this to other resource types.

[8] e.g., Packet recirculation and P4 registers.

We use binary decision variables $d_{p,q,i}$ to indicate whether q for p is assigned to i. There are four constraints imposed:

$$\forall p,q : \sum_i d_{p,q,i} = 1 \quad (1)$$

$$\forall p,q,i : d_{p,q,i} \leq C_{p,i} \quad (2)$$

$$\forall i,r : \sum_p \sum_q d_{p,q,i} \times R_{p,q,i,r} \leq Cap_{i,r} \quad (3)$$

$$\forall p : lat_p = \sum_q \sum_i d_{p,q,i} \times F_{p,q} \times L_{p,i} \quad (4)$$

First, q must be assigned to a unique i (Eq. 1). Second, q can be assigned to i if and only if p is compatible with i (Eq. 2). Third, the amount of r consumed by q on i must be less than or equal to the total amount of r on i (Eq. 3). Last, the expected latency of p is the sum of per-packet processing latency of p on i weighted by $F_{p,q}$ (Eq. 4).

The network operator provides an objective to share resources across multiple application fairly. One possible fairness metric would be minimizing the weighted sum of the expected processing latency of each application:

$$\text{Minimize} \sum_p W_p \times lat_p \quad (5)$$

Other commonly used fairness metrics such as maximizing the minimum expected throughput can be used as well. By solving the ILP, the ExoPlane resource allocator finds an optimal assignment of q to i for p, and the size of each object and flow manager for p accordingly, which are used as input for the application merger, as we describe next.

Application Merger. Given a set of P4 programs and the optimal resource allocation decision, our application merger combines the programs into a single P4 program, following our deployment model for multiple applications, described in §3.2. Our merger supports programs written in P4-16 [10] (Fig 16 in Appendix C illustrates how the merger works). First, for each app, the merger renames the main control block [10] to avoid naming conflicts between applications. Second, it specifies the size of each object (e.g., number of entries in a table) based on the decision made by our resource allocator. Third, it inserts an flow manager. Finally, in the merged P4 code, it instantiates an instance of each application and inserts execution logic. The merged P4 code is compiled using the vendor-provided compiler and loaded on the switch and external devices. Sometimes, the compilation process fails due to its proprietary heuristics for resource allocation. If so, we repeat the process with a tighter resource constraint.

In summary, ExoPlane planner allocates resources across applications based on inputs from developers and the operator and produces a merged P4 program. This process needs to be re-run when a set of applications or workloads changes, which we do not expect to happen frequently (e.g., once every hour). While this module is not on the critical path, performance results are available in Appendix D.

6 Implementation

Data plane. The data plane components of the runtime environment implemented in P4-16 consists of: (1) the flow manager implemented using a match-action table and (2) the global logical-to-physical external device ID mapping implemented using a register array (on the switch).

Tracking flow popularity. We implement a flow popularity tracker on external devices using the count-min sketch [27] that tracks the frequently accessed flow keys. When it detects a new popular key, it reports the key to the external device's control plane that has a list of flow keys and corresponding entries, and they are reported to the switch control plane. We enable the *aging* supported by the switch ASIC for the flow manager. If a certain key has not been accessed for a timeout period (T_{idle}), a callback function registered at the switch control plane is triggered, and it evicts the entry corresponding to the idle key. In our prototype, we set T_{idle} to 2 s.

Control plane. We implement the control plane of the runtime environment in Python and C++. The main capability is to initialize new flow entries and promote new popular flows' entries on the switch. On the switch side, we use Barefoot Runtime APIs to access the stateful object in the switch data plane. On the NIC side, we use Netronome Thrift APIs [12] to interact with the NIC data plane. The switch and the external device control planes are communicated via an out-of-band TCP session over the 1 Gbps management network.

Resource allocator. We implement the resource allocator in C++ based on the Gurobi C++ API [13] to encode and solve our resource allocation ILP.

Application profiler and merger. We extend the open-source P4 compiler [17] to analyze input P4 programs. Using its frontend, we extract information from each program including the entry size of each object. We implement the application merger in C++, which takes an IR generated by the compiler frontend, and produces a merged P4 program.

Supporting other hardware platforms. While our prototype uses a Tofino-based programmable switch and Netronome smart NICs, ExoPlane can be extended to other platforms. For example, ExoPlane can be applied to other types of programmable switches (e.g., Nvidia Spectrum-2 [16]) and FPGA or ASIC-based smart NICs (e.g., Xilinx and Intel FPGA NICs [1, 6] as external devices.

7 Evaluation

We evaluate ExoPlane on a testbed consisting of a programmable switch and servers equipped with a Netronome smart NIC using various workloads. Our key findings are:

- In steady-state, ExoPlane provides predictable per-packet latency (e.g., 273—384 ns at the switch) and scalable throughput with more external devices while the app-pinning model achieves a limited throughput (§7.1).

Applications	States
Per-VM NAT	Per-flow address mapping for each VM.
Per-VM Stateful FW + Packet counter	Active TCP connection list.
Per-VM SYN proxy	Per-flow sequence number translation table.
NetCache [35]	Key-value store cache.

Table 2: Switch programs written in P4 used in the evaluation in addition to ones introduced in Table 1.

- Even under dynamic workloads, ExoPlane can process packets with the correct state and sustain high throughput with multiple devices (§7.2).
- In case of an external device failure, ExoPlane can recover an end-to-end TCP throughput within 200 *ms* (§7.4).
- ExoPlane provides the above benefits with small control plane (e.g., a few tens MB) and switch ASIC resource overheads (e.g., less than 4.5% of ASIC resources) (§7.5).

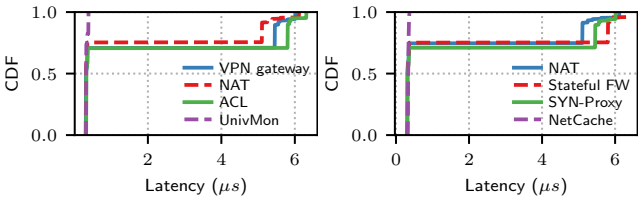
Testbed setup. We build an on-rack resource augmentation architecture consisting of Wedge100BF-32X Tofino-based programmable switches [9] and four servers equipped with Netronome Agilio 40 Gbps smart NICs [7]. We use four additional servers with 100 Gbps regular NICs to generate traffic workloads. All servers are equipped with an Intel Xeon Silver 4110 CPU and 128 GB DRAM, running Ubuntu 18.04. We repeat each experiment 100 times unless otherwise noted.

Traffic workloads. We use packet traces from a real data center [2], the Internet backbone [20], and synthetic ones. The packet sizes vary (64–1500 B) in the real trace. We generate traces with the flow key distribution in terms of the number of packets per flow that follows a Zipf distribution with the skewness parameters (α=0.9, 0.95, 0.99). We use a keyspace of 1M randomly generated IP 5-tuples when creating packet traces. We generate packet traces with different packet sizes and skewness parameters. We replay the traces using DPDK-pktgen [3] or run iperf [5] for TCP workloads.

Deployment scenarios. We use two scenarios with multiple P4 applications: (1) at the data center gateway, four applications in Table 1 and (2) at the leaf of the network, four applications from Table 2. Given packet traces, we synthesize inputs for the ExoPlane planner (e.g., per-app affinity and a flow key distribution). For example, we set the affinity level for the UnivMon [45] and NetCache [35] to high so that workloads for these are always processed at the switch.

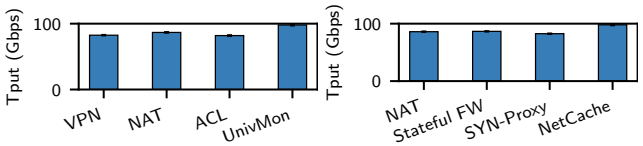
7.1 Performance in steady state

First, we evaluate the per-packet processing latency and throughput of applications running on ExoPlane in steady state (i.e., no new flows, no changes in flow popularity, and no device failures). Here, we pre-populate popular flow entries at the switch and assume that the traffic is equally distributed across the applications (i.e., $W_p = 0.25$ for all applications).



(a) Data center gateway. **(b)** Data center leaf.

Figure 9: Per-packet processing latency distribution of applications concurrently running on ExoPlane in steady state.



(a) Data center gateway. **(b)** Data center leaf.

Figure 10: Throughput of each application running on ExoPlane in steady-state with a single external device. Applications are running concurrently.

Per-packet processing latency. We define per-packet processing latency as the time difference between when a packet first arrives at the switch from a sender and when it is sent to a receiver after processing. We instrument the P4 program running on the switch to record two timestamps (48-bits each) to our custom packet header fields of each packet so that the receiver can compute the processing latency for a packet. From the sender, we replay the backbone packet traces, each of which contains more than 6M flows.

Fig 9 shows the CDF of the per-packet latency distribution for each application. For the applications that are assigned to the *high* affinity (UnivMon and NetCache), every packet is processed at the switch in 273–384 ns, depending on packet sizes. For other applications, the distributions vary depending on packet sizes and how much traffic is processed at the switch and the external device. The higher the affinity level assigned to an application, the more traffic is processed at the switch. For example, in the gateway scenario (Fig 9a), at the switch, the ACL processes ≈70% of its traffic whereas the NAT processes ≈75% of its traffic. When packets are forwarded to the external device, their processing latency becomes 5.1–6.1 µs, depending on the application (the top-right corner in Fig 9). The external device takes 3.2–4.1 µs to process each packet, while several overheads constitute the overall processing latency, including the switching latency and the propagation and transmission latency. While there is a latency gap between two cases (processing at the switch vs. external device), on each device, per-packet processing latency is predictable.

Application throughput. To measure the throughput, we replay the synthetic trace that consists of 1500 B packets at line rate (98.6 Gbps in our testbed). We use four sender nodes, each of which generates traffic for each of the four applications. We start with a single external device to demonstrate the impact of the number of external devices on the throughput.

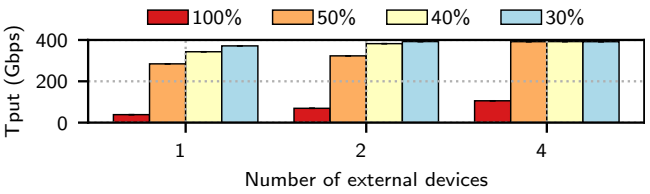


Figure 11: Scalable throughput with multiple devices with varying fractions of traffic offloaded to external devices.

Ensemble of apps	App-pinning	ExoPlane
VPN	98.6 Gbps	98.6 Gbps
VPN+NAT	137.1 Gbps	197.2 Gbps
VPN+NAT+ACL	174.6 Gbps	295.6 Gbps
VPN+NAT+ACL+UnivMon	271.3 Gbps	394.1 Gbps

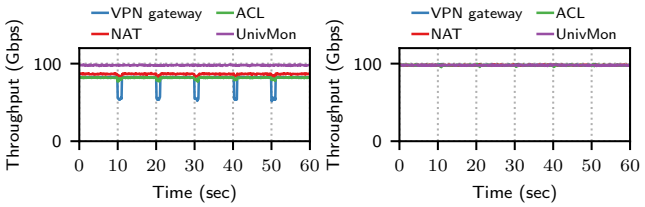
Table 3: Aggregate throughput of four applications running on the app-pinning model and ExoPlane with four external devices.

Fig 10 shows the throughput of each application. The applications that run entirely on the switch (UnivMon and NetCache) process traffic at line rate without dropping any packets. However, we observe that the others cannot process their traffic at line rate. This is because the aggregate amount of traffic across the applications, which needs to be processed at the external device (≈81 Gbps in the gateway case) exceeds the processing capacity of the single device (≈39 Gbps).

Scaling throughput with multiple devices. By adding more devices, ExoPlane can support higher throughput. To demonstrate this, we measure the aggregate throughput of the four applications in the gateway scenario (max. traffic rate in our testbed is ≈394 Gbps) while varying the fraction of traffic offloaded to an external device(s)[9] and the number of external devices. Fig 11 shows the results. In the case of 30, 40, 50% of the traffic being offloaded to external devices, we see the throughput effectively increases with more devices. In contrast, when 100% of traffic is offloaded, adding more devices is not effective due to load imbalance. This result shows the load balancing effect of serving popular flows at the switch, described in §4.4.

Comparison with the app-pinning model. We evaluate the benefit of ExoPlane over the app-pinning model (described in §3.2) while running four applications from Table 1. In this model, we place an application along with its entire state at the switch if there is room. Otherwise, we place it on one of the external devices, which has the largest remaining capacity. Table 3 compares the aggregate throughput when running an ensemble of applications. While ExoPlane provides the maximum throughput for each ensemble, the app-pinning model achieves up to 69.3% lower throughput. This is because while ExoPlane allows an application to effectively utilize available resources across different devices, the app-pinning model fixes an application to a device.

[9] In this experiment, we control the fraction of traffic offloaded to external devices by manually assigning the affinity of each application. UnivMon is still pinned to the switch.



(a) With a single external device. **(b)** With 4 external devices.

Figure 12: Throughput changes due to workload changes.

7.2 Performance under dynamic workload

Per-packet processing latency. As mentioned in §4.2, workload changes happen due to new flows arriving or changes in flow popularity. Handling new flows in ExoPlane can increase processing latency because the first packet of a new flow can be processed only after initiating the necessary state for both the flow manager and application objects. In contrast, packets in the flow that becomes popular can be processed either at the switch or an external device with the same latency shown in §7.1. Thus, for each app, we measure the processing latency of the first packet of each flow. We observe that the median latency for the first packet of a new flow is 32 *ms*, which is an order of magnitude higher than that of an external device in a steady state. There are two factors here. First, the Netronome Thrift API takes a few tens of *ms* to insert new entries to objects, which is not an ExoPlane-specific overhead. Second, since ExoPlane replicates entries for new flows to one another external device, it incurs additional latency when handling new flows. Note that as described in §4.2, ExoPlane tries to initiate state for new flows both at the switch (if there is room) and at the external device, so even for short-lived flows, the subsequent packets can be processed at the switch with lower latency. However, some short-lived flows can be entirely processed at the external device, which can incur higher latency if there is no room at the switch during its lifetime.

Application throughput. The changes in flow popularity can impact the throughput. To measure the throughput changes, we use the same setup as the previous measurement in steady state, but for every 10 s, we alter the most popular top 10 flows for the VPN gateway of the gateway scenario. Fig 12 shows the throughput changes over time. Again, we first use a single external device. As shown in Fig 12a, when the popularity changes, there is a sharp drop in the throughput of the VPN gateway. Also, the throughput of other applications slightly decreases as well. This is because until the state entries for the new set of popular flows are installed at the switch (i.e., a transient period), a high volume of traffic for the flows is routed to the external device, exceeding its processing capacity. On the other hand, as shown in Fig 12b, with four external devices, there is no such performance drop because there is enough processing capacity at the external devices to handle the traffic during the transient period. Although we assume that the traffic pattern can change at an hour or day-timescale,

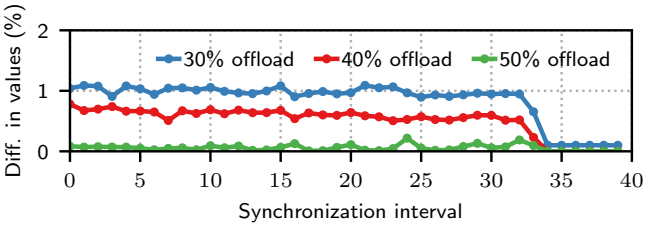


Figure 13: Difference in shared object values on the switch and external devices; the trace ends at epoch 32.

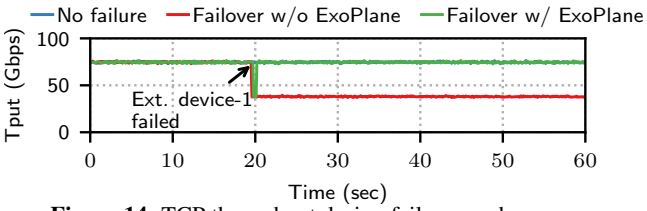


Figure 14: TCP throughput during failover and recovery.

if ExoPlane cannot detect such changes, it could suffer from severe performance degradation due to outdated resource allocation. Rapidly detecting the changes and reconfiguring the resource allocation accordingly is our future work.

7.3 Shared stateful object synchronization

Next, we evaluate the effectiveness of our state synchronization protocol (§4.3) using the per-SrcIP packet counter in the stateful FW. The metric of interest is the difference between the shared counter entries maintained by each device at each synchronization interval (T_ε=1 s). We measure this by recording the values at each device right after executing the merge operation in the data plane while injecting 1500 B packets for 60 s at 98.6 Gbps. We vary the fraction of traffic offloaded to external devices. In our setting, there are 1000 entries shared between the switch and at least one of the external devices, and we get the median of the differences. Fig 13 shows the result with three different fractions of offloaded traffic. When the switch and external devices process the same amount of traffic (i.e., 50% offload), there is almost no difference. When there is a gap between the amounts of traffic (i.e., 30% or 40% offload), there are differences because incoming packets keep updating the counter at each device during the synchronization, affecting the measured values. However, we see that the variance of the difference is small across the synchronization intervals regardless of the gap, showing that our mechanism synchronizes the values. We also confirm that after the packet transmission is done, copies at each device are synchronized with the same value as the total number of packets.

7.4 Failover

In Fig 14, we use a NAT as an example and run iperf to measure TCP throughput changes. There are four TCP connections, and we configure two of them to be processed at the switch while the remaining is processed at an external device. There are two external devices enabled, and we compare changes in TCP throughput when (1) there is no failure and (2) one of the external devices fails with and without ExoPlane. We emulate the failure by disabling a port connected to the external device. At around 20 s, when the external device-1 goes down, our failover mechanism generates a control packet that modifies the logical to physical device ID mapping in the switch data plane without involving the control plane. Then, subsequent packets are routed to the replica device. We see that the TCP throughput is recovered to its original rate within a 200 ms whereas, without ExoPlane, it cannot be recovered.

7.5 Runtime resource overheads

Control plane resource overhead. The control plane component of ExoPlane runtime environment maintains metadata for application states, including a mapping between union keys and devices and a history of each shared object entry on other devices. Each of them consumes the control plane memory. In our scenarios, the union keys-to-device mapping consumes 12.5 MB per application and the history metadata consumes 1.5 MB per shared object. Our state synchronization protocol consumes management network bandwidth as it periodically exchanges information between devices, which contains a snapshot and history of each entry. In our setting, the bandwidth consumption is 24.4 Mbps per shared object, which increases in proportion to the number of devices, the sync interval, and the number of entries.

Switch ASIC resource usage. The data plane component of ExoPlane runtime environment consumes some switch ASIC resources. Since we implement it using an exact-match table with the aging feature and a register array, it consumes SRAM, SALUs, hash bits, MAP RAM, and match crossbar,[10] whose usage increases proportionally to the number of popular flows maintained (except for SALUs). In our setting where 10240 popular flow entries are managed, it consumes 4.4% of the SRAM, 2.1% of SALUs, 3.5% of the hash bits, 3.8% of the MAP RAM, and 3.6% of the match crossbar, leaving ample resources for application logics.

8 Conclusions

Limited on-chip resources today block the deployment of concurrent stateful apps on programmable switches, limiting the adoption of in-network computing. In this paper, we argue that on-rack switch resource augmentation can be a pragmatic and incrementally expandable solution to this dilemma. To realize this vision, we present ExoPlane, which provides OS-like abstractions for the new architecture by addressing challenges in managing application states and resources across multiple devices. Our evaluation shows that ExoPlane provides low latency, scalable throughput, and fast failover, and achieves these with a small resource footprint and few/no modifications to applications. Thus, ExoPlane can be a practical basis for enabling in-network computing for future applications, workloads, and emerging data plane hardware.

[10] MAP RAMs are used for the aging feature and match crossbars are used for implementing the 'matching' part of match-action tables.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Michio Honda, for their insightful comments and constructive feedback. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by the NSF award 1700521, and by Intel Corporation and by VMWare under the Crossroads 3D FPGA Research Center.

References

[1] Netcope P4. https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/netcope-technologies--a-s-/ip/netcope-p4.html.

[2] Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2010.

[3] pktgen-dpdk: Traffic generator powered by DPDK. https://git.dpdk.org/apps/pktgen-dpdk/, 2011.

[4] Advanced network telemetry. https://www.barefootnetworks.com/use-cases/ad-telemetry/, 2018.

[5] iperf3. http://software.es.net/iperf/, 2018.

[6] P4-SDNet User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1252-p4-sdnet.pdf, 2018.

[7] Agilio CX SmartNICs - Netronome. https://www.netronome.com/products/agilio-cx/, 2019.

[8] Cisco Visual Networking Index. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html, 2019.

[9] EdgeCore Wedge 100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335, 2019.

[10] $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.0.html, 2019.

[11] Alveo U25 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u25.html, 2020.

[12] Apache Thrift. https://thrift.apache.org/, 2020.

[13] Gurobi - C++ API Overview. https://www.gurobi.com/documentation/9.1/refman/cpp_api_overview.html, 2020.

[14] Intel FPGA Programmable Acceleration Card N3000. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-n3000/overview.html, 2020.

[15] NPL Specifications. https://nplang.org/npl/specifications/, 2020.

[16] Nvidia mellanox spectrum-2. https://www.mellanox.com/files/doc-2020/pb-spectrum-2.pdf, 2020.

[17] p4c: a reference P4 compiler. https://github.com/p4lang/p4c, 2020.

[18] Pensando DSC-25 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf, 2020.

[19] The Software Switch Pipeline. https://doc.dpdk.org/guides/prog_guide/packet_framework.html#the-software-switch-swx-pipeline, 2020.

[20] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml, 2021.

[21] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice*, volume 1. Recursive books, 2014.

[22] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.

[23] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[24] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[25] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[26] Cisco. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper, 2018.

[27] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2), 2008.

[28] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX NSDI*, 2018.

[29] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SOCC*, 2011.

[30] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *ACM SIGCOMM*, 2020.

[31] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[32] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *ACM CoNEXT*, 2016.

[33] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research, 2021.

[34] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4-> netfpga workflow for line-rate packet processing. In *ACM/SIGDA FPGA*, 2019.

[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, 2017.

[36] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *USENIX NSDI*, 2017.

[37] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron: Nfv service chains at the true speed of the underlying hardware. In *USENIX NSDI*, 2018.

[38] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, 2016.

[39] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155, 2021.

[40] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, 2020.

[41] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *ACM SIGCOMM*, 2021.

[42] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic external memory for switch data planes. In *ACM HotNets*, 2018.

[43] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM SOSP*, 2017.

[44] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *USENIX OSDI*, 2020.

[45] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[46] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[47] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM*, 2021.

[48] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[49] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. Flexgate: High-performance heterogeneous gateway in data centers. In *APNET*, 2019.

[50] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *USENIX NSDI*, 2013.

[51] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. 42(4):323–334, 2012.

[52] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *USENIX NSDI*, 2021.

[53] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *ACM SIGCOMM*, 2020.

[54] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *USENIX NSDI*, 2021.

[55] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *ACM SIGMOD*, 2020.

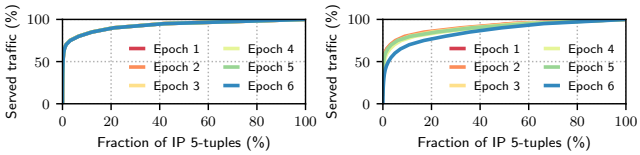
[56] William Tu, Fabian Ruffy, and Mihai Budiu. Linux network programming with p4. In *Linux Plumb. Conf.*

[57] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *USENIX NSDI*, 2018.

[58] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *IEEE ICCCN*, 2017.

[59] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.

A Skewness of traffic traces



(a) Internet backbone. **(b)** University data center.

Figure 15: Skewness in flow key (IP 5-tuple): For both Internet backbone and data center case, a few popular keys serve the most of the traffic. This is consistent across measurement epochs.

We measure the distribution of IP 5-tuple as the union key by analyzing packet traces collected from an Internet backbone [20] and a university data center [22]. Fig 15 shows the union key distributions of the two data sets.

B State synchronization algorithm

Algorithm 1: State synchronization – Switch

1 S_{switch} : The current state of the value on the switch
2 Ext: a set of external device IDs
3 SS_{switch} : The latest snapshot of the value on the switch
4 $H_{ext}[1 \ldots N]$: The latest information received from each external device
5 **Upon the snapshot timer expires**:
6 **foreach** $ext_i \in Ext$ **do**
    /* Send an initiate message to ext_i */
7   **send** $(Snap_{switch}, I_{switch}[ext_i])$;
    /* Receive a response from ext_i */
8   $(Snap_{ext_i}, I_{ext_i})$ = **recv** ();
9 **foreach** $ext_i \in Ext$ **do**
    /* Adjust snapshot values and merge them */
10 $\delta = Snap_{ext_i} \circ^{-} (I_{switch}[ext_i] \circ^{+} I_{ext_i})$;
    /* Update the information for ext_i */
11 $I_{switch}[ext_i] = Snap_{ext_i} \circ^{-} I_{ext_i}$
  /* Merge ($\circ^{+}$) the adjusted value with the current state in the data plane */
12 $S_{switch} = S_{switch} \circ^{+} \delta$

Algorithm 2: State synchronization – External device

1 SS_{ext} : The latest snapshot of the value on the external device
2 S_{ext} : The current state of the value on the external device
3 I_{ext} : The latest information received from the switch
4 **Upon receiving** a message from the switch $(Snap_{switch}, I_{switch})$:
  /* Send a response to the switch */
5 **send** $(Snap_{ext}, I_{ext})$;
  /* Adjust snapshot values and merge them */
6 $\delta = Snap_{switch} \circ^{-} (I_{ext} \circ^{+} I_{switch})$;
  /* Update the history for the switch */
7 $I_{ext} = Snap_{switch} \circ^{-} I_{switch}$;
  /* Merge the adjusted value with the current state */
8 $S_{ext} = S_{ext} \circ^{+} \delta$

In §4.3, we describe our state synchronization protocol to synchronize entries in a data-plane updatable object. Algorithm 1 and Algorithm 2 describe the detailed algorithm running on the switch and external devices, respectively.

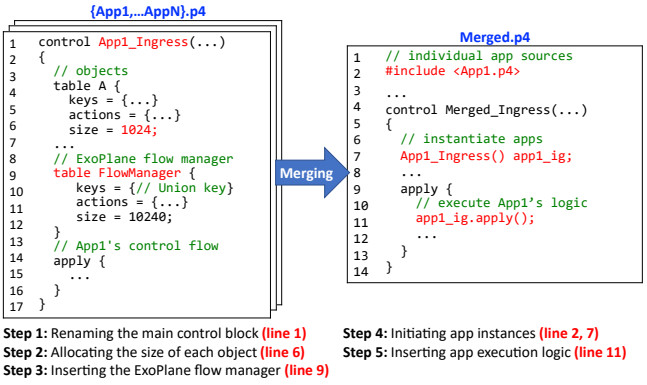
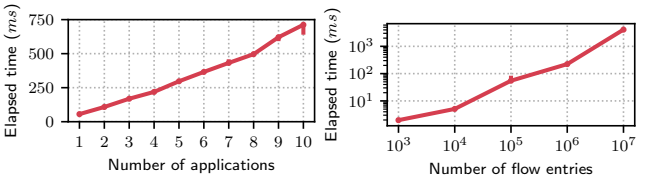


Figure 16: Merging multiple P4 programs into a single program.



(a) Different number of apps. **(b)** Different number of flow entries.

Figure 17: Elapsed time for the resource allocator.

C Details of Application Merger

Fig 16 illustrates how our application merger works for a set of P4 applications as described in §5.

D Performance of ExoPlane Planner

We evaluate the performance of the ExoPlane planner. In this experiment, we measure the elapsed time for finding optimal resource allocations and generating a merged P4 program on a server in our testbed. For the two sets of apps and the hardware configuration used in our evaluations, our resource allocation takes 54.5 *ms* and merging the program takes 642 *ms*, which is reasonable since the orchestrator needs to run this process on the hours or days timescale. To further understand the impact of different parameter values including the number of apps and traffic workload sizes, we synthesize inputs for the resource allocator and measure the elapsed time. First, we fix the number of external devices to 16 (to support a large number of apps) and the number of union key-based flow entries to 1M for each app. Then, we vary the number of flow entries while fixing the number of apps to four and the number of devices same as the above. As illustrated in Fig 17a, the resource allocation time grows linearly up to 712 *ms* as the number of app increases. Also, as shown in Fig 17b, as the number of flow entries increases, the elapsed time also increases up to 4.1 second when each app needs to handle 10M flow entries. This experiment illustrates the ExoPlane orchestrator takes a longer time as we add more apps and increases the workload size, which can be up to a few seconds, it is still within the reasonable timescale under our deployment model (§3.1).

Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches

Hun Namkung*, Zaoxing Liu†, Daehyeok Kim§, Vyas Sekar*, Peter Steenkiste*
*Carnegie Mellon University, †Boston University, §Microsoft Research

Abstract

Network operators need to run diverse measurement tasks on programmable switches to support management decisions (e.g., traffic engineering or anomaly detection). While prior work has shown the viability of running a single sketch instance, they largely ignore the problem of running *an ensemble of sketch instances* for a collection of measurement tasks. As such, existing efforts fall short of efficiently supporting a general ensemble of sketch instances. In this work, we present the design and implementation of Sketchovsky, a novel *cross-sketch optimization and composition* framework. We identify five new *cross*-sketch optimization building blocks to reduce critical switch hardware resources. We design efficient heuristics to select and apply these building blocks for arbitrary ensembles. To simplify developer effort, Sketchovsky automatically generates the composed code to be input to the hardware compiler. Our evaluation shows that Sketchovsky makes ensembles with up to 18 sketch instances become feasible and can reduce up to 45% of the critical hardware resources.

1 Introduction

Network operators need to concurrently run diverse measurement tasks for network management tasks such as traffic engineering, anomaly detection, load balancing, and resource provisioning [6, 10, 24, 35, 46]. A flow-level measurement task computes a desired statistic (e.g., heavy flow size or the distinct number of flows) based on the definition of a flowkey (e.g., srcIP or 5-tuple), a flow size (e.g., packet counts or bytes), and a measurement epoch (e.g., 1 minute).

Given resource constraints, *sketching algorithms* or *sketches* appear as a promising avenue to support measurement tasks on data collected from programmable switches (e.g., [4, 5, 7, 17, 21, 32, 40, 45, 49]). To support one measurement task, a sketch instance on a programmable switch is instantiated based on a sketching algorithm with configuration on parameters (e.g., flowkey or resource allocation). In practice, supporting the collection of measurement and management tasks will require simultaneously running an ensemble of sketch instances on the programmable switches.

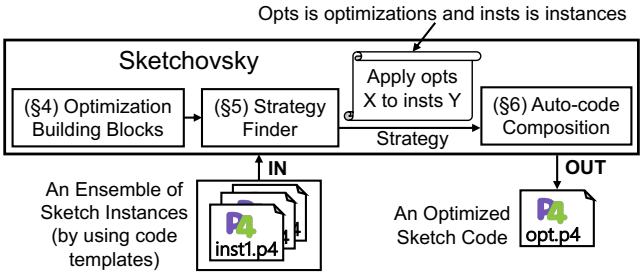


Figure 1: Sketchovsky Overview.

However, existing sketch-based telemetry efforts largely focus on running a *single* sketch instance on programmable switches and they cannot effectively support *an ensemble of sketch instances*. Naively extending a single sketch instance to support an ensemble of measurement tasks will require at least a *linear* increase in hardware resources (e.g., counters, hash units, pipeline stages), which is intractable as more measurement tasks are needed. While there have been some recent efforts on improving per-sketch efficiency (e.g., [4]), supporting P4 code composition [22, 23, 30, 42, 51], elastically trading of resource vs. accuracy (e.g., [3, 36]), and on improving the generality of sketches (e.g., [9, 15, 32, 49]), we find that these fundamentally fall short of efficiently supporting a general ensemble of sketch instances without sacrificing accuracy.

Given the limitations of current methods for running sketch ensembles, we present Sketchovsky (Sketch + Tchaikovsky), a composition framework that takes the input of sketch codes for the ensemble and outputs an optimized sketch code by leveraging cross-sketch optimizations (Fig. 1). Sketchovsky is complementary to the earlier work on implementing single-sketch algorithm more efficiently, developing more general-purpose sketches, and research that explicitly trades off accuracy reduction for resource reduction (§2). Indeed, using Sketchovsky can amplify their benefits by running expressive sketches (e.g., [32]) or extending per-sketch optimizations (e.g., [4]).

The design of Sketchovsky makes three key contributions:

```
1  packetStream
2  .map(p => (p.sIP, p.pktlen))
3  .reduce(keys=(sIP, ), f=sum)
4  .filter((sIP, count) => count > Th)
```

Query 1: Heavy hitters detection of srcIP written in Sonata [25].

```
1  packetStream
2  .map(p => (p.sIP, p.dIP, p.sPort, p.dPort, p.Proto))
3  .distinct()
```

Query 2: Distinct number of 5-tuple flows written in Sonata [25].

Optimization building blocks (§4): We observe that sketching algorithms have three common workflow steps: hash computations, counter updates, and heavy flowkey storage. We identify and formalize five novel cross-sketch optimization building blocks to *reuse* key hardware resources *across* sketch instances in each step. For *hash computations*, we show how and when (1) hash results can be reused across sketch instances or (2) can be reconstructed by cheap XOR operations. In the *counter updates* step, we discuss how (3) counter arrays can be reused or (4) can be co-located to reduce memory accesses. In *heavy flowkey storage*, we discuss (5) a novel mechanism to reuse the heavy flowkey storage by using the union of all flowkeys for sketch instances in the ensemble. Each optimization guarantees no accuracy loss.

Strategy finder (§5): Given an arbitrary sketch ensemble, there are many possible ways to use and combine the above building blocks. Naively solving this problem is intractable due to the challenges in modeling resource usage, identifying conflicts for combining optimizations, and the combinatorially large search space (e.g., it takes more than a day to solve). We identify practical relaxations to the problem and a greedy heuristic to make the problem tractable to solve. We show that our approach can quickly identify (e.g., in less than 1 second) an effective strategy that yields significant benefits.

Auto-code composition (§6): To simplify developer and operator effort, we design a simple-yet-effective switch-code generation process that realizes the selected strategy obtained above. We provide code templates of sketching algorithms to create sketch codes for an ensemble of sketch instances (Fig. 1) to enable this automatic code composition.[1]

We demonstrate the utility and benefits of Sketchovsky over a wide range of settings and a library of sketching algorithms that measure various statistics of interest [11, 14, 17, 19, 20, 21, 27, 28, 29, 32, 44]. Given this basic library, we built an ensemble generator that can create diverse ensembles using a wide range of parameters. We then used the generator to generate tens of thousands of ensembles that we used to measure the resource reduction benefits of Sketchovsky. We measure accuracy results by running four representative ensembles on the Tofino switch processing various inter-ISP packet traces [2]. Compared to the baseline of SketchLib, Sketchovsky reduces the use of hash units by 7~40%, SALUs by 9~45%, and SRAM by 0~7% for the ensembles that have the same flowkey for all sketch instances. Even for the ensembles with randomly picked sketching algorithms and parameters, resource reduction is 3~21% for the hash units, 4~26% for SALUs, and 0~0.4% for SRAM. For the accuracy experiment, we report no accuracy loss for any sketch instances.

[1] Sketchovsky is publicly available at https://github.com/Sketchovsky.

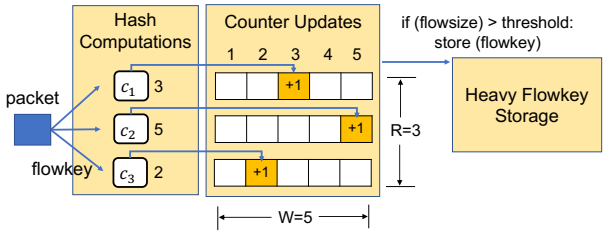


Figure 2: Sketching algorithms have three common workflow steps; hash computations, counter updates, and heavy flowkey storage.

2 Background and Related Work

We begin with background on sketches and their hardware footprints. Then we motivate the need for running an ensemble in practice. We end by explaining why state-of-the-art solutions fall short of effectively supporting an ensemble.

2.1 Sketches and Programmable Switches

Sketches or sketching algorithms on programmable switches are promising to support diverse measurement tasks due to resource efficiency and high accuracy. Sketching algorithms are randomized approximation algorithms designed to measure different statistics with a theoretical guarantee of high accuracy and sub-linear memory space in relation to the number of flows. Thus, sketching algorithms fit well for programmable switches with tight resource constraints. There are sketching algorithms to support diverse statistics for measurement tasks. For example, count-min sketch (CM) [17] can identify heavy hitters, and it can be configured with flowkey definition of srcIP to run Query. 1. HyperLogLog (HLL) [21] can estimate the distinct number of flows, and it can answer Query. 2 with flowkey definition of 5-tuple. There are other sketching algorithms, such as K-ary sketch (KARY) [27] for heavy change detection or UnivMon (UM) [32] for multiple statistics.

Sketching algorithms follow three common steps. We explain these steps with a canonical sketching algorithm count-min sketch [17] (Fig. 2). First, sketching algorithms perform *hash computations*. As each packet arrives, the count-min sketch extracts a flowkey (e.g., 5-tuple) from the packet header. On this key, the count-min sketch computes independent hash functions c_i. Second, using these hash results, sketching algorithms perform *counter updates*. They typically maintain 2D counter arrays, R independent counter arrays with the size of W, thus $R \times W$ counters in total. The hash result of c_i selects a specific column for counter update per row. The count-min sketch is a single-level sketching algorithm meaning that it maintains 2D counter arrays. There is a notion of a multi-level sketching algorithm, which uses multiple levels of 2D counter

arrays. Third, sketching algorithms need to maintain *heavy flowkey storage*. Sketching algorithms use threshold values to compare against flow size estimates to detect and store heavy flowkeys. While these steps run on the data plane, the control plane periodically reads and resets counter arrays and heavy flowkey storage to compute desired metrics [37].

While the ideas of Sketchovsky can be applied to other programmable switch architectures (as stated in §9), we showcase the effectiveness and benefits of Sketchovsky using Intel's Tofino switch [1]. Tofino is a commercially available programmable switch built on the RMT architecture [13] and the P4 language [12]. Its pipeline stage architecture is equipped with equal resources per stage, and running sketching algorithms on programmable switches requires the use of four key hardware resources. **Hash units** execute hash functions and there are a certain number of hash units per pipeline stage. The hash results of hash units can be used to select a specific column for counter updates or other purposes (§A.1). Each pipeline stage has a fixed amount of **SRAM** that can be used to maintain the state. SRAM is used by counter arrays and heavy flowkey storage. **Stateful ALU (SALU)** is the essential hardware resource that allows one read and one write operation to a register array in SRAM. A counter array for a sketch instance is mapped to a register array. Thus, a sketch instance using R counter arrays requires R SALUs. Storing heavy flowkey also requires SALUs. More **pipeline stages** are needed for more usage of the above three hardware resources. In addition, the notion of dependencies among workflow steps (e.g., *counter updates* must be executed earlier than *heavy flowkey storage*) can contribute to even more pipeline stage usage due to the imbalanced resource allocation.

2.2 Need for Ensemble of Sketch Instances

Network operators need to concurrently run diverse flow-level measurement tasks on programmable switches because the more information operators can get about the network, the more they can make the right management decisions [15, 25, 34, 38, 47, 48, 49, 52]. As concrete examples of measurement tasks, we show two network queries written by Sonata [25], a state-of-the-art query language on programmable switches. Query. 1 can detect heavy hitters based on the sum of packet length in bytes aggregated on flowkey of srcIP. Query. 2 measures the distinct number of 5-tuple flows. Network operators want to concurrently run these measurement tasks as many as possible.

Each such measurement task would entail creating a sketch instance based on a base sketching algorithm (SA) with four configurable parameters (Table 1): (1) **Flowkey** is any combination of packet header fields (e.g., srcIP or 5-tuple); (2) **Flowsize** defines whether the sketch instance keeps track of packet counts or packet bytes; (3) **Epoch** is the measurement time interval; and (4) **Resource Parameters** configure the memory size (e.g., W and R of 2D counter arrays). The network operator should choose resource parameters carefully due to a trade-off between resource use and accuracy.

SI	Base SA	Configurable Parameters			
		Flowkey	Flowsize	Epoch	Res. Param.
s_1	CM	(srcIP)	counts	10s	(3, 1024)
s_2	CM	(dstIP)	bytes	10s	(5, 2048)
s_3	KARY	(srcIP, dstIP)	counts	10s	(4, 4096)
s_4	HLL	(srcIP, srcPort)	-	5min	(1, 2048)
s_5	UM	(5-tuple)	counts	5min	(3, 2048, 16)

Table 1: An example of an ensemble of sketch instances. For resource parameters, (R, W) for single-level and (R, W, level) for multi-level sketching algorithms.

Solution	General	Resource	Accuracy
P4 Composition [22, 23, 30, 42, 51]	✓	X	✓
Per-sketch optimizations [4]	✓	X	✓
Expressive sketches [9, 15, 32, 49]	X	✓	✓
Dynamic resource allocation [3, 36, 50]	✓	✓	X
Sketchovsky (Our system)	✓	✓	✓

Table 2: Existing efforts cannot support a general ensemble of measurement tasks with low resource footprint and high accuracy.

For instance, given Query. 1 above, we can use a count-min sketch instance on the srcIP as flowkey and for Query. 2 we may use HyperLogLog on the 5-tuple. More generally, given the collection of measurement tasks with different configurable parameters (flowkey, flowsize, epoch) and statistics, we will need to concurrently run *an ensemble of sketch instances* in practice. For our work, we assume that the ensemble of sketch instances is given as input; the problem of finding the best ensemble of sketch instances given a collection of measurement tasks is outside the scope of this paper (§9).

2.3 Prior Work and Limitations

We discuss some existing efforts in sketch-based telemetry on programmable switches and why they are insufficient to tackle the ensemble of sketch instances problem (Table 2).

Composing P4 programs. Many P4 code composition works have been recently published for resource optimizations [22, 23, 30, 42, 51]. However, none of them can optimize the sketch ensemble because they did not consider stateful processing, which is at the core functionality of sketching algorithms (e.g., *counter update* step). P4visor [51], Lyra [22], and Cetus [30] focus on optimizations for match-action tables, but they did not consider optimizations for stateful processing, including MicroP4 [42]. Thus, they cannot be used to optimize an ensemble of sketch instances.

Chipmunk [23] seems to be a promising candidate for providing cross-sketch optimizations at first glance because it compiles a program written by Domino language into optimized P4 code with stateful processing optimizations. However, Chipmunk can not compile a full single sketch implementation due to its limited scope. It only supports the update part of the stateful value but does not include the addressing part (e.g., computing hash functions to address the column

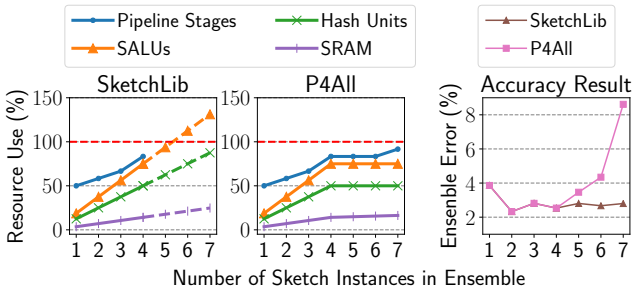


Figure 3: Existing efforts cannot efficiently run the ensemble.

index of counter arrays), which is critical for sketch implementations.

Per-sketch optimizations [4] can be used to implement the sketch ensemble. However, this approach cannot achieve low resource footprints due to *linearly* increasing resource consumption as we run multiple sketch instances.

More expressive sketches. To make improvements, recent advances in sketching theory empower a single sketch instance to support multiple measurement tasks [9, 15, 32, 49]. However, their coverage of the measurement tasks is still far from general (e.g., none of them can support the entropy estimation tasks for two different flowkey definitions).

Dynamic resource allocation. Earlier work has reduced resource use for the ensemble of sketch instances [3, 36, 50]. SCREAM [36] dynamically reduces resource parameters of sketch instances to meet specified minimum accuracy when there are variations in traffic. P4All [3] can be used to reduce the resource parameters of some sketch instances in the ensemble by identifying lower-prioritized sketch instances. FlyMon [50] enables dynamic parameter configuration at runtime (e.g., flowkeys and resource parameters). It essentially offers a time-sharing capability to run a sketch ensemble by switching out active sketch instances. However, all these techniques reduce resource use at the expense of accuracy. In contrast, our work proposes optimizations that reduce resources while maintaining accuracy for all sketch instances in the ensemble.

Quantitative results for existing efforts. We quantitatively show why existing efforts are insufficient. Fig. 3 shows the resource footprint and accuracy results for two approaches, per-sketch optimization (SketchLib) and dynamic resource allocation (P4All). To create the ensembles, we only use the count-min sketch with $(R, W) = (5, 8K)$, flowkey of 4-tuple, different measurement epochs, and different flowsize definitions. We use CAIDA traces [2]. For the P4All experiment, we fix the width of counter arrays and reduce the number of rows if necessary to fit the maximum number of SALUs on the Tofino switch. We treat all sketch instances equally in the objective function.

The results in Fig. 3 show that SketchLib cannot support more than four sketch instances. While P4All can support more than four sketch instances by reducing hardware resources, this also reduces the accuracy. In summary, we find that existing techniques cannot achieve both low resource footprint and high accuracy.

3 Sketchovsky Overview

Given that prior work is insufficient, we explore a *complementary* approach to identify and exploit *cross*-sketch optimizations to run an ensemble of sketch instances $\mathcal{S} = \{s_i\}_{i=1}^{N}$. To this end, we present Sketchovsky (Fig. 1), a novel cross-sketch optimization and composition framework. Sketchovsky identifies five cross-sketch optimization building blocks so that resource consumption increases *sub-linearly* in the number of sketch instances with guarantees of no accuracy loss. Sketchovsky uses efficient heuristics to find an effective strategy to combine these building blocks for a given ensemble and implements a module to automatically generates an optimized switch code.

Optimization building blocks (§4). We find that key hardware resources used in each workflow step of sketching algorithms can be *reused across* multiple sketch instances. We identify five optimization building blocks to reduce resource footprint while maintaining accuracy. O_{Hash1} and O_{Hash2} optimize the first step of hash computations; O_{Ctr1} and O_{Ctr2} optimize the second step of counter updates, and O_{Key} optimizes the third step of heavy flowkey storage. Note that optimizations can be generalized to other hardware (§9). Each O_j has applicable conditions to determine whether O_j can be applied to a subset of sketch instances $\mathbf{S} \subset \mathcal{S}$. Applicable conditions are expressed by *configurable parameters* introduced in (§2.2) (e.g., all $s_i \in \mathbf{S}$ have the same flowkey) and *sketch features*. The notion of sketch features captures the differences among different sketching algorithms in algorithm designs or data structures (e.g., counter array type, counter update type, or whether maintaining heavy flowkeys or not).

Strategy finder (§5). Among many valid strategies for applying five optimization building blocks to different subsets of sketch instances in the ensemble, it is challenging to quickly find the most effective strategy due to the intractably large search space. To solve this problem, we formulate an optimization problem by defining the objective function to minimize hardware resources. Next, we propose an idea of problem decomposition. We show that one large problem can be decomposed into small sub-problems, and separate solutions for sub-problems together produce the overall solution. To detect the validity of a strategy, the strategy finder takes inputs of sketch features (e.g., base sketching algorithm) and configurable parameters (e.g., flowkey and flowsize) for $\mathcal{S}$ as in Table 1. Optimization building blocks can be applied to a subset $\mathbf{S} \subset \mathcal{S}$ only if $\mathbf{S}$ satisfies the applicable conditions.

Auto-code composition (§6). Manually translating a strategy into an optimized code is challenging because the strategy contains information about the complicated interplay among multiple optimization building blocks and a set of sketch

Workflow Step	Optimizations	Reduction	Overhead
Hash Computations	HASHREUSE (O_{Hash1})	Hash unit	-
	HASHXOR (O_{Hash2})	Hash unit	Pipe Stages
Counter Updates	SALUREUSE (O_{Ctr1})	SALU,SRAM	-
	SALUMERGE (O_{Ctr2})	SALU	SRAM
Heavy Flowkey Storage	HFSREUSE (O_{Key})	SALU	Pipe Stages CP Comp

Table 3: Relationships among workflow steps, optimizations and resource reductions. CP Comp means Control Plane Computation, and Pipe Stages means Pipeline Stages.

Conditions	O_{Hash1}	O_{Hash2}	O_{Ctr1}	O_{Ctr2}	O_{Key}
Sketch Features					
C1. Same counter array type			✓	✓	
C2. Same counter update type			✓		
C3. Track heavy flowkey					✓
Configurable Parameters					
C4. Same flowkey definition	✓	*	✓	✓	
C5. Same flowsize definition			✓		
C6. Same measurement epoch			✓		

Table 4: Applicable conditions for five optimization building blocks.

instances. We build an auto-code composition that automatically translates a given strategy into optimized sketch code. This relieves the burden of manual work of network operators. Given the output of the strategy finder and a set of sketch P4 codes for $\mathcal{S}$, we generate a unified and optimized P4 program.

4 Optimization Building Blocks

Given $\mathcal{S} = \{s_i\}_{i=1}^{N}$, we identify five cross-sketch optimization building blocks (two for hash, two for counters, and one for flowkey storage) that can apply to a given set of sketch instances $\mathbf{S} = \{s_i\}_{i=1}^{n} \subset \mathcal{S}$. Table 3 summarizes relationships among workflow steps, optimizations and resource reductions. For each optimization, we explain the key idea, the conditions under which it applies, and its implications for resource use and accuracy. Table 4 summarizes applicable conditions to validate whether each optimization can be applied to $\mathbf{S} \subset \mathcal{S}$.

4.1 Hash Computations

To optimize the workflow step of *hash computations* (§2.1), we have two optimizations HASHREUSE (O_{Hash1}) and HASHXOR (O_{Hash2}). Hash unit refers to the hardware resource on programmable switches to execute hash functions. Hash result is the outcome hash value of the hash unit.

HASHREUSE (O_{Hash1}) Reusing hash results. If a set of sketch instances use the same definition of flowkey (e.g., srcIP), we can reuse hash results to reduce the usage of hash units. We explain this optimization using an example in Fig. 4. Assume we have a set of sketch instances $\mathbf{S} = \{s_i\}_{i=1}^{n}$ with a required set of independent hash results $\mathbf{E} = \{e_i\}_{i=1}^{n}$ and flowkey definition $\mathbf{F} = \{f_i\}_{i=1}^{n}$, which means that a sketch instance s_i needs e_i number of hash results based on flowkey f_i. Without optimization, $\sum_i e_i$ hash units are used. However,

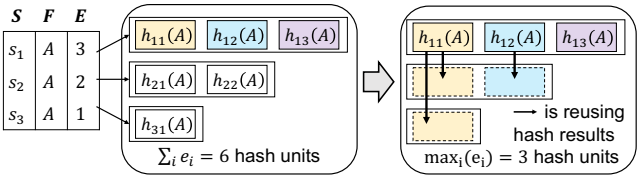


Figure 4: HASHREUSE (O_{Hash1}) reduces hash units by reusing hash results. A small box with $h_{seed}(flowkey)$ indicates one hash unit allocation.

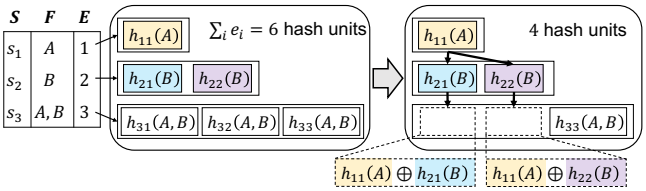


Figure 5: HASHXOR (O_{Hash2}) reduces hash units by using XOR.

we can reuse hash results, and we can reduce the allocation of hash units to $\max_i(e_i)$ on the hardware as in Fig. 4.

Applicability: Regardless of any sketching algorithms, we can apply this optimization as long as $\mathbf{S}$ uses the same flowkeys. We denote this as (C4) in Table 4.

Implication: Allocation of $\max_i(e_i)$ hash units is sufficient to preserve the accuracy of $\mathbf{S} = \{s_i\}_{i=1}^{n}$. The accuracy of sketch instances is closely related to hash independence among hash results. To implement hash independence in practice, randomly picked hash seeds are used; $h_{seed1}(A)$ and $h_{seed2}(A)$ are independent if $seed1 \neq seed2$. For a single sketch instance, hash independence among hash results is required. A key insight here is that hash independence is not required *across* sketch instances. Thus we can reuse $\max_i(e_i)$ hash results across sketch instances in a way that all hash results within any single sketch instance s_i are independent (e.g., in Fig. 4).

HASHXOR (O_{Hash2}) Less hashing, same performance with XOR-based reconstruction. We can reduce hash units even for a set of sketch instances with different flowkeys by leveraging XOR operations. We explain this optimization using an example in Fig. 5 where $\mathbf{S} = \{s_1, s_2, s_3\}$ and $\mathbf{F} = \{\{A\}, \{B\}, \{A,B\}\}$ and $\mathbf{E} = \{1,2,3\}$. A and B are different packet headers, such as A = srcIP and B = dstIP. We can reduce allocation of hash units by reconstructing independent hash results for s_3 as follows because $\{A,B\} = \{A\} \cup \{B\}$.

$$h_{31}(A,B) = h_{11}(A) \oplus h_{21}(B) \quad (1)$$

$$h_{32}(A,B) = h_{11}(A) \oplus h_{22}(B) \quad (2)$$

Note that XOR-based reconstructed hash results $h_{31}(A,B)$ and $h_{32}(A,B)$ are independent because $h_{21}(B)$ and $h_{22}(B)$ are independent. For arbitrary e_1 and e_2, we can reconstruct $e_1 \times e_2$ independent hash results for s_3.

Applicability: This optimization HASHXOR (O_{Hash2}) can be applied if $\mathbf{S}$ and $\mathbf{F}$ meet the following condition.

$$\text{For } \mathbf{S} \in \mathcal{S}, |\mathbf{S}| = 3 \text{ and } f_1 \cup f_2 = f_3 \quad (3)$$

This optimization can be applied as long as a set of sketch instances satisfies (3). Thus, we mark (*) at (C4) in Table 4 for O_{Hash2}.

Implications: This idea of XOR-based hash reconstruction is proven pairwise independent and has already been used in other contexts [26, 43]. Thus, accuracy will not be compromised, and our evaluation result confirms this. As a minor side effect, more pipeline stages might be needed by adding XOR operations in the sketch workflow. However, we will see in the evaluation that the impact of this overhead is small.

4.2 Counter Update

To optimize the second workflow step of *counter updates*, we have SALUREUSE (O_{Ctr1}) and SALUMERGE (O_{Ctr2}).

SALUREUSE (O_{Ctr1}) Reusing counter arrays (rows) across sketch instances. If all sketch instances in $\mathbf{S}$ meet certain applicable conditions, we can reuse counter arrays to reduce both SALUs and SRAM. We first see how this optimization works by looking at an example in Fig. 6, and we will describe applicable conditions later. Suppose $\mathbf{S} = \{s_1, s_2, s_3\}$ satisfies applicable conditions of O_{Ctr1} and $\mathbf{C} = \{(r_i, w_i)\}_{i=1}^{n}$ represent that s_i has r_i number of counter arrays with width w_i. Then, instead of updating three different sets of counter arrays for three sketch instances in the data plane, we can update only one set of counter arrays. Then, in the control plane, one set of counter arrays can be used to compute statistics for all three sketch instances. The way we compute the row and width of counter arrays for reuse is represented by $\mathbf{W}$:

$$\mathbf{W} = \{w_j^*\}_{j=1}^{\max_i(r_i)} \text{ where } w_j^* = \max_i\{w_i | r_i \geq j\} \qquad (4)$$

$\mathbf{W}$ represents width w_j^* per j-th counter array for reuse. Note that $\mathbf{W}$ can have different widths across counter arrays, and it does not affect the functionality of sketching algorithms. We can see that $\mathbf{W}$ has $\max_i(r_i)$ rows. Thus, we can reduce SALUs from $\sum_i r_i$ to $\max_i(r_i)$. Moreover, SRAM usage is reduced from $\sum_i r_i w_i$ to $\sum_j w_j^*$ and we show $\sum_i r_i w_i - \sum_j w_j^* \geq 0$ in §B.1. While the discussion focused on single-level sketch instances, the same idea also applies to multi-level sketch instances.

Implication: If we compare resource parameters (r_i, w_i) of any sketch instance s_i to counter arrays for reusing $\mathbf{W}$, $\mathbf{W}$ has the same or larger width. As a result, all sketch instances are guaranteed to achieve the same or improved accuracy.

Applicability: Applicable conditions for this optimization use two sketch features. The first sketch feature is **counter array type**. Sketching algorithms have different types of counter arrays; the single-level (SL) type has 2D counter arrays, and the multi-level (ML) type has multiple levels of 2D counter arrays. The second sketch feature is **counter update type**. Sketching algorithms have different ways of updating counters. It can be bitmaps (BITMAP) or integer counters that only add values (COUNTER). Refer §A.1 for a full list of five

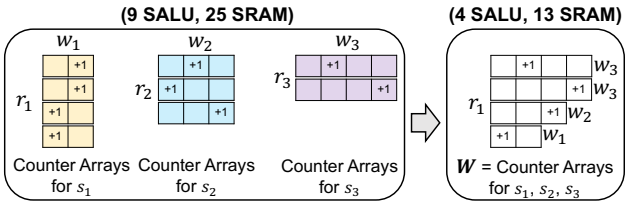


Figure 6: SALUREUSE (O_{Ctr1}) reuses counter arrays.

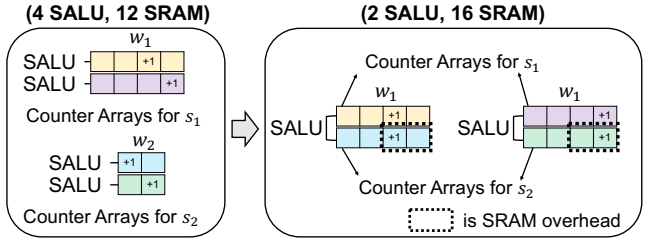


Figure 7: SALUMERGE (O_{Ctr2}) reduces SALUs by making SALUs update two counter arrays simultaneously.

counter update types. $\mathbf{S} \subset \mathcal{S}$ must satisfy five conditions to apply this optimization (Table 4): the same counter array type, the same counter update type, the same flowkey, the same flowsize, and the same epoch (C1, C2, C4, C5, C6).

SALUMERGE (O_{Ctr2}). Combining two counter updates into one SALU allocation. Leveraging the full capability of the underlying hardware resources can help resource reduction of $\mathbf{S}$. We observe that SALU can update two registers addressed in the same index and we can leverage this feature to update two counter arrays simultaneously. As a result, we can reduce SALUs by up to 2x. We explain this optimization by using an example in Fig. 7. We have two sketch instances with two counter arrays each, and we originally needed four SALUs. Then, we can make SALUs update two counter arrays simultaneously and reduce SALUs from 4 to 2.

We find two rules in the Tofino switch for a SALU to update two counter arrays. (R1) derives applicable conditions, and (R2) incurs SRAM overhead.

- (R1) Column indexes for counter updates are the same
- (R2) Two counter arrays have the same width

Applicability: (R1) derives two applicable conditions (C1, C4). If sketch instances use the same counter array type (C1) (e.g., sketch instances are either all single-level or all multi-level) and use the same definition of flowkey (C4), we can apply this optimization. Because flowkeys are the same, we can leverage HASHREUSE (O_{Hash1}), and SALU can update two counter arrays using the reused hash result for the column index. If flowkeys are different, then column indexes for the counter update can not be the same, which will violate (R1). Note that we should not let SALU update two counter arrays within a sketch instance because updating the same column index will lose hash independence and degrade accuracy.

Implication: This optimization can incur the additional SRAM overhead due to (R2). Suppose we have two sketch

instances $\{s_1, s_2\}$ with two counter arrays each as in Fig. 7 with width of $\{w_1, w_2\}$ s.t. $w_1 > w_2$. Suppose we can apply the optimization to $\{s_1, s_2\}$. Then, we should pick the longer width w_1 for counter arrays to preserve the accuracy of both $\{s_1, s_2\}$. As a result, the accuracy will be maintained (e.g., for s_1) or improved (e.g., for s_2). However, this will incur an SRAM overhead of $w_2 - w_1$ for s_2, as marked in Fig. 7. Despite the SRAM overhead, we argue that this optimization is still effective and practical for three reasons. First, increased SRAM is not wasted but will improve accuracy. Second, the overall SRAM overhead is bounded by 2x ($\frac{2\,max(w_1,w_2)}{w_1+w_2} \leq 2$). Third, SRAM is not the imminent bottleneck as we will see in the evaluation.

4.3 Heavy Flowkey Storage

To optimize the third workflow step of *heavy flowkey storage*, we have one optimization HFSREUSE (O_{Key}).

HFSREUSE (O_{Key}). Reusing heavy flowkey storage across sketch instances. A large portion of sketching algorithms store heavy flowkeys in the switch data plane [9, 14, 16, 17, 27, 32, 41]. We can reduce the usage of SALUs (the hardware used for memory accesses) by reusing heavy flowkey storage across sketch instances. If multiple sketch instances have the same definition of flowkey (e.g., srcIP), we can store heavy flowkey in one heavy flowkey storage to save SALUs. We can generalize this idea to sketch instances with different definitions of flowkey using the notion of *union-key*. Suppose we have two sketch instances $\mathbf{S} = \{s_1, s_2\}$ with two different flowkey definitions $\mathbf{F} = \{\{\text{srcIP}\}, \{\text{dstIP}\}\}$. Then, instead of maintaining two heavy flowkey storage, we use one flowkey storage using union-key of {srcIP, dstIP} where union-key can be computed by ($UK = \cup_i f_i$). Then, for a given packet, if *either* {srcIP} is identified as a heavy flowkey for s_1 *or* {dstIP} is identified as a heavy flowkey for s_2. We store {srcIP, dstIP} of the given packet in the heavy flowkey storage.

We can do a further optimization to reduce memory usage of heavy flowkey storage. Suppose $\mathbf{S} = \{s_1, s_2\}$ and $\mathbf{F} = \{\{\text{srcIP}\}, \{\text{dstIP}\}\}$. For a given packet, if {srcIP} is identified as a heavy flowkey whereas {dstIP} is not, we store {srcIP, 0} so that the control plane knows this flowkey is only for s_1. To generalize this idea to multiple sketch instances, we can compute a *conditional union-key* $UK_C = \cup_j f_j$ where (flow size estimate)$_j$ > $threshold_j$ and set 0 to ($UK - UK_C$) when we store heavy flowkey into the storage.

Applicability: We can apply this optimization to a set of sketch instances $\mathbf{S}$ if all sketch instances in $\mathbf{S}$ tracks heavy flowkeys (C3) as in Table 4. For different measurement epochs, we can compute the greatest common denominator (GCD) among all epochs, and the control plane can retrieve heavy flowkeys every time period of GCD. For example, if there are sketch instances with 10s, 20s, and 30s measurement epochs, the control plane retrieves heavy flowkeys for every 10s, and we can reconstruct heavy flowkeys for sketch instances of 20s and 30s.

Implication: By storing fine-grained heavy flowkeys by union-key, the control plane can retrieve heavy flowkeys for individual sketch instances by aggregation without missing any heavy flowkeys. This optimization incurs small additional computations on the switch control plane. However, this overhead does not affect the overall performance because this control plane computation is not on the critical path to provide measurement results. While the switch data plane updates the counter arrays, the switch control plane can independently execute heavy flowkey aggregation on the CPU. Another small overhead of the pipeline stage can occur, but we will see in the evaluation that the impact is small.

5 Strategy Finder

In the previous section, we proposed five optimizations $\{O_j\}_{j \in \{\text{Hash1, Hash2, Ctr1, Ctr2, Key}\}}$ and their applicable conditions to a subset of sketch instances $\mathbf{S} = \{s_i\}_{i=1}^{n} \subset \mathcal{S}$. In this section, we aim to develop a strategy finder that partitions $\mathcal{S}$ into the best applicable subsets so that five optimization building blocks can produce the maximum benefit for any given ensemble $\mathcal{S}$.

5.1 Problem Formulation

We formulate an optimization problem to find the optimal strategy. We consider partitions of $\mathcal{S}$ because each optimization O_j is applied to disjoint subsets of $\mathcal{S}$. Suppose $\mathcal{P}_\mathcal{S} = \{P_k | P_k \text{ is k}^{th} \text{ partition of the set } \mathcal{S}\}$ is a set containing all partitions of the set $\mathcal{S}$ where $P_k = \{\mathbf{S}_l \subset \mathcal{S} | \cup_l \mathbf{S}_l = \mathcal{S}\}$. The goal is to find the optimal strategy X^*, which minimizes hardware resources while satisfying the applicable conditions:

$$\min_X \; HwResource(X) \tag{5}$$

$$\text{s.t.} \sum_{k=1}^{|\mathcal{P}_\mathcal{S}|} x_{jk} = 1, \forall\, j \in \{\text{Hash1, Hash2, Ctr1, Ctr2, Key}\} \tag{6}$$

$$Valid(X) = 1 \tag{7}$$

The decision variable is $X = \{X_j\}_{j \in \{\text{Hash1, Hash2, Ctr1, Ctr2, Key}\}}$. X_j selects a partition $P_k \in \mathcal{P}_\mathcal{S}$ for O_j so that O_j is applied to all subsets $\in P_k$. To express this, we define $X_j = \{x_{jk} | x_{jk} \in \{0,1\}\}_{k \in \{1,\ldots,|\mathcal{P}_\mathcal{S}|\}}$ and $x_{jk} = 1$ if P_k is selected. Note that (6) makes X_j pick only one partition P_k for O_j. About constraint (7), we use $Valid(X) \in \{0,1\}$ to denote whether strategy X is *valid* or not. X is valid if all subsets $\in P_k$ satisfy the applicable conditions of O_j for $\forall\ j$. It is assumed that applicable conditions are met for the subset $\mathbf{S} \subset \mathcal{S}$ containing a single sketch instance s.t. $|\mathbf{S}| = 1$. For objective function (5), we aim to find a strategy X^* that minimizes hardware resource among all valid strategies X. To model this objective function $HwResource(X)$, we use the linear combination of four key

resource usage:

$$LinearComb(X,R) = \sum_{r \in R} a_r \cdot resource_r(X) \quad (8)$$

$$R = \{\text{SALU, HashUnit, SRAM, PipelineStage}\} \quad (9)$$

Network operators can use (8) and customize the objective function by choosing different coefficient sets $\{a_r\}_{r \in R}$ for their preference. For example, suppose network operators desire to reduce SRAM more than other resources to run the ensemble with memory-intensive applications, they can increase the weight for a_{SRAM} in (8).

5.2 Challenges

We face three challenges in formulating and solving the optimization problem.

C-1. Large search space. We have a large search space for enumeration because the number of possible partition $|\mathcal{P}_\mathcal{S}|$ increases exponentially as the number of sketch instances $|\mathcal{S}|$ increases [8]. Even after we consider constraint (6), the decision variable X has $|\mathcal{P}_\mathcal{S}|^5$ combinations because X selects five partitions among $\mathcal{P}_\mathcal{S}$. This large search space makes finding the optimal solution X^* become intractable. In practice, operators often need to reconfigure sketch ensembles, and waiting for the optimization to complete may end up being on the critical path [50].

C-2. Modeling the valid function. It is hard to define $Valid(X)$ due to the dependencies among optimizations. Specifically, there exist dependencies between O_{w1} and O_{w2} for $w \in \{\text{Hash, Ctr}\}$ because they are applied to the same workflow steps. Thus, it is unclear whether a sketch instance s_i can be benefited from O_{w1} and O_{w2} at the same time. Further, if they can, then it is also unclear how to figure out the relationship between X_{w1} and X_{w2} to detect the validity of X to define $Valid(X)$.

C-3. Modeling the objective function. We find that computing $LinearComb(X,R)$ takes a long time because accurately measuring pipeline stage usage requires the compilation of an optimized P4 code by applying strategy X. The execution time for $resource_{pipeline_stage}(X)$ takes several minutes. This delay will significantly impede the search process, and finding a solution X^* can become even more intractable.

5.3 Our Approach

Next, we reformulate the problem and show that finding the optimal strategies for each O_j will create the overall solution X^*. This reduces search space significantly and makes the problem tractable.

Excluding pipeline stage from the objective function. To handle (C-3), we make a pragmatic choice of excluding the pipeline stage from the objective function. We use $LinearComb(X,R')$ as objective function where $R' = \{\text{SALU, HashUnit, SRAM}\}$. $resource_r(X)$ for $r \in R'$ can be quickly computed because X contains information about the number of reused or XOR-reconstructed hash units, reused or co-located counter arrays, and reused heavy flowkey storage. The impact of this decision cannot be measured because the optimal objective function is unknown and difficult to define. However, we can still identify effective solutions that can yield significant benefits in practice because there is a correlation between the resource reduction on R' and the pipeline stage reduction (§8).

Search space decomposition *across* workflow steps. To overcome the challenge of large search space (C-1), we can decompose the optimization problem into three sub-problems, and solution X^* can be achieved by solving sub-problems separately. Specifically, we decompose the decision variable X into three groups corresponding to three workflow steps:

$$X_{Hash} = \{X_{Hash1}, X_{Hash2}\}, X_{Ctr} = \{X_{Ctr1}, X_{Ctr2}\}, X_{KEY} = \{X_{Key}\}$$

Then, we can also decompose the valid function and the objective function as follows:

$$X = \cup_{w \in \{\text{Hash, Ctr, KEY}\}} X_w \quad (10)$$

$$Valid(X) = \prod_{w \in \{\text{Hash, Ctr, KEY}\}} Valid(X_w) \quad (11)$$

$$HwResource(X) = \sum_{w \in \{\text{Hash, Ctr, KEY}\}} HwResource(X_w) \quad (12)$$

This problem decomposition is possible for two reasons. First, although there are dependencies in terms of applicability within X_w, there are no dependencies *across* X_w because optimizations are independently applied to different workflow steps. Thus, $Valid(X)$ can be achieved by multiplication of decomposed $Valid(X_w)$ as in (11). Second, $HwResource(X)$ can be achieved by summation of decomposed $HwResource(X_w)$ as in (12). Without the idea of excluding pipeline stage usage, this linearity property (12) does not hold because measuring pipeline stage usage must consider the overall table dependency graph (TDG) among workflow steps (X_w). As a result, a solution X^* can be achieved by $X^* = \{X^*_{Hash}, X^*_{Ctr}, X^*_{KEY}\}$ where X^*_w is a solution of each sub-problem for $w \in \{\text{Hash, Ctr, KEY}\}$ as follows:

$$\min_{X_w} HwResource(X_w) \text{ s.t. } Valid(X_w) = 1 \quad (13)$$

Two-step enumeration for X_{Hash} and X_{Ctr}. Although we can decompose $Valid(X)$ into three $Valid(X_w)$ as in (11), it is still unclear how to realize $Valid(X_w)$ for $w \in \{\text{Hash, Ctr}\}$ because there exist dependencies between O_{w1} and O_{w2}. We can solve this problem using an enumeration technique that efficiently explores the search space. Suppose the enumeration does not miss out on *valid* X_w (s.t. $Valid(X_w) = 1$) while efficiently skips *invalid* X_w. In that case, it will help to solve not only the challenge (C-2) of modeling an valid function but also the challenge (C-1) by reducing search space. To achieve this, we develop a two-step enumeration technique as in Alg. 1.

Algorithm 1 TwoStepEnumeration

```
1:  procedure TwoStepEnumeration(S, w)
2:      P_S = {P_k | P_k is k^th partition of the set S}
3:      min ← INTMAX
4:      for X_w1 s.t. selected P_w1 ∈ P_S is valid do
5:          for X_w2 s.t. P_w1 ≤ P_w2 ∈ P_S do
6:              X_w ← {X_w1, X_w2}
7:              P_w12 = NestedPartition(P_w1, P_w2)
8:              if P_w12 is valid then
9:                  if min > HwResource(X_w) then
10:                     min ← HwResource(X_w)
11:                     X*_w ← X_w
12:     return X*_w
```

We explain this algorithm by both cases of $w \in \{\text{Hash, Ctr}\}$. Let's first see an example for w = Hash, HashReuse (O_{Hash1}) and HashXOR (O_{Hash2}). Suppose we have five sketch instances $\mathcal{S} = \{s_i\}_{i=1}^{5}$ with flowkey definition $\mathbf{F}$ ={{srcIP}, {srcIP}, {dstIP}, {srcIP, dstIP}, {srcPort}}. Then the algorithm enumerates all valid P_{w1} at line 4 in Alg. 1. P_{w1} = $\{\{s_1, s_2\}, \{s_3\}, \{s_4\}, \{s_5\}\}$ can be picked because $\{s_1, s_2\}$ have the same flowkey so that we can reuse hash results to reduce hash units. Next, given picked P_{w1}, it enumerates P_{w2} s.t. $P_{w1} \leq P_{w2}$[2] to create *nested partition* P_{w12} using P_{w1} and P_{w2}. If $P_{w2} = \{\{s_1, s_2, s_3, s_4\}, \{s_5\}\}$ is picked, then the nested partition is $P_{w12} = \{\{\{s_1, s_2\}, \{s_3\}, \{s_4\}\}, \{\{s_5\}\}\}$. To see the validity of P_{w12}, check whether all subsets in P_{w12} satisfy applicable conditions of O_{Hash2} at line 8. Picked P_{w12} is valid because subset $\mathbf{S} = \{\{s_1, s_2\}, \{s_3\}, \{s_4\}\} \in P_{w12}$ satisfies applicable conditions of $|\mathbf{S}| = 3$ and {srcIP} ∪ {dstIP} = {srcIP, dstIP} as in (3) at §4.1. Note that $\{s_1, s_2\}$ is handled as if it is a single sketch instance with a flowkey of {srcIP}.

Interestingly, the same algorithm works for w = Ctr, SaluReuse (O_{Ctr1}) and SaluMerge (O_{Ctr2}). First, the algorithm enumerates P_{w1} where all subsets in P_{w1} satisfy applicable conditions (C1, C2, C4, C5, C6) of O_{Ctr1}. For picked P_{w1}, O_{Ctr1} is applied to all subsets $\in P_{w1}$, meaning that each subset has one set of counter arrays for reuse $\mathbf{W}$ as discussed as in (3) at §4.2. Then each subset can be handled as if it is a single sketch instance with counter arrays configured with $\mathbf{W}$. Next, we can detect the validity of nested partition P_{w12} using the applicable conditions (C1, C4) of O_{Ctr2} at line 8 in Alg. 1.

HfsReuse (O_{Key}) does not need this two-step enumeration. The solution for O_{Key} is one subset $\mathbf{S}$ containing all sketch instances that track heavy flowkey because this will minimize the hardware resource usage.

Search space decomposition *within* workflow steps. Although two-step enumeration reduces search space by picking P_{w1} first and then P_{w2} such that $P_{w1} \leq P_{w2}$, this enumeration technique still takes a long time to finish (e.g., more than a day). To this end, we come up with an idea to decompose

[2]If every element of partition P_{w1} is a subset of some element of partition P_{w2}, then $P_{w1} \leq P_{w2}$. In other words, P_{w1} is finer and P_{w2} is coarser.

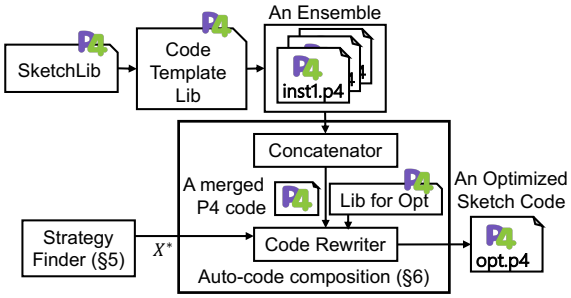


Figure 8: Overview of auto-code composition.

```
01: /* 1. hash computation step */
02: s#_h = HashUnit(seed1, FLOWKEY);
03: s#_value = TCAM_LPM(s#_h);
04: /* 2. counter update step */
05: CounterUpdate(seed2, FLOWKEY, WIDTH,
06:                     SL, PCSA, s1_value);
07: /* 3. heavy flowkey storage step - no code */
```

Figure 9: Code template example for PCSA.

X_{w1} and X_{w2} by using a greedy heuristic algorithm. Instead of running a nested loop (lines 4-5 in Alg. 1) for finding P_{w1} and P_{w2}, we can first find the optimal P^*_{w1} given $\mathcal{S}$ and then finds P^*_{w2} based on already fixed P^*_{w1}. This greedy heuristic algorithm decomposes the search space of $\{X_{w1}, X_{w2}\}$ into separate $\{X_{w1}\}$ and $\{X_{w2}\}$.

The insight behind this greedy heuristic algorithm comes from the applicability-benefit trade-off between O_{w1} and O_{w2}. O_{w1} is more difficult to apply but has a high resource reduction benefit. O_{w2} is easier to apply but has a low resource benefit. Thus, it makes sense that the algorithm first applies O_{w1} as much as possible, then next applies O_{w2}. We can not prove whether this greedy heuristic algorithm can find the same or close solution compared to the two-step enumeration. However, we empirically show that the overhead of objective function increase is small (e.g., less than 2%) while solving time of the greedy heuristic algorithm is more than three orders of magnitude faster (§D.3).

6 Auto-Code Composition

Using solution X^* from the strategy finder, we need two steps to generate an optimized P4 code for $\mathcal{S}$ as in Fig. 8.

6.1 Sketch P4 Codes and Concatenation

The first step requires network operators to provide N sketch P4 codes that should match with sketch features and configurable parameters for the ensemble $\mathcal{S} = \{s_i\}_{i=1}^{N}$ (e.g., Table 1).

Code template library. Writing N sketch P4 codes from scratch is a cumbersome task for network operators. An effective way is to provide code templates of the sketching algorithm with which P4 code for a sketch instance can be created. We build code templates for sketching algorithms so that network operators can configure the template with tunable parameters. Fig. 9 shows a code template example of

one sketching algorithm PCSA [20]. Network operators can fill out placeholders using configurable parameters.

SketchLib. To further simplify the code template, we consider using a common library to write codes for sketch instances, such as SketchLib [4]. The idea of using API calls makes code templates simple and concise. We extend SketchLib to enable the flexible configuration of various sketch features and configurable parameters. For example, API call `CounterUpdate()` in Fig. 9 at line 5 gets any definition of flowkey and any counter update type (e.g., `PCSA` type is used in this example). A full list of extended API calls of SketchLib is in §C.1.

Code concatenation. Finally, the concatenator in Fig. 8 gets the input of N sketch P4 codes created from code templates and concatenates N sketch P4 codes into one merged P4 code.

6.2 Code Rewriting

The second step is code rewriting to translate the selected strategy X^* into optimized code. Code rewriter in Fig. 8 gets three inputs; a merged P4 code from the first step, strategy X^* from the strategy finder, and Library for Optimization (Lib for Opt) that is used to apply SALUMERGE (O_{Ctr2}). Using $X^* = \{X^*_{Hash}, X^*_{Ctr}, X^*_{KEY}\}$, the code rewriter sequentially translates X^*_w to each workflow step in a merged P4 code by rewriting short lines of code. Leveraging the code templates makes the code rewriting process a lot easier. First, a merged sketch P4 code is structured in a way that the code rewriter can easily parse and apply optimizations. Second, the amount of code rewrite is minimized because sketch code templates are concise by using API calls in SketchLib and Lib for Opt.

7 Implementation

Auto-code composition. We use two examples, O_{Hash1} and O_{Hash2} to illustrate how we auto-generate an optimized code. Fig. 10 is the code snippet without optimization and we call it before code. Fig. 11 is the code snippet after applying X^*_{Hash} and we call it after code. We have $\mathcal{S} = \{s_i\}_{i=1}^4$, $\mathbf{F}$ ={{srcIP}, {srcIP}, {dstIP}, {srcIP, dstIP}}. The before code allocates hash units to generate hash results for each flowkey (lines 3, 7, 10, 13 in Fig. 10). To emulate different logical hash seeds for independence, we configure the hash units with different CRC polynomials in practice. Then, we apply optimizations using a given solution $X^*_{Hash} = \{\{\{s_1, s_2\}, \{s_3\}, \{s_4\}\}\}$, which means the code should reuse {srcIP} for $\{s_1, s_2\}$ and use XOR operation to create a hash result for {srcIP, dstIP} = {srcIP} $\oplus$ {dstIP}. If we look at line 4 in Fig. 11, the hash result of s_2 reuses the hash result of s_1. Line 6 in Fig. 11 shows XOR-based hash result reconstruction. As a result, the usage of the hash unit is reduced from 4 to 2.

Applying SALUMERGE (O_{Ctr2}) requires new codes for implementing two counter arrays to share one SALU that the before code does not have. Thus, we build a new library (Lib for Opt) shown in Fig. 8 to implement O_{Ctr2} and the code rewriter can use this library for applying O_{Ctr2}. The definition of the API call for Lib for Opt is in §C.1. For other optimizations $\{O_j\}_{j \in \{Hash1, Hash2, Ctr1, Key\}}$, we do not need new API calls because a simple rewrite is enough for implementing reusing resources (O_{Hash1}, O_{Hash2}, O_{Ctr1}) or XOR operation (O_{Hash2}) (e.g., at line 6 in Fig. 11). As a result, the code rewriter can translate all five optimizations into an optimized code. We show examples of before and after code snippets for O_{Hash1}, O_{Ctr1}, O_{Key} in §C.2.

```
01: // code for s1
02: /* 1. hash computation step */
03: s1_h = HashUnit(seed1, srcIP);
04: ... /* 2. counter update step */
05: ... /* 3. heavy flowkey storage step */
06: // code for s2
07: s2_h = HashUnit(seed2, srcIP);
08: ...
09: // code for s3
10: s3_h = HashUnit(seed3, dstIP);
11: ...
12: // code for s4
13: s4_h = HashUnit(seed4, srcIP, dstIP);
14: ...
```

Figure 10: [Before] HASHREUSE (O_{Hash1}) and HASHXOR (O_{Hash2}).

```
01: // code for s1, s2, s3
02: /* 1. hash computation step */
03: s1_h = HashUnit(seed1, srcIP);
04: s2_h = s1_h;
05: s3_h = HashUnit(seed3, dstIP);
06: s4_h = s1_h ^ s3_h;
07: ...
```

Figure 11: [After] HASHREUSE (O_{Hash1}) to $\{s_1, s_2\}$ and HASHXOR (O_{Hash2}) to $\{\{s_1, s_2\}, s_3, s_4\}$.

Strategy finder. One minor issue here is that while implementing SALUMERGE (O_{Ctr2}) on the Tofino switch, we face a known problem of sketch inaccuracy caused by the counter read and reset delays [37]. To address this, we add one more applicable condition of the same epoch (C6) to O_{Ctr2}.

8 Evaluation

Our extensive set of experiments shows that Sketchovsky can achieve both a low resource footprint and high accuracy simultaneously.

8.1 Experimental Setup

Testbed. We evaluate Sketchovsky on a local testbed with an Edgecore Wedge 100BF Tofino-based programmable switch and a server equipped with dual Intel Xeon Silver 4110 CPUs, 128GB RAM, and a 100Gbps Mellanox CX-4 NIC connected to the switch. We use the P4-16 version with the Tofino SDE version of 9.5.1.

Sketching algorithms. We use eleven sketch algorithms that measure six different statistics.[3] Although Bloom filter (BF) is not a sketching algorithm, we include BF because it also follows the workflow steps of sketching algorithms, and Sketchovsky can optimize it.

Four ensemble types. We use four types of ensembles that network operators would practically consider using. In ensembles of (Type 1. Same Sketch), (Type 2. Same Flowkey), and (Type 3. Same Epoch), all sketch instances in the ensemble use the same sketching algorithm, flowkey, and epoch, respectively. For (Type 4. Random), sketch instances in an ensemble are picked randomly.

Ensemble Generator. To create four types of ensembles, we build an ensemble generator that takes two inputs; (1) the ensemble type and (2) the number of sketch instances for the ensemble. Using these two inputs, the ensemble generator randomly picks sketching algorithms and assigns configurable parameters from a large pool of candidates. A full list of candidates for parameters is in §D.1. The ensemble generator does not allow any two sketch instances in an ensemble to have the same statistic, flowkey, flowsize, and epoch.

Metrics. We use three metrics for accuracy: (1) Relative Error (RE): $\frac{|True-Estimate|}{True}$, where *True* is the ground truth value and *Estimate* is the estimated value. We use this metric for sketching algorithms for cardinality and entropy. (2) Average Relative Error (ARE): $\frac{1}{k}\sum_{i=1}^{k}\frac{|f_i-\hat{f}_i|}{f_i}$, where k means the top k heavy flows. f_i is the actual flow size for flow i, and $\hat{f}_i$ is the estimated flow size from the sketch instances. This metric is used to evaluate the accuracy of the heavy hitter and heavy change detection. We use k=50. (3) Weighted Mean Relative Difference (WMRD) is used for MRAC [28].

For resource reduction, we use two metrics: (1) Resource Usage (RU): $\frac{\text{Used}}{\text{Available}}$, where Used is the amount of resource used for the ensemble and Available is the total amount of available resource on the switch; and (2) Resource Reduction (RR): $\frac{\text{RU (before) - RU (after)}}{\text{RU (before)}}$, where RU (before) is the amount of used resource before applying optimizations of Sketchovsky and RU (after) is the amount of used resource after optimization.

8.2 Accuracy

We show that Sketchovsky does not degrade accuracy and sometimes improves accuracy. For this experiment, we picked four ensembles from each ensemble type. A full list of base sketch algorithms and configurable parameters for picked ensembles is in §D.2. Given each ensemble as input, we generate sketch P4 codes for the Tofino switch both before and after we use Sketchovsky for optimization. All five optimizations are enabled. We then run sketch P4 codes for (four picked ensembles) × (before and after optimizations) on the Tofino switch and compare the accuracy of the sketch instances. We use five traffic workloads of inter-ISP packet traces collected on different dates.[4] For each traffic workload, we send ten 60s packet traces from a directly connected server to the Tofino switch using tcpreplay at full speed.

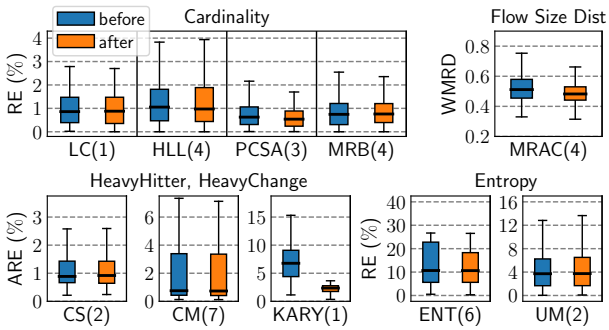


Figure 12: Overall accuracy evaluation.

Fig. 12 shows the overall accuracy results. We grouped sketch instances into four different statistics based on the sketching algorithm used. The X-axis in Fig. 12 shows the number of sketch instances with the same sketching algorithm (e.g., HLL(4) means there are four sketch instances using the sketching algorithm of HLL). The Y-axis shows the quartiles of errors for sketch instances. We see that none of the sketch instances lose accuracy after optimization. In fact, we observe some accuracy improvements. Thanks to O_{Ctr1}, counter arrays of KARY are increased from 1 to 3, and the error is reduced. In addition, we do not miss any heavy flowkeys both before and after optimization. Because BF does not produce measurement results, the accuracy result for two BF sketch instances is not shown.

8.3 Resource Reduction

Sketchovsky makes infeasible ensembles feasible. We show the resource reduction benefits of using Sketchovsky. For this experiment, we generate a total of 400 ensembles of sketch instances; (four ensemble types) × (10 different numbers of sketch instances from 2, 4, ..., 20) × (10 different ensembles). Then, we run Sketchovsky to produce 400 sketch P4 codes both before and after optimization.[5] Next, we compile the codes using the Tofino compiler to check the feasibility. To make the experiment more realistic, we append codes for L2 switching, L3 routing, and access control list (ACL) to all of the before and after optimization codes. L2 and ACL consume 55% of on-switch SRAM in total, and L3 uses 63% of TCAM.

[3] Linear counting (LC) [44], HyperLogLog (HLL) [21], PCSA [20], multi-resolution bitmap (MRB) [19] measure cardinality. Count-sketch (CS) [14], count-min sketch (CM) [17] can detect heavy hitters, and K-ary sketch (KARY) [27] can detect heavy change. Entropy sketch (ENT) [29] measures entropy, MRAC [28] measures flow size distribution (FSD). UnivMon (UM) [32] can measure general statistics. Bloom filter (BF) [11] can do the membership test. A full list of sketching algorithms that we used for our experiments with sketch features is in §D.1.

[4] We use five CAIDA backbone traces captured in 3/20/14 Sanjose, 6/19/14 Sanjose, 1/21/16 Chicago, 5/17/18 NYC, and 8/16/18 NYC [2]

[5] We use count-min sketches to create ensembles for (Ensemble Type 1) because it is one of the most popular and widely-used sketching algorithms.

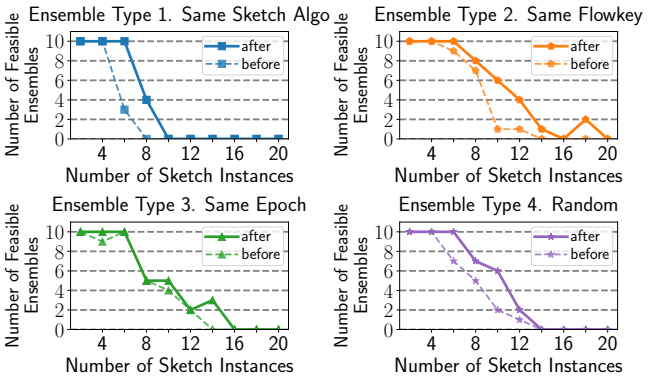


Figure 13: Feasibility comparison of ensembles before vs after.

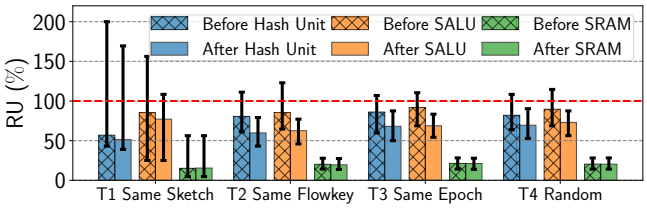


Figure 14: Resource usage comparison before vs after for the number of sketch instances = 12.

The X-axis in Fig. 13 is the number of sketch instances in the ensemble. The Y-axis is the number of feasible ensembles among ten ensembles per different number of sketch instances. The result shows that 42 ensembles that were previously infeasible become feasible with Sketchovsky. For example, if we look at "Ensemble Type 1" and "6 sketch instances", only 3 out of 10 ensembles were feasible before optimization. However, all ten ensembles become feasible after optimization. Note that the pipeline stage overhead of O_{Hash2} and O_{Key} does not negatively impact feasibility after applying them.

Resource usage before and after optimization. Fig. 14 shows the use of individual resources before and after optimization. Using the ensemble generator, we generated 1200 ensembles; (four ensemble types) × (300 different ensembles). Each ensemble has 12 sketch instances. Because some ensembles are not feasible on the Tofino switch because of the limited number of stages, we calculated resource use using the strategy finder so we are not limited by, and do not show, pipeline stages. We cross-checked the resource use between the strategy finder and the Tofino compiler for feasible ensembles. Each bar in Fig. 14 shows the median value, and the error line shows the 10% and 90% percentile among 300 ensembles. The red-dotted line shows the total available resources on the switch, so values above the red line represent infeasible ensembles. Fig. 14 visually shows how previously infeasible ensembles become feasible. SRAM usage of heavy flowkey storage before and after optimization is similar because heavy flowkeys overlap across sketch instances.

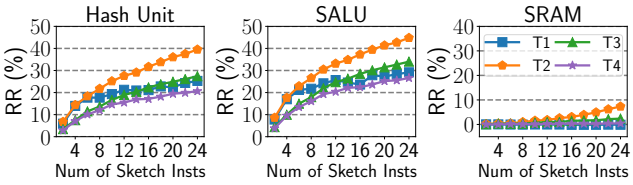


Figure 15: Resource reduction result.

	Resources	Total	O_{Hash1}	O_{Hash2}	O_{Ctr1}	O_{Ctr2}	O_{Key}
Type 1	Hash Unit	21.3	3.1	0.1			18.1
Same	SALU	25.7			-	0.8	24.9
Sketch	SRAM	-0.02			-	-0.02	
Type 2	Hash Unit	27.6	10.4	-			17.2
Same	SALU	33.1			3.8	5.9	23.4
Flowkey	SRAM	1.8			2.3	-0.5	
Type 3	Hash Unit	18.9	5.5	0.04			13.4
Same	SALU	24.7			2.2	3.7	18.8
Epoch	SRAM	0.9			1.3	-0.4	
Type 4	Hash Unit	15.5	1.9	0.04			13.6
Random	SALU	20.4			0.5	1.0	18.9
	SRAM	0.2			0.3	-0.1	

Table 5: Breakdown of resource reduction by each optimization for the number of sketch instances = 12.

Sensitivity analysis on the number of sketch instances. We show a more detailed view of resource reduction by looking at ensembles with different numbers of sketch instances. We generate (four ensemble types) × (12 different numbers of sketch instances from 2, 4, ..., 24) × (300 different ensembles). The X-axis of Fig. 15 is the number of sketch instances, and the Y-axis is the average reduction for the three resource types of 300 ensembles between before and after optimization. We can see that hash unit reduction is up to 40%, SALU reduction is up to 45%, and SRAM reduction is up to 7%. As the ensemble has more sketch instances, we have more opportunities to apply optimizations, and resource reduction benefits increase. SRAM reduction is more limited, but we do observe SRAM reduction for type 2 due to O_{Ctr1} because reusing counter arrays can reduce SRAM.

Fig. 15 also shows that the resource reduction depends on the ensemble type. Ensemble type 2 has sketch instances with the same flowkey, which makes many optimizations easier to apply. Thus, ensemble type 2 has the highest resource reduction. On the other hand, type 4 has random sketch instances, so optimizations are the least likely to be applied, resulting in the smallest resource reduction. However, even for random ensemble type, the reduction of the hash unit is up to 20% and SALU is up to 26% because Sketchovsky offers five multiple building blocks for optimization.

Breakdown on individual optimizations. We zoom into ensembles with 12 sketch instances and show the breakdown of resource reduction in Table 5. HFSREUSE (O_{Key}) shows consistently high resource reduction for all four ensemble types (18% to 25% SALU reduction). Note that O_{Key} can also reduce hash units. This is because of the specific

hardware architecture of Tofino; one hash unit must be allocated for one SALU (now we call this HashUnit-SALU coupling). HASHREUSE (O_{Hash1}) is the next impactful optimization. For type 2, O_{Hash1} reduces hash units by up to 10.4%. SALUREUSE (O_{Ctr1}) reduces both SALU and SRAM and SALUMERGE (O_{Ctr2}) reduces SALUs but increases small SRAM overhead (negative values such as -0.5%). Finally, HASHXOR (O_{Hash2}) has the least impact on Tofino because of HashUnit-SALU coupling. Note that the application of O_{Ctr1} and O_{Ctr2} enables O_{Hash1} automatically. Thus the impact of O_{Ctr1} and O_{Ctr2} is bigger than shown in Table 5.

9 Discussion

Measurement-sketch mapping. We currently assume the ensemble of sketch instances is given as input. An interesting direction for future work is to automatically generate the most efficient ensemble of sketch instances for a given set of measurement tasks. We posit that explicitly considering the characteristic of input workload and the resource-accuracy trade-off in an ensemble setting using Sketchovsky could be an interesting direction for future work [33, 47].

Generalizing to other hardware. While our prototype uses Tofino due to its open-source development API, we posit that our research contributions, such as optimization building blocks, strategy finder, and auto-code composition framework, can be adapted to other programmable switches and platforms as they have similar resource bottlenecks [31, 40].

Generalizing to other sketching algorithms. Today, some sketching algorithms are still infeasible in the data plane due to their complex data structures [39, 40]. We envision Sketchovsky to be useful for implementing other feasible sketching algorithms on programmable switches than the eleven sketching algorithms we demonstrated in §8. In this section, we elaborate on the applicability of Sketchovsky's main components, including optimization building blocks, strategy finder, and auto-code composition framework, to other sketches.

First, the optimization building blocks proposed in Sketchovsky are based on common compute and memory operations in sketching algorithms (e.g., hash computations, arithmetic counter updates, heavy flowkey storage). Since all sketching algorithms perform *hash computations*, O_{Hash1} and O_{Hash2} are generally applicable to current and future sketching algorithms. For *counter updates*, some sketching algorithms may have complicated counter update operations (e.g., threshold-based counter updates in ElasticSketch [45]). Sketchovsky cannot directly support these complicated counter updates and requires a case study to determine whether resource savings are possible. It is also possible that these complicated counter operations are fundamentally excluded from further optimizations. For *heavy flowkey storage*, the sketching algorithms that require storing heavy flowkeys [9, 16, 41] can benefit from O_{Key}. In summary, the individual optimizations introduced in Sketchovsky are broadly applicable to sketching algorithms.

Second, the strategy finder is a general optimizer to maximize resource saving. As long as the specifications of sketching algorithms such as counter array type and counter update type are given as input, the strategy finder will output an optimized strategy. For example, a user may define single-level (SL) or multi-level (ML) as the counter array type and COUNTER or SIGNCOUNTER as the counter update type. When optimizing counting bloom filters [11] as part of an ensemble, the user can define the same counter array and counter update types as the count-min sketch [17], and exclude the heavy flowkey storage part.

Finally, the auto-code composition framework can accommodate new sketching algorithms as long as their sketch templates are properly defined based on the specification of Sketchovsky. A sketch template follows three main steps, i.e., hash computation, counter updates, and flowkey storage. For the auto-code composition framework to work, the user needs to ensure that if several sketches share a common operation (e.g., signed counter update), the code provided for implementing the operation must be the same across the sketch templates. For example, implementing a counting bloom filter should reuse codes from a count-min sketch for the hash computation and counter update operations. In summary, Sketchovsky can be generalized to other sketching algorithms.

10 Conclusions

In this paper, we tackle an often ignored problem of running an ensemble of sketch instances to support a given portfolio of measurement tasks. To the best of our knowledge, Sketchovsky is the first end-to-end system that explores cross-sketch optimizations in practice. We showed that our novel cross-sketch optimization building blocks and efficient strategy finder make previously infeasible ensembles of sketch instances feasible on modern hardware.

Acknowledgment

We would like to thank the anonymous NSDI reviewers and our shepherd Anja Feldmann for their helpful comments. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF awards 1565343, 1700521, 2106946, 2107086, and 2132639. Liu was also supported by the Red Hat Collaboratory at Boston University.

References

[1] Barefoot Tofino Switch. https://barefootnetworks.com/products/brief-tofino/.

[2] CAIDA Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml.

[3] Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association.

[4] SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association.

[5] AGARWAL, A., LIU, Z., AND SESHAN, S. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 719–741.

[6] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., ET AL. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), pp. 503–514.

[7] BASAT, R. B., CHEN, X., EINZIGER, G., AND ROTTENSTREICH, O. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking 28*, 3 (2020), 1172–1185.

[8] BELL, E. T. Exponential polynomials. *Annals of Mathematics* (1934), 258–277.

[9] BEN BASAT, R., EINZIGER, G., FRIEDMAN, R., LUIZELLI, M. C., AND WAISBARD, E. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 127–140.

[10] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (2011), pp. 1–12.

[11] BONOMI, F., MITZENMACHER, M., PANIGRAHY, R., SINGH, S., AND VARGHESE, G. An improved construction for counting bloom filters. In *European Symposium on Algorithms* (2006), Springer, pp. 684–695.

[12] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* (2014).

[13] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[14] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming* (2002), Springer, pp. 693–703.

[15] CHEN, X., LANDAU-FEIBISH, S., BRAVERMAN, M., AND REXFORD, J. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 226–239.

[16] CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Finding hierarchical heavy hitters in data streams. In *Proceedings 2003 VLDB Conference* (2003), Elsevier, pp. 464–475.

[17] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[18] DURAND, M., AND FLAJOLET, P. Loglog counting of large cardinalities. In *European Symposium on Algorithms* (2003), Springer, pp. 605–617.

[19] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 153–166.

[20] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences 31*, 2 (1985), 182–209.

[21] FLAJOLET, P., RIC FUSY, GANDOUET, O., AND ET AL. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA* (2007).

[22] GAO, J., ZHAI, E., LIU, H. H., MIAO, R., ZHOU, Y., TIAN, B., SUN, C., CAI, D., ZHANG, M., AND YU, M. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 435–450.

[23] GAO, X., KIM, T., WONG, M. D., RAGHUNATHAN, D., VARMA, A. K., KANNAN, P. G., SIVARAMAN, A., NARAYANA, S., AND GUPTA, A. Switch code generation using program synthesis. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 44–61.

[24] GARCIA-TEODORO, P., DIAZ-VERDEJO, J., MACIÁ-FERNÁNDEZ, G., AND VÁZQUEZ, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security 28*, 1-2 (2009), 18–28.

[25] GUPTA, A., HARRISON, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 357–371.

[26] KIRSCH, A., AND MITZENMACHER, M. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms* (2006), Springer, pp. 456–467.

[27] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., AND CHEN, Y. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 234–247.

[28] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review 32*, 1 (2004), 177–188.

[29] LALL, A., SEKAR, V., OGIHARA, M., XU, J., AND ZHANG, H. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review 34*, 1 (2006), 145–156.

[30] LI, Y., GAO, J., ZHAI, E., LIU, M., LIU, K., AND LIU, H. H. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 371–385.

[31] LIU, Z., BEN-BASAT, R., EINZIGER, G., KASSNER, Y., BRAVERMAN, V., FRIEDMAN, R., AND SEKAR, V. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 334–350.

[32] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 101–114.

[33] LIU, Z., NAMKUNG, H., AGARWAL, A., MANOUSIS, A., STEENKISTE, P., SESHAN, S., AND SEKAR, V. Sketchy with a chance of adoption: Can sketch-based telemetry be ready for prime time? In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)* (2021), IEEE, pp. 9–16.

[34] LIU, Z., NAMKUNG, H., NIKOLAIDIS, G., LEE, J., KIM, C., JIN, X., BRAVERMAN, V., YU, M., AND SEKAR, V. Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 3829–3846.

[35] MIAO, R., ZENG, H., KIM, C., LEE, J., AND YU, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 15–28.

[36] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (2015), pp. 1–13.

[37] NAMKUNG, H., KIM, D., LIU, Z., SEKAR, V., AND STEENKISTE, P. Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations. In *Proceedings of the Symposium on SDN Research* (2021).

[38] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 85–98.

[39] SCHWELLER, R., GUPTA, A., PARSONS, E., AND CHEN, Y. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (2004), pp. 207–212.

[40] SIVARAMAN, V., NARAYANA, S., ROTTENSTREICH, O., MUTHUKRISHNAN, S., AND REXFORD, J. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research* (2017), pp. 164–176.

[41] SONG, C. H., KANNAN, P. G., LOW, B. K. H., AND CHAN, M. C. Fcm-sketch: generic network measurements with data plane support. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 78–92.

[42] SONI, H., RIFAI, M., KUMAR, P., DOENGES, R., AND FOSTER, N. Composing dataplane programs with μp4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 329–343.

[43] THORUP, M., AND ZHANG, Y. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM Journal on Computing 41*, 2 (2012), 293–331.

[44] WHANG, K.-Y., VANDER-ZANDEN, B. T., AND TAYLOR, H. M. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS) 15*, 2 (1990), 208–229.

[45] YANG, T., JIANG, J., LIU, P., HUANG, Q., GONG, J., ZHOU, Y., MIAO, R., LI, X., AND UHLIG, S. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 561–575.

[46] YU, D., ZHU, Y., ARZANI, B., FONSECA, R., ZHANG, T., DENG, K., AND YUAN, L. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 207–220.

[47] YU, M. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review 49*, 1 (2019), 11–17.

[48] ZHANG, M., LI, G., WANG, S., LIU, C., CHEN, A., HU, H., GU, G., LI, Q., XU, M., AND WU, J. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *Proceedings of NDSS* (2020).

[49] ZHANG, Y., LIU, Z., WANG, R., YANG, T., LI, J., MIAO, R., LIU, P., ZHANG, R., AND JIANG, J. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 207–222.

[50] ZHENG, H., TIAN, C., YANG, T., LIN, H., LIU, C., ZHANG, Z., DOU, W., AND CHEN, G. Flymon: enabling on-the-fly task reconfiguration for network measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 486–502.

[51] ZHENG, P., BENSON, T., AND HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (2018), pp. 98–111.

[52] ZHOU, Y., ZHANG, D., GAO, K., SUN, C., CAO, J., WANG, Y., XU, M., AND WU, J. Newton: intent-driven network traffic monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 295–308.

A Supplement to Background

A.1 Counter Update Type

We introduce five counter update types as in Alg. 2.

1. **BITMAP** is just a bitmap.
2. **COUNTER** receives index and size for the counter update, then increase the counter depending on packet counts or packet bytes.
3. **SIGNCOUNTER** receives one additional input of 1-bit hash result. Depending on this hash value, it will either increase or decrease the counter. The 1-bit hash value is computed by using flowkey.
4. **HLL** type can be used for loglog-variant sketches [18, 21]. It receives index and value as inputs and updates the counter if it is less than the value. Value can be computed by a function $\rho(hash)$ where $hash \in \{0,1\}^{32}, \rho(hash)$ is the position of the leftmost 1-bit (e.g., $\rho(0001\ldots) = 4$) and *hash* is computed using flowkey [21]. This ρ function can be implemented efficiently by using TCAM in the switch data plane [4].
5. **PCSA** receives index and bitmask as inputs. Then it uses the bit-OR operation to update the counter using the

bitmask. Bitmask value can be computed by shift operation $(1 << \rho(hash))$.

Algorithm 2 Five Counter Update Types

```
1: function BITMAP(index)
2:     A[index] = 1
3: function COUNTER(index, size)
4:     A[index] = A[index] + size
5: function SIGNCOUNTER(hash, index, size)
6:     if hash == 0 then
7:         A[index] = A[index] + size
8:     else if hash == 1 then
9:         A[index] = A[index] - size
10: function HLL(index, value)
11:     if A[index] < value then
12:         A[index] = value
13: function PCSA(index, bitmask)
14:     A[index] = A[index] | bitmask
```

B Supplement to Optimizations

B.1 SRAM reduction of SALUREUSE (O_{Ctr1})

SALUREUSE (O_{Ctr1}) reduces not only SALUs but also SRAM. Suppose $\mathbf{S} = \{s_i\}_{i=1}^{n}$ is a set of sketch instances and $\mathbf{C} = \{(r_i, w_i)\}_{i=1}^{n}$ represent that s_i has r_i number of counter arrays with width w_i. $\mathbf{W}$ represents row and width of counter arrays for reuse after applying O_{Ctr1}.

$$\mathbf{W} = \{w_j^*\}_{j=1}^{\max_i(r_i)} \text{ where } w_j^* = \max_i\{w_i | r_i \geq j\} \tag{14}$$

Then, SRAM usage changes from $\sum_{i=1}^{n} r_i w_i$ to $\sum_{j=1}^{\max_i(r_i)} w_j^*$. O_{Ctr1} will always maintain or reduce SRAM usage because $\sum_{i=1}^{n} r_i w_i - \sum_{j=1}^{\max_i(r_i)} w_j^* \geq 0$. Suppose $comp(x,y) \in \{0,1\}$ where $x, y \in \mathbb{N}$. If $x \leq y \rightarrow comp(x,y) = 1$, otherwise $\rightarrow comp(x,y) = 0$.

$$\sum_{i=1}^{n} r_i w_i - \sum_{j=1}^{\max_i(r_i)} w_j^* = \sum_{j=1}^{\max_i(r_i)} \left((\sum_{i=1}^{n} w_i \cdot comp(r_i, j)) - w_j^* \right)$$

$(\sum_{i=1}^{n} w_i \cdot comp(r_i, j)) - w_j^* \geq 0$ for $1 \leq j \leq max_i(r_i)$ due to (14)

Thus, $\sum_{i=1}^{n} r_i w_i - \sum_{j=1}^{\max_i(r_i)} w_j^* \geq 0$

C Supplement to Auto-code Composition

C.1 SketchLib and Lib for Optimization

SketchLib. We extended API calls from SketchLib as in Table 6 for the easier code-rewrite process.

- `TCAM_LPM (hash_result)` uses TCAM for the longest prefix match to compute the leftmost position of 1-bit in the hash result, which is used in many sketching algorithms. This API call is the same as `tcam_optimization()` in SketchLib.
- `CounterUpdate (seed, flowkey, width, CA_type, CU_type, ...)` does one counter update for configured flowkey, counter array type (`CA_type`) of whether single-level (SL) or multi-level (ML), counter update type (`CU_type`), width of counter array (`width`). `seed` is used for the hash unit to generate column index for the counter update. Depending on the different `CU_type`, it takes more parameters (e.g., packet length for COUNTER type or value out of `TCAM_LPM` for HLL/PCSA type). We extended `consolidate_update()` in SketchLib to build this API call.
- `AboveThreshold (LIST(estimate), threshold)` gets the threshold and a list of flow size estimates (these are return values after each counter update). This API call returns whether the overall flow size estimate is above the threshold or not[6]. This logic was part of `heavy_flowkey_storage()` in SketchLib and we separate the API call for the code rewrite process.
- `HFS (flowkey)` stores heavy flowkey. This API extends `heavy_flowkey_threshold()` in SketchLib by supporting any definition of flowkey.

```
01: /* 1. hash computation step - no code */
02: /* 2. counter update step */
03: s#_est1 = CounterUpdate(seed1, FLOWKEY, WIDTH,
04:                         SL, Counter, FLOWSIZE);
05: s#_est2 = CounterUpdate(seed2, FLOWKEY, WIDTH,
06:                         SL, Counter, FLOWSIZE);
07: s#_est3 = CounterUpdate(seed3, FLOWKEY, WIDTH,
08:                         SL, Counter, FLOWSIZE);
09: s#_th = AboveThreshold(s#_est1, s#_est2, s#_est3,
10:                                          THRESHOLD);
11: /* 3. counter update step */
12: if (s#_th) { HFS(FLOWKEY); }
```

Figure 16: Code template example for count-min sketch.

You can see how these API calls are used in Fig. 16, which is a code template example for count-min sketch. Network operators can put {srcIP, dstIP} to FLOWKEY, *hdr.ipv4.total_len* to FLOWSIZE, and *1024* to WIDTH. For different numbers of counter arrays (e.g., 3 counter arrays), network operators should write multiple lines of code for counter updates (e.g., lines 3-8 in Fig. 16).

Lib for Opt. Lib for Opt is used to implement SALUMERGE (O_{Ctr2}) as in Table 6.

- `CounterUpdate_2 (seed, flowkey, width, CA_type, CU_type1, CU_type2, ...)` This API looks similar to `CounterUpdate()` but the difference is that this API does two counter updates by using one SALU. Thus, parameters include two counter update types

[6] For count-min sketch, overall flow size estimate is min of List (estimate). For count-sketch, overall flow size estimate is median of List (estimate).

Two Libs	API Name	API Parameters	Explanation
SketchLib	`TCAM_LPM()`	`hash_result`	Same as `tcam_optimization()` in SketchLib
	`CounterUpdate()`	`seed, flowkey, width, CA_type, CU_type, ...`	Extends `consolidate_update()` in SketchLib
	`AboveThreshold()`	`LIST(estimate), threshold`	Extends `heavy_flowkey_storage()` in SketchLib
	`HFS()`	`flowkey`	Extends `heavy_flowkey_storage()` in SketchLib
Lib for Opt	`CounterUpdate_2()`	`seed, flowkey, width, CA_type, CU_type1, CU_type2, ...`	New API for SALUMERGE (O_{Ctr2})

Table 6: API calls in extended SketchLib and Lib for optimization.

```
01: // code for s1
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s1_est1 = CounterUpdate(seed1, srcIP, 2K, SL,
05:                         Counter, pktlen);
06: s1_est2 = CounterUpdate(seed2, srcIP, 2K, SL,
07:                         Counter, pktlen);
08: s1_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                         Counter, pktlen);
10: s1_th = AboveThreshold(s1_est1, s1_est2, s1_est3,
11:                                                100);
12: ... /* 3. heavy flowkey storage step */
13:
14: // code for s2
15: ... /* 1. hash computation step */
16: /* 2. counter update step */
17: s2_est1 = CounterUpdate(seed4, srcIP, 4K, SL,
18:                         Counter, pktlen);
19: s2_est2 = CounterUpdate(seed5, srcIP, 4K, SL,
20:                         Counter, pktlen);
21: s2_th = AboveThreshold(s2_est1, s2_est2, 100);
22: ... /* 3. heavy flowkey storage step */
23:
24: // code for s3
25: ... /* 1. hash computation step */
26: /* 2. counter update step */
27: CounterUpdate(seed6, srcIP, 4K, SL, Counter,
28:                                       pktlen);
29: ... /* 3. heavy flowkey storage step */
```

Figure 17: [Before] SALUREUSE (O_{Ctr1}).

`CU_type1` and `CU_type2`. There are one flowkey, one width, and one counter array type because they should be the same due to applicable conditions of O_{Ctr2}.

C.2 Before and After Code Snippets for O_{Ctr1}, O_{Ctr2}, and O_{Key}

Code rewrite for counter update. Code rewriter uses $\{X^*_{Ctr1}, X^*_{Ctr2}\}$ to apply SALUREUSE (O_{Ctr1}) and SALUMERGE (O_{Ctr2}) to counter update step. Although O_{Ctr1} and O_{Ctr2} can be applied simultaneously, we explain code rewrite logic separately for better readability. Code rewrite for O_{Ctr1} to $\mathcal{S}$ requires code changes with lines using `CounterUpdate()` in the extended SketchLib. Code rewrite for O_{Ctr2} uses a new API call, `CounterUpdate_2()`.

We first look at how to apply O_{Ctr1} using X^*_{Ctr1} by looking at the before (Fig. 17) and after (Fig. 18) code snippets. Three sketch instances $\{s_1, s_2, s_3\}$ in Fig. 17 are count-min sketch, K-ary sketch, and entropy sketch respectively and they have different resource parameters $\mathbf{C} = \{(r_i, w_i)\}_{i=1}^{3} = \{(3, 2K), (2, 4K), (1, 8K)\}$. $\{s_1, s_2\}$ tracks heavy flowkey and they check whether flow size estimate is above threshold at line 10 and 21 in Fig. 17. X^*_{Ctr1} specifies that code rewriter should apply O_{Ctr1} to $\{s_1, s_2, s_3\}$, meaning that they satisfy applicable conditions for O_{Ctr1}. Then, the code rewriter computes row and width of counter arrays for reuse $\mathbf{W}$ as discussed as in (3), §4.2. As a result, $\mathbf{W} = \{8K, 4K, 2K\}$ is computed in this example and the code rewriter applies this as in lines 4-9 in code snippet Fig. 18.

```
01: // optimized code for s1, s2, s3
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s_est1 = CounterUpdate(seed1, srcIP, 8K, SL,
05:                        Counter, pktlen);
06: s_est2 = CounterUpdate(seed2, srcIP, 4K, SL,
07:                        Counter, pktlen);
08: s_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                        Counter, pktlen);
10: s1_th = AboveThreshold(s_est1, s_est2, s_est3,
11:                                              100);
12: s2_th = AboveThreshold(s_est1, s_est2, s_est3,
13:                                              200);
14: ... /* 3. counter update step */
```

Figure 18: [After] SALUREUSE (O_{Ctr1}) to $\{s_1, s_2, s_3\}$.

Next, we look at how the code rewriter applies O_{Ctr2} by using X^*_{Ctr2}. Fig. 19 is the before code snippet and Fig. 20 is the after code snippet. $\{s_1, s_2, s_3\}$ in Fig. 19 are count-min sketch, entropy sketch, and PCSA sketch respectively and $\mathbf{C} = \{(3, 2K), (2, 4K), (1, 8K)\}$. We cannot apply O_{Ctr1} to $\{s_1, s_2, s_3\}$ for this example because flowsize definitions are different between s_1 and s_2 (s_1 tracks packet bytes if we look at lines 5, 7, 9 in Fig. 19 whereas s_2 tracks packet counts at lines 17-18 in Fig. 19). Counter update types are also different between $\{s_1, s_2\}$ and $\{s_3\}$. $\{s_1, s_2\}$ uses COUNTER type whereas $\{s_3\}$ uses PCSA type.

Instead of O_{Ctr1}, we can apply O_{Ctr2} and X^*_{Ctr2} specifies that the code rewriter can apply O_{Ctr2} to $\{s_1, s_2, s_3\}$. Using the information in X^*_{Ctr2}, the code rewriter knows that the first two counter arrays of s_1 can share SALUs with s_2, and the last counter array of s_1 can share a SALU with s_3. We use the new API call `CounterUpdate_2()` to apply this optimization at lines 4-9 in Fig. 20. For the first two counter arrays (lines 4-7), both counter update types are COUNTER type. Thus, the API call takes two additional parameters of flowsize definitions of packet bytes and packet counts. For the third counter array (lines 8-9), counter update types are COUNTER and PCSA.

```
01: // code for s1
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s1_est1 = CounterUpdate(seed1, srcIP, 2K, SL,
05:                         Counter, pktlen);
06: s1_est2 = CounterUpdate(seed2, srcIP, 2K, SL,
07:                         Counter, pktlen);
08: s1_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                         Counter, pktlen);
10: s1_th = AboveThreshold(s1_est1, s1_est2, s1_est3,
11:                                              100);
12: ... /* 3. heavy flowkey storage step */
13:
14: // code for s2
15: ... /* 1. hash computation step */
16: /* 2. counter update step */
17: CounterUpdate(seed4, srcIP, 4K, SL, Counter, 1);
18: CounterUpdate(seed5, srcIP, 4K, SL, Counter, 1);
19: ... /* 3. heavy flowkey storage step */
20:
21: // code for s3
22: ... /* 1. hash computation step */
23: /* 2. counter update step */
24: CounterUpdate(seed6, srcIP, 8K, SL, PCSA,
25:                                    s3_value);
26: ... /* 3. heavy flowkey storage step */
```

Figure 19: [Before] SALUMERGE (O_{Ctr2}).

```
01: // optimized code for s1, s2, s3
02: ... /* 1. hash computation step */
03: /* 2. counter update step */
04: s_est1 = CounterUpdate_2(seed1, srcIP, 8K, SL,
05:                     Counter, Counter, pktlen, 1);
06: s_est2 = CounterUpdate_2(seed2, srcIP, 4K, SL,
07:                     Counter, Counter, pktlen, 1);
08: s_est3 = CounterUpdate(seed3, srcIP, 2K, SL,
09:                 Counter, PCSA, pktlen, s3_value);
10: s1_th = AboveThreshold(s_est1, s_est2, s_est3,
11:                                          100);
12: ... /* 3. counter update step */
```

Figure 20: [After] SALUMERGE (O_{Ctr2}) to $\{s_1, s_2, s_3\}$.

Thus, two additional parameters are flowsize definition of packet bytes and an output value of `TCAM_LPM` written as `s3_value` at line 9 in Fig. 20.

Code rewrite for heavy flowkey storage. Code rewriter uses X^*_{Key} to apply HFSREUSE (O_{Key}) to the heavy flowkey storage step. Fig. 21 is the before code snippet and Fig. 22 is the after code snippet. We have four sketch instances $\{s_1, s_2, s_3, s_4\}$ with different flowkeys F = {{srcIP}, {srcIP, dstIP}, {srcIP, srcPort}, {srcIP, dstIP, srcPort, dstPort}} and all sketch instances track heavy flowkey. O_{Key} uses union key $UK = \cup_i f_i$ for the heavy flowkey storage for reuse. In this example, UK ={srcIP, dstIP, srcPort, dstPort} is written at line 14 in Fig. 22. Recall that we have further optimization using conditional union-key $UK_C = \cup_j f_j$where (flow size estimate)$_j$ > $threshold_j$ and set 0 to $(UK - UK_C)$. This optimization is written in the code at lines 6-11 in Fig. 22. For

```
01: // code for s1
02: ... /* 1. hash computation step */
03: ... /* 2. counter update step */
04: /* 3. heavy flowkey storage step */
05: if (s1_th) { HFS(srcIP); }
06:
07: // code for s2
08: ...
09: /* 3. heavy flowkey storage step */
10: if (s2_th) { HFS(srcIP, dstIP); }
11:
12: // code for s3
13: ...
14: /* 3. heavy flowkey storage step */
15: if (s3_th) { HFS(srcIP, srcPort); }
16:
17: // code for s4
18: ...
19: /* 3. heavy flowkey storage step */
20: if (s4_th) { HFS(srcIP, dstIP, srcPort, dstPort); }
```

Figure 21: [Before] HFSREUSE (O_{Key}).

```
01: // code for s1, s2, s3, s4
02: ... /* 1. hash computation step */
03: ... /* 2. counter update step */
04: /* 3. heavy flowkey storage step */
05: hf_srcIP = hf_dstIP = hf_srcPort = hf_dstPort = 0
06: if (s1_th || s2_th || s3_th || s4_th) {
07:   hf_srcIP = srcIP;
08: }
09: if (s2_th || s4_th) { hf_dstIP = dstIP; }
10: if (s3_th) { hf_srcPort = srcPort; }
11: if (s4_th) {hf_dstPort = dstPort; }
12:
13: if (s1_th || s2_th || s3_th || s4_th) {
14:   HFS(hf_srcIP, hf_dstIP, hf_srcPort, hf_dstPort);
15: }
```

Figure 22: [After] HFSREUSE (O_{Key}) to $\{s_1, s_2, s_3, s_4\}$.

each packet header field (e.g., dstIP), it detects which sketch instances have this header field (e.g., s_2 and s_4 because f_2 and f_4 have dstIP). Then if any of those sketch instances is above the threshold (at line 9), those header fields are included in UK_C. If not, this header field is set to zero (at line 5). As a result, we can reduce 4 heavy flowkey storages to 1 heavy flowkey storage.

D Supplement to Evaluation

D.1 Eleven Sketch Algorithms for Evaluation

We use eleven sketching algorithms for our evaluation as in Table 7. They have different sketch features. Counter array type can be single-level (SL) or multi-level (ML). We also show a pool of candidate configurable parameters per each sketching algorithm in Table 7. For entropy sketch, counter update type SIGNCOUNTER guarantees theoretically better accuracy due to F2 estimation. However, we found that the COUNTER type produces better accuracy in practice. Thus, we use this COUNTER type for entropy sketch in our evaluation.

Sketch Algorithms		Sketch Features			Configurable Parameters Candidates					
Statistic	Name	Counter Array	Counter Update	Heavy Flowkey	Flowkey	Flowsize	Epoch	Row	Width	Level
Membership	BF [11]	SL	BITMAP	N	{(srcIP), (dstIP), (srcIP, dstIP), (srcIP, srcPort), (dstIP, dstPort), (4-tuple), (5-tuple)}	{counts}	{10s, 20s, 30s, 40s}	{1}	{128K, 256K, 512K}	-
Cardinality	LC [44]	SL	BITMAP	N					{16K, 32K}	{8, 16}
	MRB [19]	ML	BITMAP	N						
	PCSA [20]	SL	PCSA	N					{4K, 8K, 16K}	-
	HLL [21]	SL	HLL	N						
HH/HC	CS [14]	SL	SIGNCOUNTER	Y		{counts, bytes}		{1, 2, 3, 4, 5}		
	CM [17]	SL	COUNTER	Y						
	KARY [27]	SL	COUNTER	Y						
Entropy	ENT [29]	SL	COUNTER	N		{counts}		{3,4,5}	{2K}	{16}
General	UM [32]	ML	SIGNCOUNTER	Y						
FSD	MRAC [28]	ML	COUNTER	N				{1}		{8, 16}

Table 7: Eleven sketch algorithms with sketch features and possible configurable parameters. (4-tuple) = (srcIP, dstIP, srcPort, dstPort). (5-tuple) = (srcIP, dstIP, srcPort, dstPort, proto).

D.2 Four Ensembles for Accuracy Evaluation

Table 8 - Table 11 shows four picked ensembles for four ensemble types. All five optimizations are found in four picked ensembles.

Ensemble Type 1.

- HASHREUSE (O_{Hash1}): none
- HASHXOR (O_{Hash2}): none
- SALUREUSE (O_{Ctr1}): none
- SALUMERGE (O_{Ctr2}): none
- HFSREUSE (O_{Key}): $\{s_1, s_2, s_3, s_4, s_5, s_6\}$

Ensemble Type 2.

- HASHREUSE (O_{Hash1}): $\{s_3, s_4, s_6, s_{10}\}$
- HASHXOR (O_{Hash2}): none
- SALUREUSE (O_{Ctr1}): $\{s_8, s_9\}$
- SALUMERGE (O_{Ctr2}): $\{\{s_1\}, \{s_2\}\}$, $\{s_7, \{s_8, s_9\}\}$
- HFSREUSE (O_{Key}): $\{s_2, s_8, s_9\}$

Ensemble Type 3.

- HASHREUSE (O_{Hash1}): $\{s_3, s_4\}$
- HASHXOR (O_{Hash2}): none
- SALUREUSE (O_{Ctr1}): $\{s_8, s_9\}$
- SALUMERGE (O_{Ctr2}): $\{\{s_3\}, \{s_4\}\}$, $\{\{s_6\}, \{s_7\}\}$
- HFSREUSE (O_{Key}): $\{s_4, s_5\}$

Ensemble Type 4.

- HASHREUSE (O_{Hash1}): none
- HASHXOR (O_{Hash2}): $\{\{s_1\}, \{s_2\}, \{s_3\}\}, \{\{s_4\}, \{s_5\}, \{s_9\}\}$
- SALUREUSE (O_{Ctr1}): none
- SALUMERGE (O_{Ctr2}): $\{\{s_7\}, \{s_8\}\}$
- HFSREUSE (O_{Key}): none

SI	Base SA (*)	Configurable Parameters			
		Flowkey	Flowsize	Epoch	Resource
s_1	CM	(srcIP)	counts	40s	(1, 16K)
s_2	CM	(srcIP)	bytes	10s	(5, 4K)
s_3	CM	(srcIP, dstIP)	bytes	30s	(2, 16K)
s_4	CM	(srcIP, srcPort)	bytes	30s	(5, 8K)
s_5	CM	(dstIP, dstPort)	bytes	20s	(2, 4K)
s_6	CM	(5-tuple)	counts	40s	(5, 8K)

Table 8: Ensemble Type 1. Same Sketch Algorithm

SI	Base SA	Configurable Parameters			
		Flowkey(*)	Flowsize	Epoch	Resource
s_1	ENT	(dstIP, dstPort)	counts	10s	(3, 16K)
s_2	CS	(dstIP, dstPort)	counts	10s	(3, 16K)
s_3	MRB	(dstIP, dstPort)	-	20s	(1, 16K, 8)
s_4	MRAC	(dstIP, dstPort)	counts	20s	(1, 2K, 8)
s_5	BF	(dstIP, dstPort)	-	30s	(3, 128K)
s_6	MRB	(dstIP, dstPort)	-	30s	(1, 16K, 16)
s_7	ENT	(dstIP, dstPort)	counts	30s	(4, 4K)
s_8	CM	(dstIP, dstPort)	bytes	30s	(3, 4K)
s_9	KARY	(dstIP, dstPort)	bytes	30s	(1, 4K)
s_{10}	MRAC	(dstIP, dstPort)	counts	40s	(1, 2K, 8)

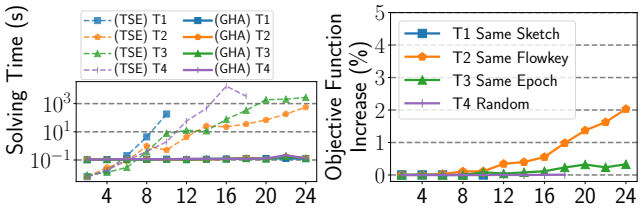
Table 9: Ensemble Type 2. Same Flowkey

SI	Base SA	Configurable Parameters			
		Flowkey	Flowsize	Epoch(*)	Resource
s_1	HLL	(srcIP)	-	30s	(1, 16K)
s_2	HLL	(dstIP)	-	30s	(1, 4K)
s_3	MRAC	(srcIP, dstIP)	counts	30s	(1, 2K, 8)
s_4	UM	(srcIP, dstIP)	counts	30s	(3, 2K, 16)
s_5	UM	(srcIP, srcPort)	counts	30s	(4, 2K, 16)
s_6	PCSA	(dstIP, dstPort)	-	30s	(1, 8K)
s_7	ENT	(dstIP, dstPort)	counts	30s	(2, 16K)
s_8	BF	(4-tuple)	-	30s	(3, 128K)
s_9	LC	(4-tuple)	-	30s	(1, 128K)
s_{10}	ENT	(5-tuple)	counts	30s	(5, 4K)

Table 10: Ensemble Type 3. Same Epoch

SI	Base SA	Configurable Parameters			
		Flowkey	Flowsize	Epoch	Resource
s_1	MRAC	(srcIP)	counts	20s	(1, 2K, 16)
s_2	MRB	(dstIP)	-	30s	(1, 16K, 8)
s_3	MRB	(srcIP, dstIP)	-	20s	(1, 32K, 8)
s_4	HLL	(srcIP, srcPort)	-	10s	(1, 4K)
s_5	PCSA	(dstIP, dstPort)	-	20s	(1, 16K)
s_6	ENT	(dstIP, dstPort)	counts	30s	(3, 8K)
s_7	ENT	(4-tuple)	counts	30s	(5, 4K)
s_8	CS	(4-tuple)	counts	30s	(3, 8K)
s_9	PCSA	(4-tuple)	-	40s	(1, 16K)
s_{10}	HLL	(5-tuple)	-	30s	(1, 8K)

Table 11: Ensemble Type 4. Random



(a) Solving Time **(b)** Performance on Reduction

Figure 23: Two-step enumeration (TSE) vs greedy heuristic algorithm (GHA).

D.3 Experiment for Greedy Heuristic Algorithm

In the strategy finder section (§5), we propose the greedy heuristic algorithm to tackle the problem of large search space. Here we show that the performance loss of the greedy heuristic algorithm is small while solving time is three orders of magnitude faster.

Metric. We introduce two metrics for this experiment.

- Solving Time: time to find the solution.
- Objective Function Increase: $\frac{HwResource(X_G)}{HwResource(X_T)}$ where X_T is a found solution using the two-step enumeration and X_G is from the greedy heuristic algorithm.

We can see in Fig. 23a that the greedy heuristic algorithm is three orders of magnitude faster than two-step enumeration. However, the objective function increase is less than 2% (Fig. 23b). For solving time, we measure time for 300 ensembles per data point in Fig. 23a and show the worst solving time. Data points that take more than 24 hours are not shown.

RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs

Jiaxin Lin *UT Austin*
Adney Cardoza *UT Austin*
Tarannum Khan *UT Austin*
Yeonju Ro *UT Austin*
Brent E. Stephens *University of Utah*
Hassan Wassel *Google*
Aditya Akella *UT Austin*

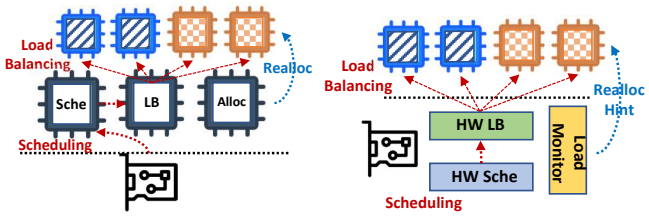
Abstract

Careful orchestration of requests at a datacenter server is crucial to meet tight tail latency requirements and ensure high throughput and optimal CPU utilization. Orchestration is multi-pronged and involves load balancing and scheduling requests belonging to different services across CPU resources, and adapting CPU allocation to request bursts. Centralized intra-server orchestration offers ideal load balancing performance, scheduling precision, and burst-tolerant CPU re-allocation. However, existing software-only approaches fail to achieve ideal orchestration because they have limited scalability and waste CPU resources. We argue for a new approach that offloads intra-server orchestration entirely to the NIC. We present RingLeader, a new programmable NIC with novel hardware units for software-informed request load balancing and programmable scheduling and a new light-weight OS-NIC interface that enables close NIC-CPU coordination and supports NIC-assisted CPU scheduling. Detailed experiments with a 100 Gbps FPGA-based prototype show that we obtain better scalability, efficiency, latency, and throughput than state-of- the-art software-only orchestrators including Shinjuku and Caladan.

1 Introduction

Modern cloud services generate thousands of RPCs in response to a single external request [35]. The services often need to provide microsecond-scale tail latencies for these RPCs to meet service level objectives (SLOs) [4]. What makes this challenging is that each server in a distributed system running multiple services receives many RPC requests of varying importance, and *intra-server orchestration*, which is necessary to provide low tail latencies and high CPU efficiency, itself incurs substantial latency and wastes CPU cycles.

Intra-server orchestration entails three aspects (Figure 1a): request scheduling, load balancing, and core assignment [6, 13, 14, 19, 24, 31, 33]. These tasks play an indispensable role in maintaining microsecond-scale tail latency, achieving high



(a) **Intra-server orchestration** (b) **Ringleader**

Figure 1: Intra-server orchestration: today vs. Ringleader.

CPU efficiency and high throughput, and enforcing appropriate request prioritization. Request scheduling and load balancing determine, within and across services, in what **order** requests are processed and by **which** worker core [6, 14, 19, 33]. Load balancing reduces tail latencies by reducing worker queue lengths and improves CPU efficiency as fewer cores in the system are left idle when they could instead be processing requests. When requests or services have different SLOs or priorities, scheduling can eliminate head-of-line (HoL) blocking and guarantee tail latencies for critical workloads. Core re-allocation decides how cores process requests belonging to different services [13, 24, 31]. Fast re-allocation maintains low tail latency and improves CPU efficiency and throughput, as it can repurpose cores that are not needed by a latency-sensitive service toward batch services during periods of low load.

Coordinating orchestration tasks is a vision shared by other recent systems that have either on-loaded orchestration onto dedicated CPU cores [6, 14, 16, 19, 31, 33] or offloaded some aspects to SmartNICs [15]. Unfortunately, both sets of approaches have key limitations (Sections 2 and 8). On-loading has high latencies, poor scalability, and wastes CPU cycles. Using a dedicated centralized orchestrator core does not scale with increasing network line rates and worker core counts. Offloading orchestration to SmartNICs using on-NIC CPU cores has similar issues: the wimpy on-NIC cores have high latency overheads and scalability limitations.

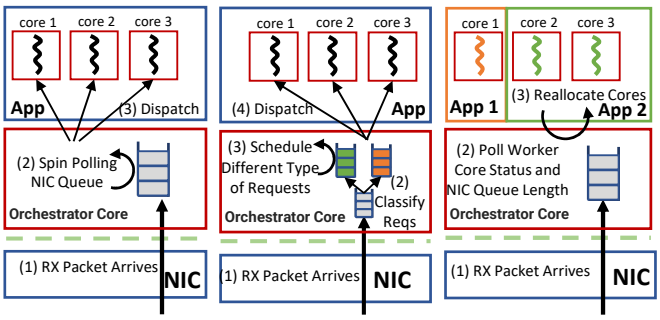
We argue that effective orchestration requires a fundamentally different division of labor than onloading or SmartNIC-based approaches: Given recent advances in programmable

network hardware, we start with an approach that offloads as many of the different aspects of orchestration as possible onto NIC hardware while systematically onloading onto host cores minimal functionality for precise scheduling and high performance. As NICs already process all incoming packets, offloading orchestration tasks can reduce request processing latency and save CPU cycles. We realize this division-of-labor in RingLeader, a system for offloading and executing intra-server orchestration on 100+Gbps NICs (Figure 1b).

In RingLeader, software running on CPUs uses a new OS-NIC interface to provide the NIC with per-core updates on request completions and relative priorities across arriving requests. Custom-built load balancing and scheduling units on the NIC interface with each other and leverage software-provided information to schedule precisely and enqueue requests within/across services at cores. By tracking NIC-local queues of requests waiting to be scheduled, the RingLeader NIC detects load changes and provides fine-grained reallocation hints to host cores via the same OS-NIC interface.

Several challenges arise in making this division-of-labor effective (Sec. 2.3): (1) carefully distributing packet buffering across the NIC and CPU cores to avoid core idling while tightly controlling request dispatch from the NIC to CPU cores; (2) coordinating request dispatching among per-core buffers and the on-NIC load balancing and scheduling engines to meet various load targets and scheduling policies; (3) developing hardware support to combine load balancing and scheduling decisions at line-rate; and (4) developing an efficient OS-NIC interface to enable low overhead coordination between the NIC and host cores. We make several innovations (Secs. 4 and 5) to overcome these challenges:

1. We leverage *shallow per-core request priority queues* alongside limited on-NIC buffering to overcome the challenges caused by PCIe latency and ensure requests dispatched by the NIC are processed quickly and with suitable prioritization.
2. We develop a novel load balancing algorithm, Join-Bounded-Shortest-Ranked-Queue (JBSRQ), which accounts for multi-service isolation/priorities *and* ensures good load balance across the per-core buffers. We build a new first-eligible-out (FEO) line-rate request scheduler that coordinates with the request load balancer.
3. We develop new NIC hardware that uses a reduction tree to calculate which core should process the current highest priority request at the line rate.
4. We introduce an OS-NIC interface with low CPU overheads and avoid generating extra PCIe messages. This provides an API for services to benefit from on-NIC orchestration, and achieves $\sim$ 50M messages-per-second for OS-NIC communication.
5. We develop simple NIC-assisted algorithms that support burst-sensitive core re-allocation across high/low priority requests by leveraging re-allocation hints provided by a new on-NIC load monitoring module.



(a) **Load Balancing** (b) **Scheduling** (c) **Core Allocation**

Figure 2: Illustrations of existing intra-server orchestration.

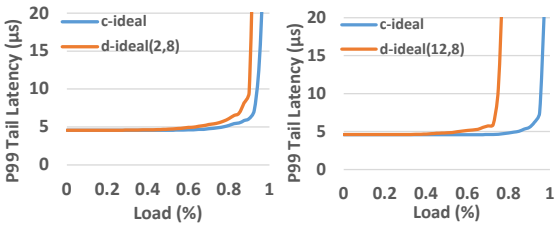
We present a full evaluation of RingLeader's feasibility and effectiveness. From experiments performed on a 100 Gbps FPGA-based prototype, we find RingLeader is high-performance, scalable and CPU-efficient, and RingLeader provides better latency and throughput than existing state-of-the-art intra-server orchestrators, including Shinjuku [19] and Caladan [13]. For example, in an experiment with 30 worker hyperthreads, RingLeader was able to service 3$\times$ as many requests within a P99 SLO of 45μs as Shinjuku and RSS. We compare RingLeader's core allocation with Caladan running both a latency-sensitive service and a batch service, and RingLeader achieves up to 50% less latency for the latency-sensitive service and 1.3$\times$ throughput for the batch service.

2 Background and Motivation

Online cloud services such as search, distributed model serving pipelines, and key-value caches are deployed today across thousands of physical machines. User requests to these services are composed of sequences of RPCs. Each RPC is processed using a two-layer scheduling framework: first, RPCs are assigned to servers, and then RPCs are dispatched to a service instance running on one of the server cores [43]. The latter, i.e., intra-server orchestration, which consists of load balancing requests and scheduling (ordering) them across service instances, and reallocating cores across services based on demand, play a crucial role in the ultimate performance experienced by requests.

2.1 Intra-server Orchestration Today

State-of-the-art (SOTA) intra-server orchestration relies on a centralized software-based approach running in user-space [6, 10, 13, 19, 31, 33]. This approach addresses the unpredictable/high tail latency issues of conventional in-kernel approaches [4, 14, 18]. It also addresses both the load imbalance, poor tail latencies, and poor request scheduling issues of decentralized randomized RSS (receive-side steering) approaches such as IX and ZygOS [6, 33] and the imbalance and



(a) **16 Worker Cores** (b) **96 Worker Cores**

Figure 3: Simulation results comparing centralized and decentralized scheduling. **C-ideal** uses the ideal centralized scheduling policy. **D-ideal(X,Y)** uses X decentralized orchestrator cores to schedule Y worker cores per orchestrator core.

imprecision that results from the asynchrony of reactively programming aRFS (accelerated Receive Flow Steering) rules.

Figure 2 shows how existing orchestration mechanisms work: **(1) Request Load Balancing** (Figure 2a): For each service, one orchestrator core is dedicated to: 1) polling a centralized receive queue and 2) dispatching packets to worker cores according to their load. Packets are delivered from the NIC to the orchestrator core in a centralized, First Come First Serve manner. **(2) Request Scheduling** (Figure 2b): The orchestrator core identifies different request types and schedules competing requests, *e.g.*, by suitably prioritizing them. Request scheduling reduces HoL blocking and ensures RPCs with higher priority receive service first. **(3) CPU Allocation** (Figure 2c): When there are multiple services running on the host, the orchestrator core detects when services would benefit from more cores, and reallocates cores to ensure low latency and high CPU utilization under fast-changing load.

Multiple dedicated busy-polling orchestration cores may be needed to support demanding service workloads running across many cores, or when multiple services run on a server. **Limitations:** A core that is used as an orchestrator incurs overhead and is unable to perform service-specific processing; this is problematic because the CPU is the key bottleneck in today's network-intensive workloads. Further, a single orchestrator core's maximum throughput determines scalability w.r.t request processing rates. Shinjuku's "dispatcher" that performs request scheduling and load balancing only achieves 5M RPS (Requests-Per-Second) with a single core [15, 19]. With μs-scale requests, one orchestrator core can saturate ~ 5 worker cores. However, servers today may be equipped with 100s of cores and serve 100+Gbps demand.

The overheads of performing reallocations over a large pool of worker cores are not negligible either, limiting reallocation speed and precision. For example, with 16 worker cores (hyperthreads), Shenango's core allocator can only support packet rates of up to 6.5 Mpps, and this can only saturate a 10 Gbps NIC with 128B packets [31].

To achieve higher throughput, multiple orchestrator cores could be used. Each orchestrator core handles a set of worker cores, and the server relies on NIC RSS (Receive-Side Scaling) to spread requests across orchestrator cores. However, because orchestrator cores operate independently, it is not possible to simultaneously enforce request scheduling policies and ensure even load and high core utilization.[1]

We built a discrete-event simulator to quantify the impact of using multiple orchestrator cores on a given service. For simplicity, we focus here on comparing the load balancing performance between an ideal centralized approach (c-ideal), and an ideal decentralized approach (that ignores the costs of using many orchestrator cores). We generate requests with service times following an exponential distribution with a mean of 1μs (Exp(1)).

Figure 3 shows the results for 16-core and 96-core systems. The saturation point of the d-ideal is much earlier than the c-ideal, especially when the worker core count is high. This is because the processing time for each request is unpredictable, and using RSS to partition requests between orchestrator cores leads to severe load imbalance. This imbalance causes CPU underutilization, unnecessary queuing, and increased latency.

Recent work improves on RSS by enabling work-stealing between cores to avoid load imbalance [24]. However, work-stealing incurs CPU overheads; it is hard to enforce request weights or priorities under work-stealing; and, as recent work has shown, centralized orchestration still significantly outperforms work-stealing (Fig. 3 in [24]).

2.2 A Case for NIC-Offloaded Orchestration

Using the NIC to perform orchestration has the potential to solve the key limitations associated with software-based approaches. Because all incoming requests necessarily pass through the NIC, the NIC could be an ideal location to perform *request scheduling* and *load balancing*; the NIC can buffer incoming requests and, in theory, make centralized scheduling and load balancing decisions without added latency. In contrast with software-only approaches that must sacrifice performance and efficiency to operate at scale, high-performance on-NIC accelerators can be designed to operate at the hyperscale required by today's line rates and core counts. Additionally, offloading orchestration tasks onto NIC hardware can further improve host CPU efficiency by freeing up host cores, removing inter-thread communication overheads, and improving the accuracy of scheduling and load balancing decisions. The NIC is also a good vantage point for fine-grained network load profiling and queuing delay monitoring, so the NIC can assist with CPU scheduling by providing hints regarding incipient load arriving over the network.

We further argue that once a decision has been made to offload load balancing to the NIC, it is necessary to *also* offload scheduling and load monitoring. To achieve the c-ideal line in Figure 3, it is necessary to perform centralized buffering and

[1] aRFS allows the orchestrator cores to program flow steering rules on the NIC [8], this cannot prevent load imbalance at short time scales because rules must be installed reactively and is imprecise because rules are installed asynchronously.

load balancing. However, once requests are buffered on the NIC, it is not possible to prevent a high priority request from being blocked inside the on-NIC buffer, and centralized on-NIC buffering hides information about buffered requests from a CPU-based scheduler, precluding informed scheduling.

2.3 On-NIC Orchestration Challenges

Achieveing on-NIC orchestration is challenging:

C1: To Buffer at cores or not: On-NIC orchestration requires tight coordination between the NIC and the host. A NIC-only approach where: (1) all incoming packets are buffered on the NIC, (2) the NIC computes which core and in what order to process incoming requests, and (3) cores pull "ready" requests from the NIC to process can, in theory, yield good load balance and adhere to scheduling policies perfectly, but can experience poor throughput and fallow cores due to PCIe latency. To improve throughput and utilization, we need to unload some amount of buffering onto the cores by allowing the NIC to send new packet descriptors to a core that is not yet finished processing its current request. But it is unclear how deep these per-core buffers should be and what queueing discipline they should implement. Deep FIFO buffers can improve utilization but impose HoL blocking with high-priority requests stuck behind low priority ones at a core.

C2: Coordination across cores, load balancing, and scheduling: Per-core buffering also needs to be coordinated with the load balancing and scheduling algorithms running at the NIC. For example, a NIC-based load balancer agnostic of the priorities of requests enqueued at per-core buffers - e.g., "enqueueing a request at the shortest queue" - can easily lead to HoL blocking. Likewise, a NIC-based scheduler that simply dequeues highest priority requests buffered at the NIC and tries to enqueue them at per-core buffers may inadvertently stall both high and low-priority requests and lead to non-work-conserving behavior when the buffers at the cores serving high-priority requests are all full (Section 4).

C3: Lack of existing hardware: Existing hardware architectures are insufficient for precise on-NIC load balancing and scheduling. For example, modern hardware priority queues, notably PIFO [39], can only be used to provide programmable packet scheduling; we cannot support both programmable scheduling and load balancing with just the PIFO abstraction.

C4: Host and NIC Communication Overheads: To efficiently offload request load balancing to the NIC, the CPU needs to provide load feedback to the NIC at a fine granularity (*e.g.*, per-packet). Furthermore, with 100+ Gbps NICs, PCIe throughput can become the performance bottleneck even in combination with optimized software stacks [29]. Thus, it is necessary to ensure that the CPU and PCIe overheads of CPU-NIC communication are low.

Overall, for effective orchestration, we need new NIC architectures for offloading load balancing and scheduling, coupled with new algorithms, and new OS-NIC interfaces.

3 RingLeader Overview

RingLeader is a new NIC architecture and OS-NIC interface that enables efficient and precise orchestration. In RingLeader, scheduling and load balancing are performed in tandem by an efficient and precise novel hardware offload on the NIC, and core allocation is performed by a host datapath OS with information from a new OS-NIC interface. We aim our discussion at servers equipped with a single NIC; we discuss multi-NIC support in Sections 9 and 12.2.

3.1 System Assumptions

In RingLeader, we assume that an application runs multiple services, where each service processes a specific type of request (e.g., latency-sensitive reads vs. throughput sensitive scans). A service can be replicated using multiple instances running across cores to handle load (e.g., deploying many read-oriented instances to serve heavy read traffic). We assume that distinct services can share a core; but our system also applies to cases where services need to be isolated across cores.

In RingLeader, the host uses a Demikernel-like single address space datapath OS [42]. The datapath OS achieves 1) fast multiplexing between OS tasks (e.g., buffer management, I/O processing, core allocation, coroutine scheduling) and application-specific work, and 2) fast context switching between different services' computations. We have chosen to use a datapath OS to manage host services instead of a kernel-based OS, as the traditional kernel-based OS abstractions (such as threads or processes) impose high overhead in multiplexing and context switching [42]. Figure 4 shows the system running multiple services. Each service launches multiple coroutines[2] on multiple cores; the coroutines are scheduled and managed by the datapath OS. We assume that each service is designed to run well on multiple cores.

The datapath OS uses cooperative scheduling: a long-running coroutine will *yield* voluntarily after a few microseconds of running. The datapath OS schedules the highest priority runnable coroutine once a running coroutine yields. The policy for yielding and scheduling depends on cross-service priorities.

The RingLeader NIC *buffers* received packets. This is reasonable because commercial NICs have a large amount of memory (tens of MBs of SRAM and 4–16 GBs of DRAM) [2, 7, 25–27]. If additional buffer capacity is needed, host DRAM can be used to buffer packet data with the RingLeader NIC only buffering packet descriptors.

RingLeader is designed to operate regardless of whether the transport layer is implemented in the NIC or on the CPU. On-NIC transport enables RingLeader to easily load balance

[2] As defined in Demikernel, coroutines are light-weight user-level threads that encapsulate the OS or application computation.

and schedule at the RPC granularity, while on-CPU transport necessitates load balancing at the flow or flowlet granularity.

3.2 Key Ideas and Design Overview

RingLeader can schedule and load balance requests from different services in a given application; to this end, inter-service policies can be specified in RingLeader. We discuss how RingLeader can support policies *across* applications in Section 9. Furthermore, each service provides input to the RingLeader NIC to assist with scaling up/down the per-service allocated cores.

Ideas: RingLeader approximates an ideal centralized orchestration approach using the following ideas that address the challenges in Sec.2.3: (1) We employ *shallow priority queues* on each core (Sec. 4.2). The per-core coroutine scheduler prioritizes dequeuing certain requests from these queues to avoid HoL blocking inside the buffer. (2) The NIC uses a new Join-Bounded-Shortest-Ranked-Queue (JBSRQ) load balancing algorithm that utilizes the per-core priority queue behavior to inform load balancing decisions (Sec. 4.2). In addition, we develop a new priority-based on-NIC request scheduler called first-eligible-out (FEO) and a simple interface between the scheduler and the load balancer (Sec. 4.3); this helps coordinate the scheduler's dequeue actions with the load balancer by exposing available room at per-core buffers to the scheduler. (3) We present a novel NIC hardware architecture that uses a reduction tree to combine scheduling and load balancing decisions at line rate (Sec. 5). (4) We use memory-mapped IO and inlining metadata in packet descriptors to develop an efficient OS-NIC communication interface (Sec. 4.1).

Example: Figure 4 illustrates how RingLeader operates when network packets are received. For simplicity, we only focus on load balancing and scheduling. Here, two services are running on a host. When a request packet enters RingLeader, it is processed by a programmable match+action (RMT) pipeline, which parses the packet's L3-L7 packet headers as necessary to identify the service that the packet belongs to and compute appropriate ranks. Then the packet is enqueued into the per-service packet buffer queue waiting to be scheduled.

The on-NIC request scheduler uses the FEO queue to schedule different services' requests according to a programmable policy. FEO schedules the highest priority service for which there is available room at a core where the service can run (this "eligibility" is provided via a mask). The on-NIC load balancer then steers this highest priority service's request to an eligible core that has the lowest rank (akin to queuing time) as computed by JBSRQ.

On each host core, the coroutine scheduler launches the runnable coroutine corresponding to the highest priority request. After the request finishes processing, the datapath OS provides load feedback to the NIC through the TX packet's descriptor or a separate MMIO register write.

We conclude the overview with a few more details.

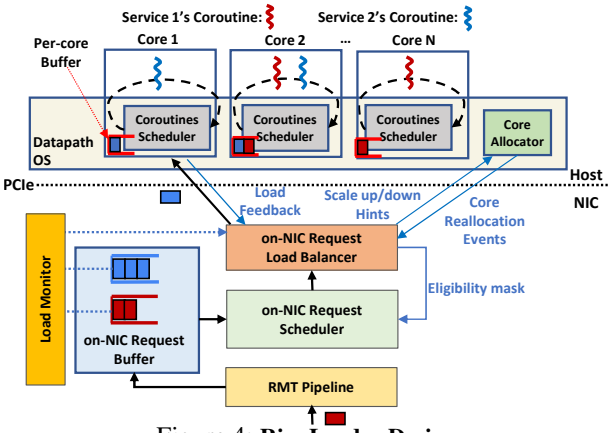


Figure 4: **RingLeader Design.**

Policies: RingLeader's load balancer, scheduler, and core allocator cooperate from top to bottom to enforce a given inter-service policy. Scheduling policies in RingLeader are expressed as a hierarchy of functions that compute the rank, rate, and/or transmission time for a packet [38–40]. Having a hierarchy of functions enables policies where multiple services can be grouped together, *e.g.*, two latency-sensitive services can be given equal priority over another service but different weights when competing with each other.

Core assignment: An on-NIC load monitor tracks the queuing condition for each service/request type. Each service can configure its trigger condition; when the scale-up/down threshold is met, the NIC sends a scaling hint to the least loaded core which runs this service/request type. The core allocator runs inside the datapath OS in a distributed fashion, *e.g.*, it can run on any core depending on which core receives the NIC hint.

4 RingLeader Design

We now discuss the design of the individual components of RingLeader. In Sec 4.1, we introduce the interface and mechanism that the NIC uses to communicate with the OS. In Sec 4.2, we introduce RingLeader's on-NIC request load balancer and our JBSRQ algorithm. In Sec 4.3, we describe the design of our FEO request scheduler and the non-blocking interface between the request scheduler and the load balancer. In Sec 4.4, we describe our NIC-assisted CPU re-allocation.

4.1 OS-NIC Interface

The OS-NIC interface in RingLeader (Table 1) is designed to minimize both HoL blocking latency and the CPU overheads of communicating orchestration metadata. We focus on the mechanisms here and outline the metadata exchanged over the interface at relevant places in later subsections.

CPU-to-NIC metadata: The datapath OS communicates with the NIC by writing to the NIC control registers via memory-mapped IO (MMIO) and via metadata in descriptors.

OS-to-NIC Interface	Description
RegisterService(s_id: X, ip: I, port: P, prio: O)	Register a new service X with the NIC.
EnableService(s_id: X, core_id : Y)	Notify the NIC that core Y is running service X.
DisableService(s_id: X, core_id : Y)	Notify the NIC that core Y is no longer running service X.
LoadFeedback(s_id: X, core_id : Y, count: C)	Notify the NIC that service X finishes C packets on core Y.
EnableLoadMonitor(s_id: X, trigger: T)	Enable load monitor for service X, with trigger condition T.
RearmLoadMonitor(s_id: X)	Notify the NIC that the host is ready to receive the next load hint for service X.
NIC-to-OS Interface	**Description**
LoadHint(s_id: X, hint: H)	Notify the host that service X's load has triggered the scale-up/down condition.

Table 1: **RingLeader OS-NIC Interface**

Each core accesses a different set of cache-aligned NIC registers to increase MMIO write performance. Our microbenchmarks in Section 7.4 show the throughput for OS-to-NIC communication is roughly 50M messages per second.

NIC-to-CPU metadata: The NIC communicates with the OS through packet descriptors. The NIC-generated reallocation hint is inlined into the packet descriptor and sent to the per-core NIC queue. [3] The datapath OS polls the NIC queue and parses the NIC hint. To avoid HoL blocking, the NIC limits the number of outstanding unACKed hints per core.

Our interface allows the NIC to monitor and control the length of each per-core queue despite the inherent asynchrony caused by PCIe latency. This design also overcomes PCIe throughput limitations by avoiding generating new PCIe messages in the common case.

4.2 On-NIC Load Balancing with JBSRQ

RingLeader performs hardware-based request load balancing for each service using a Join-Bounded-Shortest-Rank-Queue (JBSRQ) algorithm to decide **when** and **where** to send a packet. JBSRQ is an extension of the Join-Bounded-Shortest-Queue (JBSQ) algorithm [21] that considers inter-service inference and priorities.

As defined in R2P2 [21], JBSQ(n) approximates an ideal, work-conserving single queue policy using a combination of an on-NIC centralized queue and short, bounded queues at each worker. Each worker queue has a maximum depth of n messages. JBSQ(1) is equivalent to a single-centralized-queue model, whereas JBSQ(∞) is equivalent to JSQ.

Per-core shallow priority queues: JBSRQ approximates centralized pull-based load balancing (which achieves ideal load distribution) using a combination of an on-NIC buffer and *shallow* bounded-size (*e.g.*, 4 requests) per-core queues.

When multiple services with different priorities co-exist in the same core, the per-core buffers (no matter how small) can cause undesirable HoL blocking. To avoid this HoL blocking, we implement the per-core buffers as *software priority queues*; a core's coroutine scheduler uses the priority queue to enforce lightweight prioritized scheduling. The enqueue overhead of this priority queue is minimal given that the queue depth is ≤ 4, and priority calculation overhead is eliminated by the fact that the NIC scheduler computes priorities (Section 4.3) and simply carried along with packet descriptors. Further, the currently running lower priority request will yield to the highest priority request, which further reduces HoL blocking.

JBSRQ: Before describing our approach, we outline the sub-optimality of the classical join-bounded-shortest-queue (JBSQ) approach.

The main issue is that JBSQ does not consider the behavior of the host's priority queue.

Figures 5 (a), (b) show this limitation for two types of JBSQ algorithms: global-JBSQ and per-service-JBSQ. In global-JBSQ, which is used in RackSched [43], the NIC tracks per-core NIC queue lengths and always steers new requests to the core with smallest queue length. In per-service-JBSQ, which is used in nanoPU [17], the NIC tracks per-service queue length on each core and implements a JBSQ per service.

Given two services running on core 1 and core 2 where service A's priority is higher than service B, the example in Figure 5 (a) shows how global-JBSQ prevents new arriving high priority requests from preempting the on-host low priority requests. Since core 1's queue length is larger than core 2, global-JBSQ would dispatch the newly arrived service A's request to core 2. However, the optimal decision is to steer the request to core 1 as A's request would be served before B's request; the low priority request's queue length has little impact on the high priority request's completion time.

Similarly, Figure 5 (b) shows that per-service-JBSQ leads to sub-optimal performance for low priority requests.

To overcome JBSQ's limitations, we introduce JBSRQ. For simplicity, we assume each service has a single request type and that we are given priorities across services. In JBSRQ, the NIC tracks same-core services' queue lengths and service priorities. Then, for each service's request, the NIC selects the core that has the minimal rank, where rank is calculated as follows for a request for service A:

$$R[A].c = \sum_{P_x \geq P_A} Q[x].c + \lambda * \sum_{P_x < P_A} Q[x].c$$

Here, $R[A].c$ represents service A's rank on core c. P_A represents service A's priority, $Q[x].c$ represents service x's queue length on core c. λ is a constant factor between 0 and 1.

The underlying idea in JBSRQ is: when dispatching one service A's packet, the load balancer should consider the amount of the queue on a core that is contributed by requests of at least the same priority as A (because A's request cannot be scheduled ahead of such requests); the first term captures this. The rank calculation ignores the queue length contribution from all lower-priority requests. The factor λ and the sum-

[3] If there is no active packet descriptor being sent from the NIC to the host, RingLeader will generate a new packet descriptor (for scaling down).

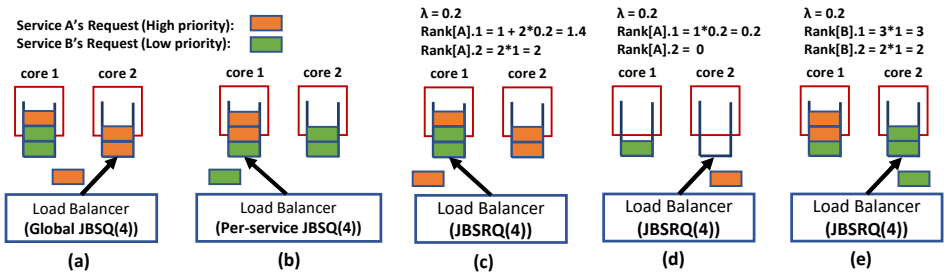


Figure 5: **Comparison between JBSQ and JBSRQ. In (b): Since core 1's low priority request queue length is smaller than core'2, per-service-JBSQ would dispatch the new arrived service B's request to core 1. However, the optimal decision is to steer the request to core 2. This is because a high-priority request's queue length would impact the low-priority request's completion time.**

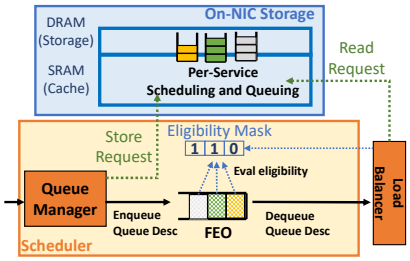


Figure 6: **On-NIC Request Scheduler**

mation in the second term captures the cost of waiting for a lower-priority request to yield before the higher-priority request is scheduled.

We now exemplify the benefit of using the JBSRQ policy. Figure 5 (c) shows that, when dispatching A's request, we could mostly ignore B's queue length. The calculated core 1's rank is smaller than core 2. Thus the newly arriving A's request is steered to core 1. Figure 5 (d) shows that, at low load, B's queue length can influence the load balancing policy for A's request; this is because we have added a small constant factor for the low priority request's queue length, which allows B to obtain fair service at low load. In this example, selecting idle core 2 is the optimal decision. This is because the overhead of scheduling or preempting B's request is non-negligible; thus choosing core 1 leads to a sub-optimal decision. Figure 5 (e) shows that when dispatching B's request, we should consider A's queue length, as a high priority request will always be served before a low priority request.

4.3 Non-blocking On-NIC Request Scheduler

We develop an FEO (First-Eligible-Out) priority scheduler that provides programmable per-cycle scheduling while supporting a non-blocking interface with the load balancer.

To understand why FEO is needed, consider PIFO [39], which assumes that, at any given time, all elements are eligible for scheduling. PIFO always schedules the smallest ranked element in the entire list of enqueued requests. However, given that we use shallow per-core buffers, we require that, for a given service, if a request's rank at all worker cores exceeds the queue-length bound, the load balancer must *hold* the request to avoid it getting dropped at a core. In this situation, the scheduler would block the rest of the lower-priority requests, which the load balancer could have dispatched to other potentially-idle cores.

FEO extends PIFO to avoid this problem by interfacing with the load balancer. As Figure 6 shows, when dequeuing elements, we first filter the set of elements eligible for dispatch and then schedule the smallest ranked element from that set. To enable this, the load balancer provides a bitmask that records the dispatching eligibility of each service.

The main difference between PIFO and FEO is the dequeue operation, which proceeds in two steps. First, we evaluate each element's eligibility in parallel by looking up its bit in the bitmask. Second, FEO uses priority encoding to select the front-most element whose eligibility is true, pops out the selected entry, and shifts the array. This design achieves a fast and parallel evaluation of elements upon dequeue. Additionally, FEO also provides buffer isolation by ensuring that the lowest priority request is dropped when buffers overflow.

4.4 NIC-assisted CPU Assignment

In RingLeader, each service could enable its own on-NIC load monitor through the interface defined in Table 1. RingLeader's load monitor supports rich triggers based on services' performance goals (e.g., latency or throughput) or scheduling policies. By default, scale-up trigger uses the congestion detection policy in Caladan [13]: if any service's request is found to be present in the on-NIC queue for two consecutive intervals, the NIC generates a scale-up hint. The scale-down policy is more conservative: within a time interval, if the maximum on-NIC queue length for a service never exceeds a threshold, the NIC generates a scale-down hint.

When a threshold is reached for a service, the load monitor generates a scale-up/down hint, inlines into the packet descriptor, and sends it to the buffer of the service's least-loaded core; it then *disarms* hint generation for this service. If no active packets are sent from the NIC to the cores, RingLeader will generate a new packet descriptor (to aid scale down).

The datapath OS polls the per-core queue and receives NIC hints. Then, the OS calls the core allocation function to decide

whether/where to scale up/down this service.

Assignment strategies: In an ideal system with a perfect load balancing policy and no multiplexing overhead, the best core allocation policy would be **complete-share**: similar to Shinjuku [19], all services run on all the cores, and a service is immediately granted CPU when its request is dispatched. This policy can tolerate bursts well and ensure good CPU efficiency. In practice, though, multiplexing overheads (including preemption and yielding) are non-negligible. For example, even with state-of-the-art low-overhead interrupt mechanisms, multiplexing tasks in a core could incur at least 24% overhead [15, 19]. Therefore, frequent switching between services waste considerable CPU resources under complete-share.

Thus, RingLeader supports two additional core assignment strategies to balance the trade-off between burst tolerance and wasted CPU. In the **no-sharing dedicated** model (similar to Shenango [31], Caladan [13]), each service has its own dedicated core set. The core allocator reallocates cores between services at fine-granularity (e.g., 5 μs per reallocation). A dedicated core improves cache locality and avoids multiplexing overhead. But such a system will have worse burst tolerance since even a 5 μs reallocation interval cannot react to transient micro bursts [24].

In the **allow-sharing hybrid** model, each service has some dedicated cores, as well as cores shared with other services. The dedicated core is used to handle the long-term constant load, and the shared core is used when a burst of requests arrives. This balances multiplexing overhead and burst tolerance.

After the datapath OS successfully scales up/down a service, it calls the rearm function (Table 1) to rearm the load monitor for the service. Before being rearmed, the load monitor will not generate further hints for the service; this ensures only one in-flight hint per service and reduces the synchronization overhead inside the host's core allocator (e.g., only one core will receive the hint for a service at a time).

5 Hardware Design

We describe the hardware design of RingLeader's programmable load balancer and provide details on how it interfaces with FEO and with software priority queues. We end with an example to show how requests flow through the RingLeader hardware. We provide benchmarks in Sec. 7.5.

5.1 Load Balancer Hardware

The load balancer unit uses two fundamental building blocks:

Per-core rank register array: JBSRQ needs to sum up the queue lengths of all services/request types on a core at or above a certain priority (Sec 4.2). Instead of spending cycles scanning and summing up different queue lengths, we maintain a pre-calculated rank register array for each priority level in the on-NIC SRAM. This register array stores each priority level's current rank on each core. When a request arrives, RingLeader directly reads out its rank according to the priority. The rank register array is updated asynchronously either when a request is dispatched or when load feedback is sent back from the worker core.

Reduction-tree-based "choose min": We use a hierarchical tree-based circuit for computing the "choose min" operation in JBSRQ to select a core. Although PIFOs are typically used for "choose min" operations in scheduling, using the same design in our load balancer is not feasible. This is because ranks are frequently updated as requests are dispatched and completed, and it is not possible to update the ranks of entries in a PIFO. Using a new reduction-tree-based design in RingLeader overcomes this limitation and allows for ranks to be updated frequently and in parallel.

Because the "choose min" operation can also be costly when the core count is high, RingLeader uses a hierarchical tree-based circuit to compute this minimum value. This circuit lends itself to pipelining, and it can calculate a minimal ranked core at every cycle. In RingLeader, we found that with 64 cores, a 3 staged reduction tree pipeline can fit on a middle-end FPGA without any utilization or timing issues.

5.2 End-to-End Example

We now present a simple example that puts the hardware components of RingLeader all together. We have two active services running on a host with two cores; the services are prioritized as shown in the top right; all requests in service are the same priority. Figure 7 (A) shows each service's priority, as well as how five existing requests are queued on the host. We assume the λ in JBSRQ is 0.2, and the depth of each per-core queue is 3; the scheduling policy is a strict priority.

Our example is shown via the numbered steps in Figure 7: To start with, in the PIFO unit, both services' eligibility bit in the mask is true. Then, (1) PIFO schedules service 1's queue descriptor, with the highest priority (see top right). (2) The request scheduler reads service 1's request F from the request buffer. (3) Request F is sent to the load balancer, which then looks up the priority register and directly reads ranks from the register array. Priority 2's rank on core 1 is 0.4 and on core 2, rank is 1.4. (4) The load balancer checks the core bitmask. If a given service is not running on a core, the rank for this core is set to infinite. However, in this example, both cores run service 1, so the rank is not reset or modified.

(5) Per-core ranks are then sent to the hierarchical reduction tree to identify the destination core with the minimal rank, core 1. Then, request F is dispatched to core 1. (6) We update core 1's priority registers according to the JBSRQ algorithm (Sec 4.2). As shown in Figure 7's (B) table and looking at core 1 queue occupancy before enqueueing F at the top right, we add one to the ranks of both priority 1 and priority 2, and we add λ to priority 3's rank. [4]

[4] F belonged to a service with priority 2; after enqueueing it, priority 1's

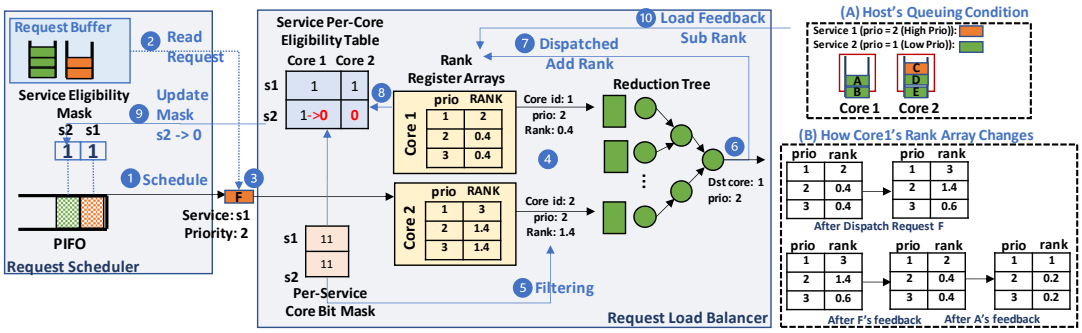


Figure 7: **End-to-End Example in RingLeader NIC**

(7) After the update, priority 1's rank on core 1 reaches the rank boundary (which is 3). We then update the per-core eligibility table - where we log which service on which core has reached the rank bound, meaning that buffers are exhausted at the core and requests from the service can no longer be scheduled on the core. Given service 2 is mapped to priority 1 on core 1, we change service 2's eligibility on core 1 into 0. (8) Since service 2 is now ineligible to be dispatched on all the cores (core 2's queue was already full – see top right), in the request scheduler, we set service 2's eligibility bit into 0. Therefore, PIFO will no longer schedule service 2's request. (9) Finally, Figure 7's (B) table further shows how the rank array is updated when receiving feedback from core 1 after processing requests F and A. [5]

6 Implementation

Our implementation consists of an FPGA prototype for the RingLeader NIC, a user space NIC driver, and a library operating system built over Demikernel.

FPGA-based Prototype: The FPGA prototype is implemented in 4K lines of Verilog code, and uses the DMA Engine, Ethernet MAC and PHY provided in Corundum [12] run at a 250 MHz frequency with a data width of 512 bits.

RMT pipeline: We implemented a single-stage RMT pipeline in our FPGA prototype. The datapath OS preinstalls the appropriate rules in the pipeline through the NIC-OS interface.

On-chip Request Buffer: The request buffer is implemented using high-speed BRAM, which supports concurrent reads and writes at 128 Gbps. The size of the on-chip BRAM buffer is set to 800 KB. This buffer size can be increased by utilizing on-NIC DRAM in the future.

FEO scheduler and reduction tree: In our implementation, the FEO block runs at a 125 MHz frequency with a queue size of 64. The reduction tree supports 64 worker cores with a three-stage pipeline in the dispatcher. In the rank register array, each core has 8 physical priorities.

User space NIC driver: The user space poll mode driver for the RingLeader NIC is implemented in 1.5K lines of C code and provides DPDK-like kernel-bypass access to the NIC for standard NIC functions, in addition to providing all of the functions in Table 1.

The Datapath OS: We integrated RingLeader with Demikernel's catnip libOS using 800 lines of Rust. We made the following modifications to Demikernel: (1) We extended the catnip libOS to add support for RingLeader's user space driver. (2) We added multi-core support to Demikernel, which previously only ran on a single core. (3) We extended Demikernel's coroutine scheduler to enforce prioritized scheduling between different services' coroutines. (4) RPC requests yield to the coroutine after a fixed amount of work instead of always running to completion (Section 3).

7 Evaluation

Our evaluation answers the following questions:

(1) Does RingLeader achieve high performance for load balancing and request scheduling? How does RingLeader's tail latency and scalability compare to the state-of-the-art software-only approaches across different workloads and service time distributions? (Sections 7.2 and 7.3)

(2) How much do the individual components of RingLeader contribute to overall improvements? (Section 7.4)

(3) How do our NIC-assisted core assignment's resulting CPU efficiency and burst tolerance compare to the state-of-the-art software-only approaches? (Section 7.5)

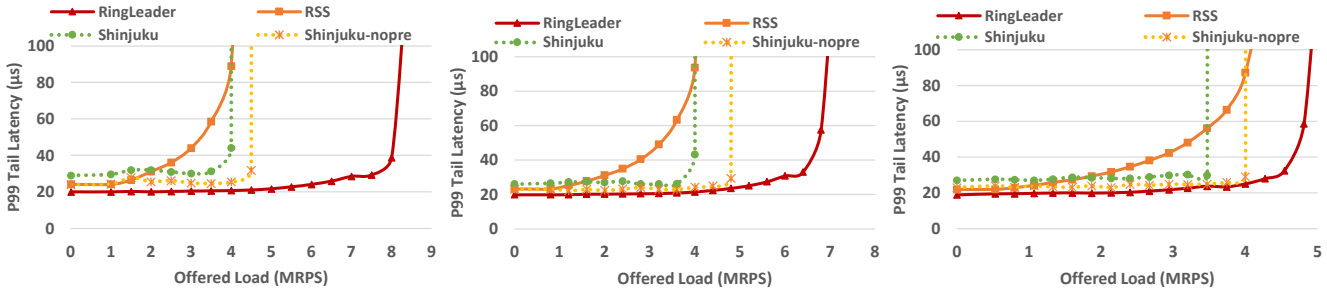
(4) What is the scalability and hardware resource usage of the RingLeader NIC? (Sections 7.6)

7.1 Methodology

Testbed: We evaluate our system on a server with two Intel Xeon Gold 6326 16-core (32-thread) CPUs and 128 GB of RAM. This server runs Ubuntu LTS 20.0.4 with the 5.4.0 Linux kernel. In addition, the server has a 100G Alveo U280 Data Center Accelerator Card [1] atop which we implemented our 100G FPGA prototype. The server also has a Mellanox ConnectX-5 Ex 100 Gb NIC, which we use to run the Caladan

rank will see all 3 entries in the queue; priority 2's rank will see priority 2 requests (1) + λ (=0.2) times priority 1 requests (2); priority 3's rank will see priority 3 requests (0) + λ (=0.2) times priority 1 and priority 2 requests (3).

[5] F's feedback comes before A because of the software priority scheduler.



(a) **30 workers** (b) **24 workers** (c) **16 workers**

Figure 8: Load balancing performance under Exp(3) workload.

baseline (described below). We use another client machine with a Mellanox ConnectX-5 Ex 100Gb NIC to generate load using DPDK. The client has the same CPU and OS version as the server. Our experiments (Ringleader and baseline experiments) don't consider NUMA and direct all interrupts, memory allocations, and threads to the NIC-local socket.

RingLeader configuration: Software priority queue depth is set to 4, and λ (Sec. 4.2) is set to 0.2. (We study sensitivity to λ in the appendix (12.1), and find $\lambda = 0.2$ to be a good setting). The yielding interval is set to 5μs.

Baselines: We compare RingLeader to three baselines:

Shinjuku [19]: Shinjuku uses centralized preemptive scheduling to achieve high-performance request load balancing and scheduling. Shinjuku only supports Intel 10G NICs and Linux kernel version 4.4.x, and it can only run on Intel cores because its fast preemption mechanism requires VT-x support. We use two Cloudlab [34] c6420 nodes (one client and one server, connected through a ToR switch) to run Shinjuku with kernel v4.4.0; each node is equipped with two 16-core (32-hyperthread) Intel Xeon Gold 6142 CPUs, and an Intel X710 10 Gigabit NIC. By default, Shinjuku uses two hyperthreads for orchestration – one for the network and another for the load balancer – collocated on the same physical core. The preemption interval is set to 5μs. To ensure a fair comparison, we always assign one more physical core (two hyperthreads) to Shinjuku than RingLeader for running orchestration tasks.

Caladan [13]: Caladan reallocates cores between applications at a fine granularity to increase CPU efficiency under changing workloads. We run Caladan on the same server and the same OS and kernel version as Ringleader.

Caladan runs its IOKernel on a single dedicated core. Therefore, like Shinjuku, we always assign one more physical core to Caladan than RingLeader.

RSS: We also study a decentralized RSS-based system. In this baseline, worker hyperthreads are managed by the Demikernel datapath OS, and each worker polls its own large receiving NIC queue. Here, we use a bare metal 100G U280 FPGA NIC [1] that performs standard NIC functions.

Workloads: We employ both synthetic workloads and RocksDB.

Synthetic Workloads: (Table 2) Our synthetic workload is a server application where requests perform dummy work that we can control to emulate any target distribution of service times. This allows us to run microbenchmarks that systematically study how RingLeader and different baselines perform under different performance limits.

Workloads	Description
Exp(3)	Single request type, service times follows exp distribution with mean 3μs.
Bimodal (95-5,5-100)	95% requests are high priority, take 5μs. 5% requests are low priority, take 100μs.
High Bimodal (99-3,1-100)	99% requests are high priority, take 3μs. 1% requests are low priority, take 100μs.

Table 2: **Synthetic Workloads**

RocksDB Workloads: We also performed experiments with RocksDB, a popular and widely deployed in-memory key-value store developed by Facebook [11]. We use RocksDB queries that are either GET/PUT requests or range SCANs.

To generate both the synthetic and RocksDB workloads, we developed an open-loop load generator similar to Shinjuku [19] that generates requests over user space UDP. It uses 12 threads to generate requests following a Poisson arrival process and specific service-time distributions and another 12 user space threads to receive server replies. Request latency is measured through timestamps carried inside packets. We ensure that the network speed and the load generator are not bottlenecks in any experiment by checking for packet drops.

7.2 Load Balancing Performance

First, we evaluate the RingLeader load balancing unit using an Exp(3) workload that, for simplicity, only has a single type of request (with service times following an exponential distribution with a mean of 3μs). We compare RingLeader against three baselines: Shinjuku, Shinjuku-nopre, and RSS. In Shinjuku-nopre, we use Shinjuku without preemption.

Figure 8 shows the load balancing results when the server runs 30, 24, and 16 workers. Each worker is a hyperthread, and all workers run on the NIC-local socket. Across all levels of load, RingLeader provides the lowest tail latency. Also, RingLeader has the highest saturating throughput for all worker counts. This shows that RingLeader has the best scalability and load balancing precision. In contrast, the dedicated orchestrator core Shinjuku uses for networking and

load balancing becomes a performance bottleneck when the offered load is $>$ 4.8 MRPS (Million-Request-Per-Second) and preemption is disabled and when the offered load is $>$ 4 MRPS and preemption is enabled. Figure 8 also shows that RingLeader consistently outperforms RSS, which distributes load unevenly across cores, hurting tail latency.

7.3 Scheduling Performance

Synthetic Workload: We now study RingLeader's ability to achieve high-performance scheduling across services/request types. We use the High Bimodal workload (Table 2) with two types of requests for one service: high priority requests that follow Exp(3) and low priority requests that follow Exp(100). We turn off core assignment in these experiments, so the two types of requests run on all worker hyperthreads.

Figure 9 shows the results from this experiment. For all worker counts, RingLeader consistently outperforms Shinjuku. This is because Shinjuku's orchestrator cores become bottlenecked when the load is larger than 3.5 MRPS. In contrast, RingLeader has better scalability and lower latency.

Also, in RingLeader, high priority requests can still maintain low tail latency even when the low priority requests' load is saturated (*e.g.*, at load $>$ 3.5 in Fig 9a). This is because the on-NIC scheduler provides buffer isolation (Sec. 4.3) and ensures each request type is dropped separately.

RocksDB Workload: Next, we evaluate RingLeader's scheduling performance under the RocksDB workload. We use two request types: GET requests for a single key-value pair that execute within 5μs; SCAN requests that scan 200 key-value pairs and require 60μs. We also vary the yielding interval for SCAN across 40 items-per-yield (Y40), 20 items-per-yield (Y20), and 10 items-per-yield (Y10). Figure 10 shows that RingLeader's prioritized scheduler allows GET requests to avoid long queuing times due to SCAN requests. Aggressive yielding improves tail latency performance for short requests and adds a constant overhead to scan requests. RingLeader-assisted core re-allocation is a way to get around the constant yielding overhead.

7.4 Benefits of RingLeader Components

We now study how the individual components in RingLeader's load balancing and scheduling functionality contribute to overall performance; core assignment is turned off here (we study it later in Sec. 7.5). Figure 13 (in appendix) presents a comparison of RingLeader and reduced versions of RingLeader that remove/replace a single component . We use the bimodal workload shown in Table 2.

FEO: Here, we turn off our scheduler eligibility bitmask (Section 4.3), causing it to be degenerate to vanilla PIFO (Blocking_PIFO). Figure 13 shows that PIFO's performance is much worse than RingLeader when the load increases for high priority requests. This is because, in our two-request-type setting, low priority requests prevent high priority requests from entering the load balancer at high load.

Global-JBSQ: Next, we evaluate global-JBSQ(4), a load balancing algorithm similar to RackSched [43] where the NIC tracks per-core queue lengths and always steers new requests to the core with the smallest queue length. The queue length bound for each core is set to 4. Figure 13 shows that, even with preemption and the software priority queue enabled, the high priority request's tail latency is much worse than JBSRQ when the offered load is $>$ 2.24 MRPS. Because global-JBSQ does not consider the behavior of the software priority queue, a burst of long requests can occupy all per-core NIC queues, and new arriving high priority requests cannot be dispatched.

Per-service-JBSQ: We evaluate per-service-JBSQ(4), a load balancing policy that is similar to nanoPU [17] where the NIC implements JBSQ(4) per service. On each worker, the queue length limit for a service is 4. Figure 13b shows that low priority requests have worse performance than RingLeader because, when dispatching a low priority request, the NIC ignores the influence of high priority requests on the queuing delay of low priority ones.

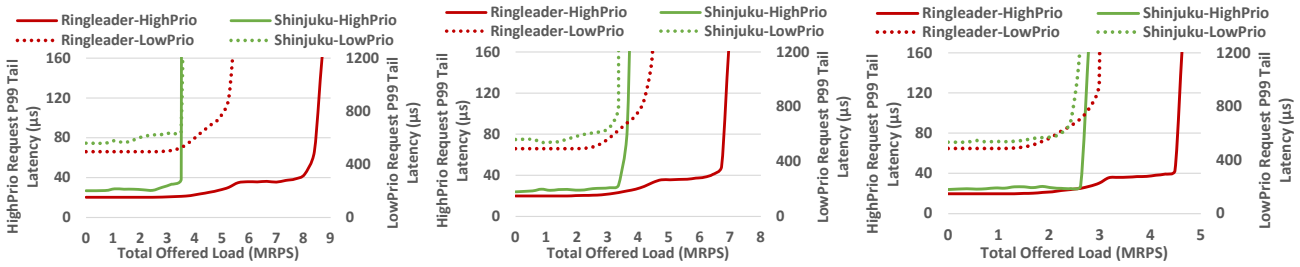
No Software Priority Queue: Figure 13a shows what happens when we disable per-core software priority queues (and cooperative yielding). High priority requests suffer a lot because a burst of low-priority requests can enqueue at currently idle cores and unduly delay later-arriving sensitive requests.

7.5 NIC-Assisted Core Assignment

We evaluate RingLeader's ability to detect load changes and aid in fast core reallocation. We experiment with two types of services running on the host. One serves high-priority latency-sensitive RocksDB GET requests. The other runs a best-effort analytics workload that continuously scans a range of the RocksDB database and performs data comparisons over the scanned results. We increase RocksDB GET's load gradually and measure offered load averaged over 10s intervals.

We compared RingLeader's core reallocation performance with Caladan. In this experiment, Caladan and Ringleader have 16 worker hyperthreads, and the core assignment decision interval is 8μs. Furthermore, in this experiment, Ringleader uses the same CPU assignment strategy as Caladan, which is the no-sharing dedicated model.

We evaluate the analytics service's throughput and the GET request's tail latency. Figure 11 shows that RingLeader keeps the tail latencies of the GET request low while also allowing for spare CPU cycles to be shared with the best-effort analytics service. Furthermore, RingLeader yields both better GET requests tail latency and higher analytics workload throughput than Caladan because load-imbalance and work-stealing in Caladan increase latency and CPU load. For example, in Figure 11b, Caladan's latency goes up at load 1.44 Mpps since work-stealing happens most frequently at this point. In contrast, RingLeader consistently achieves near-optimal



(a) **30 workers** (b) **24 workers** (c) **16 workers**

Figure 9: Load balancing and scheduling performance under High Bimodal workload.

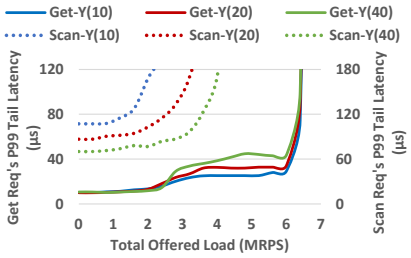
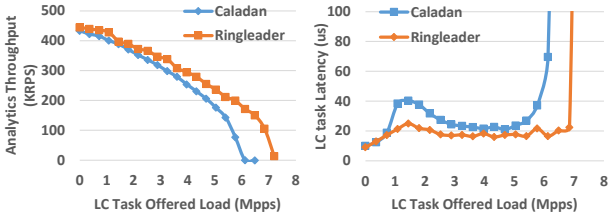


Figure 10: **RocksDB performance.**



(a) **Analytics throughput.** (b) **GET's P99 tail latency.**

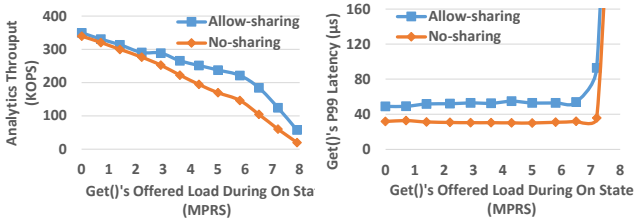
Figure 11: Comparison with Caladan.

centralized scheduling and low overhead core assignment.

Next, we use the on-off traffic pattern to compare the performance of two CPU allocation policies: allow-sharing hybrid and no-sharing dedicated (Section 4.4). During the on state, the traffic source generates GET requests; during the off state, the traffic source stops sending. The switching time between the on/off states is 0.8 ms. Under this pattern, core reallocation happens several times every 0.8 ms. Figure 12 shows that the allow-sharing policy has better analytics throughput as it allows the two services to coexist in the same core, accommodates small timescale bursts of arrivals, and minimizes CPU waste. However, the no-sharing has better tail latency for GET requests because using dedicated cores improves cache locality and avoids multiplexing overheads; nevertheless, the allow-sharing tail latency stays relatively low and flat for the most part. Given this information, an admin can configure RingLeader to pick a core assignment policy based on the relative importance of low tail latency for sensitive services versus not starving batch services.

7.6 Scalability and Resource Usage

We now study RingLeader's performance upper bound and it's hardware resource usage.



(a) **Analytics throughput.** (b) **GET's P99 tail latency.**

Figure 12: Core reallocation under different policies.

Module	Setting	LUTs(%)	BRAM(%)
Load Balancer	(16 priorities * 64 cores)	2.82	0.10
	(16 priorities * 128 cores)	2.86	0.10
	(32 priorities * 64 cores)	6.07	0.00
Scheduler	FEO = 16	0.24	0.01
	FEO = 64	1.00	0.01
Packet Buffer	800 KB	0.16	6.54

Table 3: **FPGA resource usage for different components.**

OS-NIC interface: Figure 16a shows the throughput for the OS-to-NIC interface. The communication throughput between a single worker hyperthread and the NIC is 6M register writes per second, and with 8 workers, the throughput can reach 50M. The result shows that RingLeader can achieve low-overhead, fast OS-to-NIC communication.

System Throughput and Latency Overhead: Figure 16b shows that RingLeader can achieve line-rate load balancing and scheduling. In this experiment, the host uses 8 worker hyperthreads, every request finishes immediately, and the rank bound is set to 16. The result shows that RingLeader achieves 100G with MTU-sized packets and 50Mpps for 64B packets.

We measure RingLeader's latency overhead by adding hardware timestamps. We find that a request can be scheduled and dispatched within 150 ns. The end-to-end host ping-pong latency is 6μs, which is close to commercial NICs.

Hardware Resource Usage: Our U280 FPGA has 1300k LUTs in total. Table 3 shows different components' resource usage under different settings. The load balancer and scheduler occupy most of RingLeader's on-chip logic. When the load balancer is configured with 16 priorities and 64 worker counts, it consumes around 2.82% of the logic area. With 32 priorities, it consumes 6.07%. Furthermore, when FEO uses 16 entries, it consumes 0.24% of the logic area, and when the size is 64, it consumes 1.00%. Overall, we find that RingLeader can easily fit on an FPGA.

8 Related Work

Software approaches: RingLeader addresses the key scalability and performance limitations of other orchestration systems like IX [6], ZygOS [33], Shenango [31], and Shinjuku [19]. ghOSt [16] and Syrup [20] use userspace CPU scheduling policies; they are complementary with RingLeader.

Hardware approaches: Shinjuku-on-SmartNIC [15] provides centralized preemptive request scheduling on an ARM-based SmartNIC, but scheduling requests on wimpy on-NIC cores has limited processing speed and introduces tens of microseconds of latency [22].

PIEO [38] extends PIFO to support efficient extraction for time-based scheduling algorithms, but it cannot simultaneously support scheduling and load balancing. This is because it only supports packet extraction as a function of time and hence cannot be used to support an eligibility mask.

Recent related works, such as nanoPU [17], RackSched [43], Shinjuku-Offload [15], and R2P2 [21], use JBSQ to offload load balancing in systems with communication latency. However, as demonstrated in Section 7.4, JBSQ is suboptimal when requests have different priorities. Our new JBSRQ algorithm can improve performance under multi-priority scenarios. Related works, such as RackSched, RPC-Valet [9], and nanoPU, offloads request scheduling. However, RackSched and RPCValet do not use centralized scheduling, which can cause high-priority requests to suffer more from HoL blocking. In the case of nanoPU, request scheduling requires changes to the CPU architecture by using the hardware thread scheduler. In contrast, RingLeader achieves centralized scheduling with no need for changes to the CPU architecture.

Elastic RSS [36] uses a NIC to perform both load balancing and core allocation. However, it buffers packets at CPU cores and not on the NIC, leading to load-imbalance, and it does not schedule packets, leading to HoL blocking.

Transports: Improvements in transport protocols are complementary to RingLeader. Both new transport protocols like MTP [41] and EQDS [30] and projects that offload transport protocols to SmartNICs [3, 28, 32, 37] can enable message-level load balancing and scheduling in an orchestration system, so RingLeader would benefit from their adoption.

9 Discussion

Multi-NIC support: Although our design as presented so far assumes a single NIC per server, there are a few different ways RingLeader can be configured to support multiple NICs in a single server: 1) a master/slave configuration (described below), 2) hard-partitioning workers (Section 12.2), or 3) a cooperative configuration (Section 12.2). In a master-slave configuration, each NIC will transfer data to main memory independently but not perform dispatching. Instead, each slave NIC sends descriptors about pending requests to the master NIC, which is solely responsible for orchestration.

Multiplexing Mechanism: RingLeader uses cooperative scheduling, requiring developers to insert yield statements for low priority services. However, RingLeader is also compatible with other multiplexing mechanisms, such as: 1) optimized APIC interrupts [15], and 2) compiler interrupts [5]. Optimized APIC interrupts are a low priority service that can set a timer that will deliver a low-overhead interrupt once the time slice expires. Compiler interrupts use compile-time instrumentation to allow programs to call an interrupt handler at a regular intervals with little performance impact.

Multi-process Support: RingLeader inherits a key assumption in Demikernel today [42], namely that the data path OS and services run in a single process. However, similar to recent work like Snap [23], we can extend RingLeader to support multiple processes by using Demikernel as a standalone process that multiplexes I/O across client processes through shared memory regions. Such an extension would naturally enable RingLeader to support policies across applications (as opposed to policies across services in an application).

Applicability to a general kernel: Fine-grained multiplexing between services on the same core is too expensive for μs-scale applications in a traditional kernel. This is why many previous works [13, 19, 31, 42] and RingLeader use highly specialized data path OSes. However, our system can also be applied to existing Linux kernels. With a general kernel, users may want to avoid processor sharing by isolating services across cores, and our load balancer and CPU allocator still work effectively.

10 Conclusions

Existing intra-server orchestration approaches have limited scalability, poor precision, and high overheads. We address these problems by introducing RingLeader, a new system that efficiently offloads orchestration in their entirety to a programmable NIC while minimally onloading limited functions to host cores. RingLeader introduces a novel OS/NIC interface, a new load balancing algorithm and scheduler, and a hardware element that combines the decisions of the two. Our experiments with a prototype on a 100 Gbps FPGA NIC show that RingLeader offers good tail latency, high throughput, good CPU utilization, and effective core reallocation.

11 Acknowledgements:

We thank our shepherd, Mina Tahmasbi Arashloo, and the anonymous NSDI reviewers for their feedback that significantly improved the paper. We thank CloudLab [34] and Christopher Rossbach for providing equipment used to test and evaluate RingLeader. This research was supported by NSF awards CNS-2214015, CNS-2202649, and CNS-2207317. Jiaxin Lin is supported by a Meta PhD Research Fellowship.

References

[1] Xilinx alveo u280. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[2] Alpha Data. ADM-PCIE-9V3 - High-Performance Network Accelerator. https://www.alpha-data.com/pdfs/adm-pcie-9v3.pdf.

[3] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 93–109, 2020.

[4] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[5] N. Basu, C. Montanari, and J. Eriksson. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1249–1263, 2021.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. {IX}: A protected dataplane operating system for high throughput and low latency. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 49–65, 2014.

[7] Broadcom. Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[8] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding host network stack overheads. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM. Association for Computing Machinery, 2021.

[9] A. Daglis, M. Sutherland, and B. Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[10] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 621–637, 2021.

[11] Facebook. RocksDB. http://rocksdb.org/.

[12] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46. IEEE, 2020.

[13] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[14] H. Golestani, A. Mirhosseini, and T. F. Wenisch. Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 337–350, 2019.

[15] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 60–68, 2019.

[16] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis. GhOSt: Fast and flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2021.

[17] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, N. McKeown, and C. Kim. The nanopu: Redesigning the cpu-network interface to minimize rpc tail latency. *arXiv preprint arXiv:2010.12114*, 2020.

[18] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 489–502, 2014.

[19] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 345–360, 2019.

[20] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2021.

[21] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. {R2P2}: Making {RPCs} first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[22] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333, 2019.

[23] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[24] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[25] Mellanox Technologies. Innova - 2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.

[26] Mellanox Technologies. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[27] Mellanox Technologies. NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[28] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020.

[29] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[30] V. Olteanu, H. Eran, D. Dumitrescu, A. Popa, C. Baciu, M. Silberstein, G. Nikolaidis, M. Handley, and C. Raiciu. An edge-queued datagram service for all datacenter traffic. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[31] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 361–378, 2019.

[32] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the RPC tax in datacenters. In *Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO. Association for Computing Machinery, 2021.

[33] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[34] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. Association for Computing Machinery, 2015.

[36] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 71–77, 2019.

[37] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[38] V. Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379, 2019.

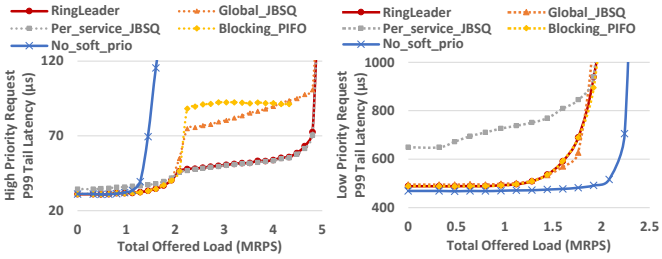
[39] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[40] B. Stephens, A. Akella, and M. Swift. Loom: Flexible and efficient {NIC} packet scheduling. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 33–46, 2019.

[41] B. E. Stephens, D. Grassi, H. Almasi, T. Ji, B. Vamanan, and A. Akella. Tcp is harmful to in-network computing: Designing a message transport protocol (MTP). In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets. Association for Computing Machinery, 2021.

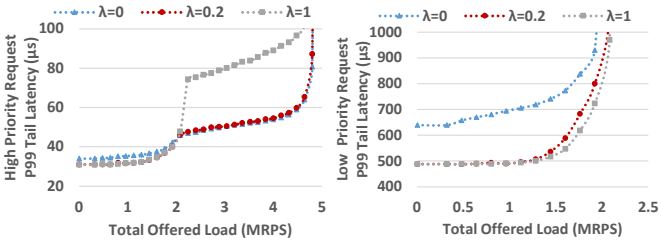
[42] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The Demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2021.

[43] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1225–1240, 2020.



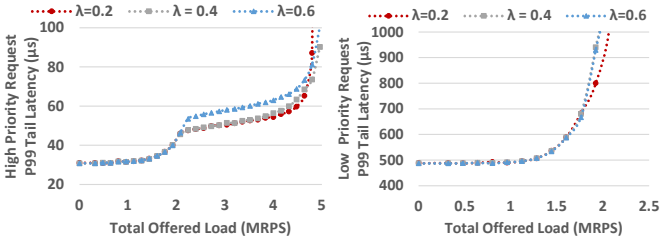
(a) **High Prio's P99 tail latency** (b) **Low Prio's P99 tail latency**

Figure 13: Comparison of RingLeader and reduced versions.



(a) **High Priority Request's P99 tail latency** (b) **Low Priority Request's P99 tail latency**

Figure 14: JBSRQ's performance under different λ.



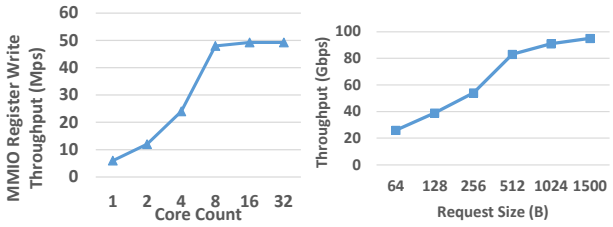
(a) **High Priority Request's P99 tail latency** (b) **Low Priority Request's P99 tail latency**

Figure 15: JBSRQ's performance under different λ (0< λ <1).

12 Appendix

12.1 JBSRQ Policy parameters

We evaluate how the constant factor λ in JBSRQ influences system throughput and tail latencies under the Bimodal workload. Figure 14 shows that, for both high priority and low priority requests, a constant λ between 0 and 1 brings significant performance gain over $\lambda = 0$ or $\lambda = 1$. When $\lambda = 0$, the rank calculation does not consider preemption overheads. Under low load (load < 2.24), $\lambda = 0$ leads to unnecessary preemption, increasing tail latency for both high priority and low priority requests. When $\lambda = 1$, JBSRQ equals global-JBSQ. This prevents new arriving high priority requests from preempting the on-host low priority requests when load > 2.24, increasing tail latencies for high priority requests. An intermediate λ is necessary to achieve good performance under both high and low load. Figures 15 shows how different λs between 0 and 1 influence JBSRQ's performance. There is not much difference between $\lambda = 0.2$ and $\lambda = 0.4$. When λ grows to 0.6, performance drops because of the same issue with $\lambda = 1$.



(a) **MMIO throughput.** (b) **RingLeader throughput.**

Figure 16: Scalability of RingLeader

12.2 Supporting Multiple NICs

While our design aimed at servers equipped with a single NIC, we believe that it can be extended to support multi-NIC settings as well. We discuss a few options below.

Hard-partitioning workers: The simplest approach is to partition workers across NICs. Here, for example, each NIC orchestrates its local NUMA node. This allows each NIC to perform ideal centralized scheduling independently. However, this requires all network traffic for a service to be sent to a specific NIC.

Cooperative multi-NIC orchestration: In this scenario, each worker can receive packets from multiple NICs. The control message in Table 1 is replicated across all NICs to achieve cooperative orchestration. When a request finishes, the worker sends load feedback to all NICs, so that every NIC knows up-to-date worker queue lengths. Also, when a NIC dispatches a request to a worker, the NIC notifies other NICs of this action. Replicating control messages is the primary trade-off introduced by this approach. However, this is unlikely to be a bottleneck given that control messages are small and MMIO throughput (16a) over PCIe is high.

StarryNet: Empowering Researchers to Evaluate Futuristic Integrated Space and Terrestrial Networks

Zeqi Lai[†‡], Hewu Li[†‡*], Yangtao Deng[†], Qian Wu[†‡], Jun Liu[†‡], Yuanjie Li[†‡], Jihao Li[†], Lixin Liu[†]
Weisen Liu[†], Jianping Wu[†‡]
[†]*Tsinghua University,* [‡]*Zhongguancun Laboratory*

Abstract

Futuristic integrated space and terrestrial networks (ISTN) not only hold *new opportunities* for pervasive, low-latency Internet services, but also face *new challenges* caused by satellite dynamics on a global scale. It should be useful for researchers to run various experiments to systematically explore *new problems* in ISTNs. However, existing experimentation methods either attain realism but lack flexibility (*e.g.,* live satellites), or achieve flexibility but lack realism (*e.g.,* ISTN simulators).

This paper presents StarryNet, a novel experimentation framework that enables researchers to conveniently build credible and flexible experimental network environments (ENE) mimicking satellite dynamics and network behaviors of large-scale ISTNs. StarryNet simultaneously achieves constellation-consistency, networked system realism and flexibility, by adopting a real-data-driven, lightweight-emulation-aided approach to build a digital twin of physical ISTNs in the terrestrial virtual environment. Driven by public and real constellation-relevant information, we show StarryNet's acceptable fidelity and demonstrate its flexibility to support various ISTN experiments, such as evaluating different internetworking mechanisms for space-ground integration, and assessing the network resilience of futuristic ISTNs.

1 Introduction

Thanks to the resurgence in the space industry [41, 46, 48], big competitors such as SpaceX and Amazon are actively planning and deploying hundreds or even thousands of broadband satellites in low earth orbits (LEO). Such emerging mega-constellations (*e.g.,* Starlink [34], Kuiper [9]) can be integrated into existing terrestrial Internet, *i.e.,* constructing an integrated space and terrestrial network **(ISTN)** to: (1) provide pervasive last-mile network access; (2) enable low-latency and high-bandwidth Internet transit [40, 52, 56]; and (3) facilitate efficient acquisition and delivery for big data from space (*e.g.,* earth observation images) [44, 74, 77, 78].

While holding great promise, several unique characteristics of LEO satellites (*e.g.,* high LEO dynamics) impose new challenges at various layers of the ISTN networking stack, and open a door to many new research problems, such as: (1) how should LEO satellites and ground facilities be interconnected to provide low-latency and continuous network services? (2) how should satellite routers be integrated into existing terrestrial Internet routing? and (3) are current constellation and protocol designs resilient enough to satellite failures in complex and harsh space environments? With many *unexplored* problems facing the "NewSpace" industry, it is thus foreseen that in the near future, there will be a surge of new ideas on the system and networking research relevant to ISTNs. But, *how can researchers build an experimental network environment **(ENE)** to test, evaluate and understand their new thoughts?*

Typically, existing approaches for creating an ENE can be classified into three categories: (1) live networks or platforms [7, 20, 34, 75, 81], which allow experiments in real deployments; (2) network simulation [60, 61, 76], which uses discrete events to model and replicate the behavior of a real network; and (3) network emulation [6, 55, 68, 69], which can test real applications/protocols in a virtual network. However, as will be illustrated in §3, all existing approaches have their limitations in creating a desired ENE for ISTNs: (1) the *feasibility* and *flexibility* of live satellite networks are *technically* and *economically* limited for normal researchers; (2) the abstraction level of simulation might be too high to capture low-level system effects, hiding practical issues such as the resource competition under heavy workload, energy drain or software errors; (3) existing emulators fail to characterize the high dynamicity of LEO satellites and thus are insufficient to build an experimental environment with acceptable fidelity.

The key challenge of building an expected ENE for ISTN research is: it is difficult to *simultaneously* achieve *realism* and *flexibility* in the experimental environment. First, terrestrial devices inherently lack the ability to reasonably mimic the high dynamics, system and network behaviors of realistic satellites. Second, mega-constellations typically consist of thousands of satellites. Thus the network scale required by an

*Hewu Li is the corresponding author.

ENE for mega-constellations might be far more than the extent supported by existing ENE methods (*e.g.,* [54, 55, 69]). Third, as a large number of satellites simultaneously move at a high velocity, continuously mimicking such frequent variations at scale could involve significant system overhead on the ENE.

This paper presents StarryNet, an integrated experimentation framework that empowers researchers to conveniently build ENEs with acceptable realism, flexibility and cost (*e.g.,* requiring only a few number of local/cloud machines) to satisfy various experimental requirements of ISTNs. The design of StarryNet is inspired by a key insight obtained from the satellite Internet ecosystem: many organizations or communities in this ecosystem have released and shared their constellation-relevant data, including regulatory information [2, 73], satellite trajectories [16, 38], ground station distribution [26, 39] and measurements from user terminals [32, 33], *etc.* Therefore, the key idea behind StarryNet is to build an experimental *digital twin*, *i.e.,* a virtual presentation of a physical ISTN, in terrestrial environments by: (1) leveraging terrestrial machines to virtualize a large number of lightweight virtual nodes to emulate satellites in mega-constellations; and (2) exploiting a *crowdsourcing* approach to collect, combine and use realistic constellation information to drive the emulation of *spatial* and *temporal* characteristics of ISTNs.

To achieve acceptable realism, StarryNet employs a *constellation synchronizer* based on realistic constellation-relevant information to make the virtual ENE as consistent as possible to a real ISTN, such as: (1) *constellation consistency:* the ENE is built with the same scale of a physical mega-constellation, where each node emulates a satellite, a ground station or a terrestrial host. The spatial and temporal characteristics, such as time-varying satellite locations and inter-visibility, are also configured and updated in each node based on our data-driven model-based analysis; (2) *system and networking stack consistency:* the ENE can support the run of unmodified applications as in real deployments; and (3) *capability consistency:* network and computation capabilities in the ENE are configured based on real hardware specifications. Further, to flexibly support various ISTN experiments and mimic *large-scale* and *highly-dynamic* mega-constellations, StarryNet adopts a *constellation orchestrator* that judiciously schedules and manages system resources on multiple machines to collaboratively construct ENE on demand.

We evaluate the ability of StarryNet based on real constellation information in two steps. First, we show the acceptable fidelity of StarryNet by comparing the experiment results obtained by StarryNet with live satellite networks and other state-of-the-art ISTN simulators. Second, facing futuristic ISTN scenarios, we demonstrate StarryNet's flexibility by conducting three case studies to: (1) explore the trade-space of various space-ground inter-networking mechanisms; (2) evaluate the resilience of routing protocols in various constellation designs; and (3) perform hardware-in-the-loop tests to measure system effects under various workloads.

Summarily, this paper makes the following key contributions: **(1)** we design StarryNet, a data-driven, emulation-aided ISTN experimentation framework (§4); **(2)** we implement StarryNet with a collection of open APIs for creating and manipulating user-defined ENEs (§5); **(3)** we evaluate and analyze StarryNet's experimentation overhead and fidelity (§6), and show StarryNet's flexibility (§7) by conducting various case studies driven by realistic constellation information. StarryNet is now available at: https://github.com/SpaceNetLab/StarryNet.

2 Preliminaries

Integrated space and terrestrial networks (ISTN). Recent satellite operators/organizations are actively developing their mega-constellations [4, 8, 9, 25, 34, 36], with hundreds to thousands of low earth orbit (LEO) satellites working together as a system. These satellites can be equipped with high-speed inter-satellite links (ISLs), and construct an LEO satellite network **(LSN)**. An LSN can further be integrated into existing terrestrial Internet via globally distributed ground stations [3, 26, 39] and very-small-aperture terminals (VSAT) [31], constructing an *integrated space and terrestrial network (ISTN)* that promises to provide pervasive, low-latency, broadband Internet services [40, 52, 56, 57] for terrestrial users globally.

Unique characteristics of ISTNs, as well as the new challenges. Two critical characteristics differentiate LSNs from existing terrestrial networks, and involve new challenges on the integration of satellites and terrestrial Internet. First, LEO satellites are moving at a high-speed with the respect to the earth surface. An LEO satellite might be visible for a certain ground vantage point only within *a few minutes* in one pass. Such high dynamics could inevitably result in technical challenges (*e.g.,* frequent connectivity disruptions and routing re-convergence) at the networking stack of ISTNs. Second, while evolved, resources (*e.g.,* bandwidth, CPU, energy) are still limited and costly in space, as compared with terrestrial network systems. Resource-intensive technologies might not be doable for resource-constrained satellites to sustain good network performance (*e.g.,* applying sophisticated network coding techniques for packet recovery in remote space).

Call for new research for futuristic ISTNs. The above characteristics and challenges accordingly raise a series of *unexplored* research problems in ISTNs, such as: (1) ***topology:*** how should LEO satellites and ground facilities be interconnected under the high space-ground dynamicity? (2) ***routing:*** how should we integrate hundreds or thousands of LEO satellites into Internet routing and tackle the potential performance degradation due to intermittent connectivity? (3) ***system effect:*** how much energy would a new functionality consume in space under various workloads? It is foreseeable that in the near future, in parallel with the evolution of real mega-constellations, there would be a surge of new research focusing on emerging satellite network systems. But, how should researchers test, assess and understand their new

Category / Tools		(i) Constellation Consistency	(ii) System and Networking Stack Realism	(iii) Flexible and Scalable Environment	(iv) Low-cost and Easy-to-use
Live LSNs or platforms	Live Starlink ([34])	✓	✓	✗	✗
	PlanetLab ([20])	✗	✓	✗	limited
	Emulab ([7])	✗	✓	✗	limited
Simulators and orbit analysis tools	STK ([35])	✓	✗	✓	limited
	GMAT ([11])	✓	✗	✓	✓
	SNS3 ([76])	for GEO only	✗	✓	✓
	Hypatia ([60])	✓	✗	✓	✓
	StarPerf ([61])	✓	✗	✓	✓
Emulators and variations	MiniNet ([55, 68])	✗	✓	✓	✓
	DieCast ([54])	✗	✓	limited at scale	✓
	Etalon ([69])	✗	✓	limited at scale	✓
StarryNet **(this paper)**		✓	✓	✓	✓

Table 1: Comparison of popular network experimentation platforms across different ENE requirements for ISTNs.

thoughts? The community requires a *technically* and *economically* feasible approach to construct ***Experimental Network Environments (ENE)*** and advance futuristic ISTN research.

3 How Can Researchers Evaluate Their New Thoughts for ISTNs?

3.1 ENE Requirements

Ideally, an ENE built for ISTN research is expected to simultaneously accomplish *acceptable* realism and flexibility. We summarize four baseline requirements as follows.

- **(1) Constellation-consistency.** The ENE is expected to be *spatially* and *temporally* consistent to the characteristics of real mega-constellations. For example, the ENE is desired to mimic a large number of network nodes at the same scale of a real mega-constellation, and can characterize the high dynamicity of LEO satellites, as well as its impact on network conditions (*e.g.,* connectivity, delay variations).
- **(2) System-level and networking stack realism.** The ENE is expected to run user-defined system codes and network functionalities like in a real system and networking stack.
- **(3) Flexible and scalable environment.** Emerging mega-constellations are evolving rapidly. As most state-of-the-art constellations are still not in their final stage, the ENE is expected to flexibly support various network topologies at different scales to meet diverse research requirements.
- **(4) Open, low-cost and easy-to-use interface.** Finally, it is expected that the ENE could be open to the community, and can provide low-cost and easy-to-use programmable interfaces for researchers to carry out various experiments.

3.2 Why Existing ENEs are Insufficient?

Existing approaches for building an ENE can be classified into three categories, differing in their realism, flexibility and cost: (1)live LSNs/platforms, (2)simulators, and (3)emulators.

Live LSNs or platforms. A straightforward approach for ISTN experimentation is to construct an ENE based on *live LSNs*, *e.g.,* recently SpaceX's Starlink has started its initial services in certain regions. Although this approach guarantees good realism, directly manipulating and inspecting a live LSN might be *technically* and *economically* difficult for a common research group. Current live LSNs are also limited in their *flexibility* when they face diverse, exploratory research requirements. Realistic mega-constellations are still under-constructed and evolving rapidly, and their regulatory information in practice cannot be flexibly modified for *what-if* analysis. Further, the network community has many public experimentation platforms [7, 20] that can be shared among researchers. However, these platforms are originally designed for tests in terrestrial networks, not for ISTNs, and thus cannot characterize the unique network behaviors under large-scale LEO dynamics.

Simulators and orbit analysis tools for ISTNs. Numerical or discrete-event-based simulation presents another extreme as compared with live LSNs and platforms. STK [35] and GMAT [11] are representative *orbit analysis tools* that can perform complex analysis of spacecrafts as well as ground stations. However, both STK and GMAT mainly focus on orbit and spacecraft analysis and have limited support for network simulation. More recently, SNS3 [76], Hypatia [60] and StarPerf [61] are emerging simulators for ISTNs. SNS3 is an extension to the ns-3 platform, and it models a full satellite network with a geostationary (GEO) satellite and bent-pipe payload. Hypatia is a framework for simulating and visualizing the network behavior of emerging mega-constellations. Similarly, StarPerf is a simulator that enables users to characterize, estimate and understand the achievable network performance under a variety of constellation options. Although the above simulators can *flexibly* simulate various satellite characteristics as well as the impact of high dynamics on network behaviors, a fundamental limitation of those simulators is that they can not support the run of system codes/functionalities and interactive network traffic as in real deployments. The abstraction-level of simulators might be too high to capture system-level effects, and could hide other practical issues (*e.g.,* software overhead under heavy workloads) in real systems.

Network emulators, and their variations. Emulation is a hybrid approach that integrates real applications, protocols

and operating systems in a synthetic network environment. Similar to live networks, emulators can load and run real codes with interactive network traffic. Similar to simulators, emulators can support controllable and diverse topologies and their virtual hardware requires fewer resources as compared with live networks. The community has many prior efforts focusing on emulated environments, *e.g.,* VM- or container-based emulation [6, 45, 54, 55, 68–70, 72, 79–81, 84, 85].

However, existing emulators suffer from two limitations when they are applied for generating ENEs for ISTN research. First, they are not constellation-consistent, since existing emulators inherently lack the ability of mimicking planet-wide LEO dynamics and time-varying network behaviors in ISTNs. Second, the network scale for mega-constellations could be significantly larger than that in prior experimentation. For example, authors in [54] use 10 physical machines (25 VMs on each) to support a networked cluster with 250 nodes. Etalon [69] is a container-based emulator and its local testbed uses four servers to emulate 48 hosts in a data center network. Different from prior scenarios, ISTN experiments require a much larger network environment: only the first shell of Starlink Phase-I includes about 1584 LEO satellites. Since both VMs and containers can involve software overhead on the physical host machine, it is difficult for existing emulators (*e.g.,* [54, 55, 68]) to support such large-scale and dynamic emulations for mega-constellations.

Our motivation. Table 1 summarizes the landscape of existing experimentation approaches that can be used to build ENEs. Collectively, we find that none of existing approaches can simultaneously satisfy the four expected features. Limitations of existing approaches thus motivate us to seek for a constellation-consistent, credible, flexible, and low-cost methodology to advance the test and evaluation of new research for futuristic ISTNs. We present such a framework, namely StarryNet, aiming at empowering researchers to build ENEs accomplishing the four goals as described in §3.1.

4 StarryNet Design

4.1 System Overview

Key idea. The design of StarryNet is inspired by an important insight obtained from the satellite Internet ecosystem: many organizations (*e.g.,* regulators and satellite operators) and end users have shared a collection of public data for the community, including constellation regulatory information [2, 73], orbital data observed from realistic satellites [16, 38], ground station distributions [3, 26, 39], and performance results measured from terrestrial user terminals [32, 33], *etc.* Based on this important fact, StarryNet creates ISTN experimental environments on demand by judiciously combining: (1) crowd-sourced real data trace; (2) model-based orbit and network analysis; and (3) large-scale network system emulation, to construct a real-data-driven digital twin, *i.e.,* a virtual presentation synchronized to a real

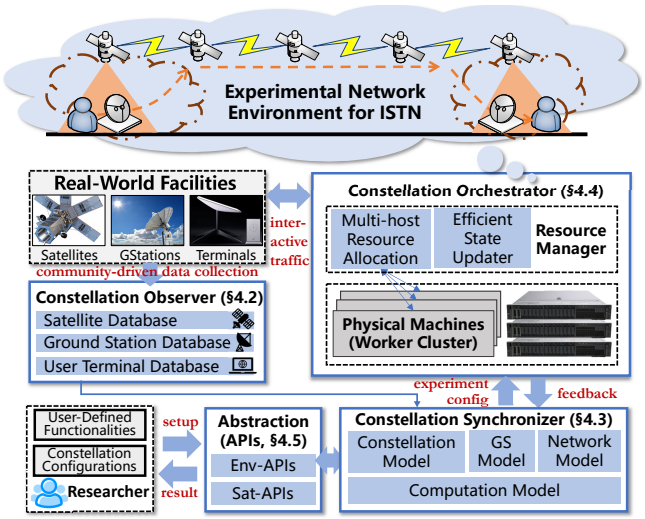


Figure 1: StarryNet system architecture.

physical ISTN in terrestrial environments. In particular, the key idea behind our StarryNet design can be summarized as follows: (1) leveraging a crowd-sourcing approach to collect, combine and explore realistic constellation-relevant information to calculate the *spatial* and *temporal* characteristics consistent to real mega-constellations; then (2) driven by such realistic information, exploiting a large number of networked *virtual nodes and links* to flexibly emulate a customized experimental environment, which characterizes system-level effects and network behaviors consistent to a real ISTN.

System architecture. Figure 1 depicts StarryNet's architecture, including four core components as described below.

- A ***Constellation Observer*** (§4.2) that leverages a *crowd-sourcing* approach to collect public constellation information, network performance, ground station distributions *etc.*, from the satellite ecosystem. It maintains several databases to support, guide and drive the construction of ISTN experimental environments for various research requirements.
- A ***Constellation Synchronizer*** (§4.3) which exploits a collection of hybrid models to calculate the *spatial and temporal characteristics* of a specific mega-constellation, based on collected data as well as user-defined configurations. Specifically, such characteristics include constellation scale, visibility, connectivity, time-varying propagation delay, *etc.*, which are further used to configure the network emulation.
- A ***Constellation Orchestrator*** (§4.4) for automating the management, coordination and allocation for resources used to build the experimental environment upon multiple physical machines. The orchestrator can also interact with real-world facilities (*e.g.,* real satellite hardware) to support network experiments with interactive Internet traffic.
- A ***Unified Abstraction*** (§4.5) offering flexible and easy-to-use *APIs* for researchers to create and manipulate the configurations of the ISTN experimental environment.

4.2 Constellation Observer

The constellation observer is designed as a collector for constellation-relevant information, and it maintains three databases related to satellite, ground station and user terminal information respectively. Specifically, the observer searches and collects the latest: (1) regulatory information (*e.g.,* from [2, 10, 73]) which describes frequency and orbital coordination of mega-constellations; (2) operating satellites information (*e.g.,* from [16, 38, 71]); (3) ground station distributions (*e.g.,* from [3, 26, 39]); (4) Internet user statistics (*e.g.,* from [59]) which can be used to generate the distribution of terrestrial users; and (5) network measurements from end users (*e.g.,* from [32, 33]). The constellation observer classifies the above information, and saves them in the databases, which then can be used to drive other components and build ENEs to flexibly support various research experiments.

4.3 Constellation Synchronizer

StarryNet's synchronizer leverages a collection of models to calculate various constellation characteristics and network behaviors, and accordingly generate a virtual network presentation synchronized to a real ISTN. At runtime, values in these models are assigned primarily based on real ISTN information collected by the constellation observer. In addition, to improve the flexibility, StarryNet also allows researchers to manually configure model values based on his or her customized experimental requirements.

4.3.1 Hybrid models

Constellation model. StarryNet's synchronizer characterizes a satellite system in both *constellation-granularity* and *orbit-granularity* descriptions. First, StarryNet uses the Walker notation [82] to describe a constellation: $N/P/p$, where N is the number of satellites per plane, P is the number of planes, and p is the number of distinct phases of planes to control spacing offsets in planes. Second, the orbit-granularity description enables a fine-grain notation for orbits in a certain constellation via specifying several primary orbital elements, including: (1) inclination, which is the angle between an orbit and the Equator as satellites move; (2) altitude, which is the height above sea level and determines the orbital velocity; (3) phase offset [56], which is a factor between $[0, 1]$, describing the relative position between two satellites in adjacent orbits.

Ground station model. StarryNet leverages the following parameters to describe a ground station: (1) geographic location; (2) the number of available antennas for ground communication; (3) the elevation angle, which determines the light-of-insight (LoS) of the ground station and can affect the available duration of space-ground communication.

Network model. The inter-satellite or ground-satellite network connectivity is mainly affected by following factors: (1) the visibility between two communication ends; (2) the amount of available ISLs or antennas in satellites or ground stations; and (3) the connectivity policy, which decides how to establish a connection between two communication ends. StarryNet enables two prefabs of connectivity policies for inter-satellite connection: (1) +Grid [58], where each satellite connects to two adjacent satellites in the same orbit and to two in adjacent orbits; (2) Motif [40], which is a repetitive pattern where each satellite connects to multiple visible satellites and each satellite's local view is the same as that of any other. For ground-satellite connectivity, StarryNet offers multiple optional schemes that control a ground station to connect to a visible satellite, *e.g.,* selecting the one with the shortest distance or the longest remaining visible time. Since the delay is typically determined by the network topology and speed of the light, StarryNet calculates the propagation delay of each link based on the physical distance between two ends. As network capacity might be too speculative in practice, link capacity is set by user-specific configurations.

Computation model. The computation capability of satellites varies greatly in different real-world deployments. Generally, conventional space-grade processors have limited capability [53, 65] as compared with that used in terrestrial computer systems. For example, the operating frequency of spaceborne processors (*e.g.,* BAE-RAD series [21, 22]) ranges from 110 to 466MHz per core. Recently, satellite operators and researchers start to use commercial off-the-shelf (COTS) processors in space stations [14] or in LEO small satellites [23, 44] to reduce the manufacturing cost. To support various experimental requirements, StarryNet allows researchers to manually configure the CPU capability of each satellite node through approximating the frequency and number of available cores of each emulated node, by scaling download CPU frequency and enforcing a maximum time quota for each node.

4.3.2 Constellation consistency

Spatial consistency. Based on constellation-wide information, StarryNet first determines the amount of required containers. Each container runs realistic system environments and networking stack to emulate a satellite in the constellation, a ground station or a terrestrial user terminal. Second, using orbit-granularity information, StarryNet calculates the latitude, longitude and height (*i.e.,* LLH information) of each satellite at any given time slot. Finally, exploiting the above LLH information and minimum elevation angles, StarryNet calculates the visibility between arbitrary two satellites, or between satellites and ground facilities.

Temporal consistency. Since emerging LEO satellites are inherently moving at a high velocity, the locations, intervisibility, and network conditions (*e.g.,* connectivity and propagation delay between two nodes) of an ISTN are changing over time. To achieve realism, these states should be temporally consistent to real scenarios. StarryNet splits time into slots, calculates LLH information in each slot, and follows the hybrid models to determine time-varying network conditions in different slots. Such dynamic states will further be used for driving the dynamic emulation operated by the orchestrator.

4.4 Constellation Orchestrator

The orchestrator is designed for four goals. First, the orchestrator exploits container-based emulation to construct an emulated ISTN environment upon one or many physical machines (depending on the constellation size). Second, the orchestrator configures the computation and network capability of each emulated node, according to users' configurations and the spatial and temporal results calculated by the constellation synchronizer. For example, a space-ground connectivity should be dynamically updated based on the time-varying visibility between its two ends. Third, the orchestrator can connect the emulated ISTN to real-world network facilities (if any, *e.g.,* real satellite prototypes or terminals) and support interactive Internet traffic. Finally, at runtime the orchestrator leverages a measurement agent to monitor and report the runtime resource usage (*e.g.,* CPU/memory/bandwidth usage) to the user as a feedback of the experiment for further analysis.

4.4.1 Multi-machine support for constellation emulation

Since each emulated satellite consumes computation, network and storage resources in practice, it is challenging to support the emulation of large-scale constellations on a single machine, especially when additional user-defined payloads (*e.g.,* a new satellite routing protocol) need to be loaded for experimentation. StarryNet addresses this limitation by integrating resources on multiple machines to support large-scale, time-varying constellation emulations. When deployed on multiple machines, StarryNet's orchestrator divides these machines into two parts. First, one machine, selected as the *resource manager*, manages, schedules and allocates resources upon all machines to jointly create and maintain the ENE. Second, other machines, working as *worker clusters*, receive and follow commands from the resource manager. For each node in the ISTN (*e.g.,* a satellite or ground station), the orchestrator creates a container to emulate it, and creates a virtual bridge connecting two nodes to emulate a link.

Topology creation on multiple machines. One big challenge made by extending an emulated mega-constellation to multiple machines is to *ensure that the emulated constellation is topologically consistent to the real constellation*. Figure 2 shows an example of emulating an LEO satellite constellation including two adjacent orbits on two physical machines. Specifically, Figure 2a plots two Starlink inclined orbits, each of them having 22 satellites evenly spaced with available ISLs. Assume that we use two machines for this emulation, and each machine creates 22 containers to emulate 22 satellites. In a real constellation, each satellite connects its two neighbors in the same orbit, and to another satellite in the adjacent orbit. Ideally, an emulated topology for the constellation in Figure 2a can be easily created if each machine has more than 22 physical network interfaces. However, in practice the number of available interfaces of a machine is likely to be limited. As shown in Figure 2b, if we directly bridge the emulated network interface of each satellite to the physical interface, due to the lack of traffic isolation, emulated satellites in two machines will establish an "all-to-all" topology which is inconsistent to the grid-like topology in Figure 2a .

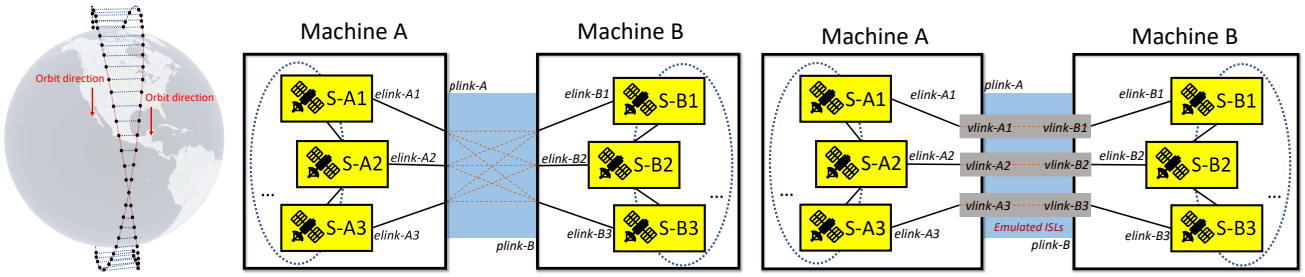
StarryNet addresses this inconsistency on multiple machines by creating multiple VLANs to emulate independent ISLs, interconnect satellites in two machines and isolate inter-satellite traffic as that in a real constellation. As plotted in Figure 2c, we build a VLAN for each ISL crossing different machines (*e.g.,* vlink A1-B1 interconnects emulated satellite A1 and B1), and thus StarryNet obtains the correct virtual network topology consistent to the real constellation topology as illustrated in Figure 2a upon multiple machines.

Topology update on multiple machines. Due to the high dynamics of ISTN, the network topology fluctuates over time. If a connectivity change occurs on a single machine, *i.e.,* all affected nodes are located on the same machine, StarryNet deletes the old virtual link, and creates a new link connecting corresponding nodes. Otherwise, if a connectivity change involves nodes on multiple machines, StarryNet exploits a VLAN-based approach to limit the link update operation to a single machine. Figure 3 plots an example of the emulation of space-ground handovers upon multiple machines. Assume there are three satellites, two ground stations in two sequential time slots. In the first slot, satellite S2/S3 connects to ground station GS1/GS2 respectively. As satellites move, the space-ground connectivity changes, and S1/S2 connects to GS1/GS2 respectively in the second slot. StarryNet emulates the ground-satellite links (GSLs) by two VLAN-based virtual links, and performs the correct handover by properly adjusting the connectivity change between S1/S2/S3 and vlink-GS1/GS2, in different slots on machine A.

4.4.2 Efficient time synchronization and state update

Another challenge made by multi-machine extension is the *time synchronization and link update overhead.* Specifically, to achieve temporal consistency, StarryNet's constellation synchronizer assigns a sequence of update events to the orchestrator to inform each emulated satellite *when* an update (*e.g.,* a location/connectivity change) should happen. To trigger such events, a baseline approach is to let the centralized manager send commands to all emulated satellites in every slot, and trigger corresponding update events. However, such a real-time approach has limited scalability as the amount of emulated satellites increases, since each event requires to update the state of a certain virtual interface/container, and continuously and simultaneously performing a large number of updates can overload the manager, result in delayed update, and invalidate the temporal consistency of the emulation.

To reduce the state update overhead caused by mimicking temporal dynamics in mega-constellations, StarryNet leverages a prediction-based *multi-thread event memorization* approach. We define the synodic period as the amount of time that it takes for the constellation to reappear at the same projection upon the earth surface. In the emulated environment,



(a) Two orbits with ISLs. (b) Emulated topology without traffic isolation. (c) Emulated topology with emulated ISLs for traffic isolation.

Figure 2: Multi-machine extension to support consistent topology emulation of satellite networks. Machine A and B interconnect by their physical interfaces plink-A/B (plink: physical link, elink: emulated link, vlink: virtual network link created by `VLAN`).

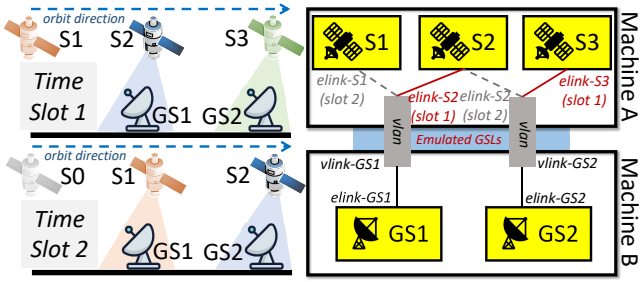


Figure 3: Emulated ground satellite links (GSLs) and space-ground handover in two time slots on multiple machines.

at the beginning of each *synodic period*, the orchestrator pre-generates an event list for each satellite that includes all its update events during the current synodic period. At runtime, each emulated satellite adopts an independent *event update thread* to read the local event list in each slot, and triggers the corresponding event scheduled in the current slot. Time clocks upon all machines are synchronized by the NTP [17].

4.5 Open APIs for ISTN Experiments

Environment APIs. StarryNet provides the environment APIs for a researcher to load trace from the database, and create/control/run an ENE upon one or multiple machines with user-defined ISTN configurations. Once constellation and GS information are completely loaded, these configurations are delivered to the synchronizer to calculate spatial and temporal characteristics, which are further used by the orchestrator to construct the ENE. The environment APIs also allow users to configure the interval of discrete time slot to adjust the dynamicity. Note that the ENE not only maintains the runtime of emulated constellations, but also needs to run the test workload deployed by the researcher. We design a resource threshold Δ to control the percentage of CPU/memory used by the framework. In other words, at least (100%-Δ) of the total CPU/memory should be left for the researcher's test cases.

Self-node APIs. StarryNet's self-node APIs are designed to be called by the researcher's programs on each emulated satellite. These APIs expose underlying satellite-related information to user programs. Specifically, in each emulated satellite, user program can obtain the index of current satellite/orbit, sunlight state, time-varying geo-location information, current satellite velocity and the index of adjacent reachable satellites, *etc.* Such satellite-specific information can be used for developing new on-board capabilities in ISTNs.

5 Implementation and Usage

We highlight the salient aspects of StarryNet's implementation and usage in this section.

Framework implementation. StarryNet's observer is implemented as a combination of a crawler based on `Scrapy` [27], together with a MySQL-based data store. We implement StarryNet's synchronizor based on `SkyField` [29], an astronomy library that supports the calculation of high precision research-grade positions for satellites. StarryNet's orchestrator is implemented upon `Docker` [6] and it spans the emulated constellation across multiple machines. We use `OpenvSwitch` [19] to emulate and configure links, and use `tc` [67] to dynamically set artificial network conditions according to the numeric results calculated by StarryNet's synchronizer. Specifically, we optimized the link management module in `tc` to satisfy the requirement of light-weight state update. To accomplish flexibility, StarryNet's abstraction is implemented as a combination of a `lib`-StarryNet library and a collection of shell commands. Collectively, the core components of StarryNet are implemented in about 6500 lines of Python codes and scripts.

Framework usage. We illustrate the usage of StarryNet with a concrete example as plotted in Figure 4: a researcher wants to evaluate a novel geo-location-based routing mechanism based on [63], under the Starlink constellation. In particular, this experiment can be conducted with StarryNet in three steps. First, leveraging StarryNet's APIs, the researcher writes a user-defined experimental program (Figure 4a) for test. In this example, we show a geo-routing policy similar to [63], which runs on each satellite, and forwards received packets to the adjacent satellite that is the geographically closest to the destination. Second, the researcher prepares a set of manifest files describing the constellation configurations, *e.g.,* orbital information and ground station distribution (Figure 4b). Finally, the researcher runs a batch of shell commands exposed by StarryNet to load manifest files (*e.g.,* starlink.json and gs.json), create experimental en-

```python
# geo_routing.py
from lib_starrynet import *;
def geocast_next_hop(dst_addr):
   # Obtain adjacent satellites info
   n_sats = sn_get_sat_neighbors()
   # Find the sat closest to dst
   for sat in n_sats:
      if dis(sat, dst_addr)
         < dis(next_sat, dst_addr):
         next_sat = sat
   return next_sat
```

(a) User-defined experimental program.

```
"starlink":[ #starlink.json
  {  "name": "Starlink-S-I",
     "altitude": "550km",
     "inclination": "53.0",
     "plane_count": "72",
     "satellites_per": "22" }]
"ground-station":[ #gs.json
{ "name": "Chicago",
  "latitude": "41.850",
  "longitude": "-87.650",
  "altitude": "0.144km"},...]
```

(b) Constellation configurations.

```
#(1)initialize sn and monitor an interface of the manager machine
@manager:/$ sn manager init --m-addr=192.168.0.1
#(2)connect each worker machine to the manager to join the framework
@workers:/$ sn worker join --m-addr=192.168.0.1
#(3)load manifest files and create the ENE named "sl_cons"
@manager:/$ sn create --name sl_cons -c 'starlink.json' -gs 'gs.json'
#(4)manually set uplink/downlink capacity to 20Mbps/200Mbps
@manager:/$ sn network --name sl_cons --gsl-up=20 --gsl-down=200
#(5)start the ENE, and run it for 3600 seconds
@manager:/$ sn start sl_cons --duration=3600 --delta=0.5
#(6)run user-specific program in all satellites in the first orbit
@manager:/$ sn cmd sl_cons.orbit[0] python /home/geo_routing.py
```

(c) Load configurations to initialize the ENE and run experiment.

Figure 4: A getting-started example of StarryNet (`sn`).

vironment (*e.g.,* "sl_con") on multiple machines, configure network parameters (*e.g.,* uplink/downlink capacity), and run the user-specific program on emulated satellites (Figure 4c).

6 Framework Evaluation

In this section, we evaluate StarryNet by exploring two important aspects related to the framework. **Q(i):** Can StarryNet flexibly scale to various experimental requirements, with acceptable system and configuration overhead? **Q(ii):** How faithful are the results obtained by StarryNet, as compared with other state-of-the-art simulators, and live network performance? Our framework evaluations are conducted on a typical enterprise cluster, including eight DELL R740 servers connected to a LAN. Each server is equipped with two Intel Xeon 5222 Processors (4-core, 3.8GHz for each processor), 8*32G DDR4 RAM, and Ubuntu20.04-LTS.

6.1 Ability to Satisfy Various Experimental Requirements for ISTNs

Elastic scaling to various constellation configurations. In reality, satellite operators *incrementally* deploy their satellite mega-constellations, which consist of multiple *shells*. As depicted in Table 2, StarryNet is able to flexibly create a user-defined experiment environment for different shells, or multi-shell combinations of representative mega-constellations to satisfy various research requirements. The emulated constellation size can scale from about 300 (*e.g.,* the T1 shell of Telesat) to 4408 (*e.g.,* the full-scale Starlink Phase I with five shells) following different users' configurations.

Environment setup overhead. StarryNet's APIs have concealed complex underlying processing for trajectory calculation and resource orchestration for the emulation. Thus a researcher can easily establish each ENE listed in Table 2, by writing about a dozen lines of code based on constellation prefabs (*e.g.,* like Figure 4b) predefined in StarryNet's database. The creation time of a certain ENE upon StarryNet tightly depends on the experiment scale, and the hardware capability of these machines used for experiments. Concretely, as shown in Table 2, the total creation time, including both node and link creations, increases as the constellation size scales up, and ranges from several minutes (for small size ENE) to tens of minutes (for large size ENE) in our current StarryNet implementation.

System overhead. Table 2 also plots the average CPU and memory overhead consumed on each machine by running various ENEs. We make several observations. First, as expected, when the constellation size increases, StarryNet requires more worker machines, consuming more CPU/memory resources to emulate ISTN nodes, links, and their constellation-wide dynamics. Second, if StarryNet updates satellite dynamics more frequently (*i.e.,* with shorter update intervals), it consumes more resources to accomplish fine-granularity updates. Note that in this experiment we limit the CPU usage below $\Delta = 50\%$ in each machine. This is because in an ENE, the runtime overhead of the underlying StarryNet should not use up all CPU/memory resources. It is reasonable to leave sufficient resources for the tested workloads and functionalities running upon the ENE.

6.2 Fidelity Analysis

Next we analyze the fidelity of StarryNet by comparing the experiment results obtained by StarryNet with live satellite networks and other state-of-the-art simulators.

Network performance under a live Starlink topology. We leverage StarryNet to establish an ENE following the network topology of a recent live Starlink test conducted in Europe in 2021 [33]. Specifically, this real-world Starlink topology involves several key components as illustrated in Figure 5: (1) a user terminal together with a Starlink satellite dish located at the campus Klagenfurt Primoschgasse; (2) a SpaceX's ground station located in Frankfurt, Germany; (3) a Point of Presence (PoP) connecting the ground station to terrestrial Internet; and (4) a Web server deployed in Vienna. This experiment publicly reports the `ping` and `iperf` results measured between user terminal and the Web server, over the ISTN integrating Starlink satellites and terrestrial Internet. We use StarryNet, Hypatia [60] and StarPerf [61] to generate network performance under the same topology configuration. The latter two are state-of-the-art ISTN simulators. Figure 6 plots the comparison for the latency results. First, we find that existing simulators *underestimate* the latency, since their latency estimations are based on a high-level abstraction without considering system effects like packet processing overhead. Second, StarryNet achieves acceptable fidelity, as it attains similar latency performance in each case (*i.e.,* aver-

Constellation \ Metrics	Height (km)	Constellation Size (number of satellites)	Creation Time (min) Nodes/Links/Total			Avg. CPU (%) Interval = 1/2/3 (s)			Avg. Memory (%) Interval = 1/2/3 (s)			Minimum # of Required Workers
Starlink S1 (72*22, 53°)	550	1584	5.9	4.6	10.5	7.2%	7.0%	6.3%	3.9%	3.5%	3.4%	2
Starlink S2 (72*22, 53.2°)	540	1584	5.9	4.6	10.5	7.2%	7.0%	6.3%	3.9%	3.5%	3.4%	2
Starlink S3 (36*20, 70°)	570	720	3.0	2.1	4.9	1.2%	1.1%	1.0%	2.7%	2.6%	2.6%	1
Starlink S4 (6*58, 97.6°)	560	348	1.9	1.3	3.2	1.0%	1.0%	1.0%	2.7%	2.6%	2.4%	1
Starlink S5 (4*43, 97.6°)	560	172	1.6	1.2	3.2	1.0%	1.0%	1.0%	2.3%	2.3%	2.3%	1
Starlink Full (4408 satellites)	hybrid	4408	13.3	7.9	21.2	39.6%	37.0%	34.3%	10.4%	9.1%	8.9%	7
Kuiper K1 (34*34, 51.9°)	630	1156	4.4	3.8	8.2	2.6%	2.4%	2.3%	3.8%	3.5%	3.2%	2
Kuiper K2 (36*36, 42°)	610	1296	4.7	4.2	8.9	3.9%	3.6%	3.2%	4.0%	3.6%	3.5%	2
Kuiper K3 (28*28, 33°)	590	784	3.2	2.4	5.6	1.3%	1.2%	1.2%	2.7%	2.6%	2.6%	2
Kuiper Full (3236 satellites)	hybrid	3236	5.7	4.8	10.5	24.6%	23.9%	23.2%	6.3%	6.2%	6.2%	6
Telesat T1 (27*13, 98.98°)	1015	351	1.9	1.3	3.2	1.0%	1.0%	1.0%	2.6%	2.5%	2.4%	1
Telesat T2 (40*33, 50.88°)	1325	1320	4.8	4.2	9.0	3.9%	3.7%	3.3%	4.0%	3.6%	3.5%	2
Telesat Full (1671 satellites)	hybrid	1671	3.1	2.4	5.5	7.2%	7.0%	6.4%	4.2%	3.7%	3.6%	3

Table 2: Ability to support mega-constellation emulation with various experimental configurations and system overheads.

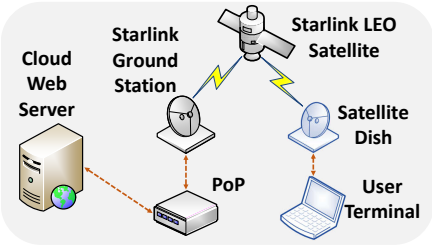


Figure 5: A live Starlink topology.

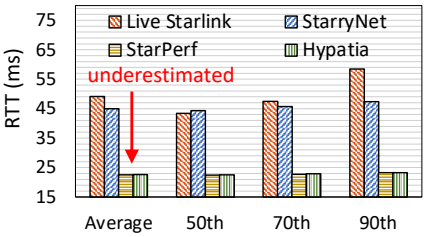


Figure 6: Latency comparison.

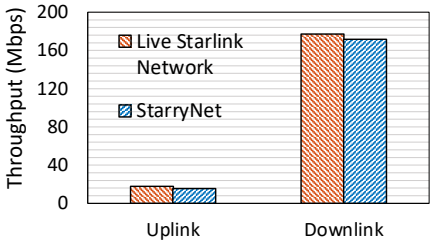


Figure 7: Throughput comparison.

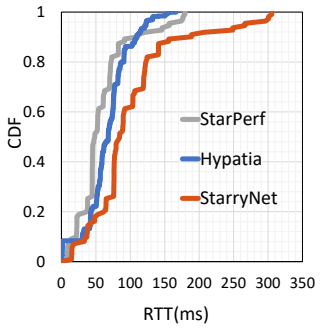


Figure 8: Latency comparison in an ISL-enabled topology.

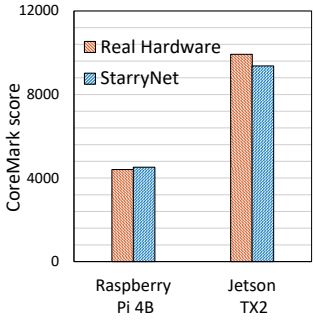


Figure 9: Flexible computation capability.

age/50th/70th/90th percentile) as compared with the real measured data from live Starlink. This is because StarryNet jointly combines model calculation, data-driven calibration and real networking stack to create the ENE.

Bandwidth is a metric that can be affected by many operational factors. Therefore, in a research experiment StarryNet allows the researcher to manually configure the link capacity on demand. For example, we follow the realistic Starlink trace in [33] to set the uplink/downlink capacity, and run `iPerf` to measure the TCP throughput in each direction. Since Hypatia and StarPerf can not load real network traffic by `iPerf`, we compare the throughput results of live Starlink and StarryNet. Specifically, evaluation results in Figure 7 demonstrate that StarryNet can be tuned to accurately emulate the bandwidth of a live ISTN.

Network performance under an ISL-enabled topology. As of the date of this paper submission, most real mega-constellations like Starlink and Kuiper are still in their early stage and under heavy construction. Although Starlink has started to deploy laser ISLs on its LEO satellites, those ISLs are still under internal test, and it is difficult to directly compare the network performance estimated by StarryNet with a real ISL-enabled satellite network. To analyze the fidelity of StarryNet when ISLs are activated, we compare the performance results obtained by StarryNet with other ISTN simulators. Figure 8 plots the CDF of latency between a collection of real ground-station pairs [26] with the same constellation configuration based on the ISL-enabled Starlink network. The latency results of StarryNet are measured by `ping` test in the emulated ENE, while the results of other simulators are generated by numeric or event-driven calculation. As shown in Figure 8 the latency obtained by StarryNet is slightly higher than other simulators, because StarryNet incorporates realistic system-level overhead (*e.g.,* packet processing) which could be neglected in simulators.

On-demand computation capability. Researchers may need to conduct their experiments on different satellite hardware with various computation capabilities. For example, authors in [53] studied the application performance achieved by two space-grade processors RAD-5545 [21] and HPSC [13]. Recent works like [23, 44] explored new satellite functionalities running upon commercial low-power processor such as Raspberry Pi [24] and Jetson TX2 [18]. StarryNet is able to flexibly adjusting the computation capability on each emulated satellite to satisfy various experimental requirements. To validate the computational flexibility, we use `CoreMark` [5],

a well-known processor benchmark to measure the performance of the real hardware and its facsimile created by StarryNet. As plotted in Figure 9, CoreMark score is a metric that quantifies the computation capability. Higher scores indicate stronger computing capability. For various computation requirements, StarryNet can mimic similar processor capability based on concrete experimental requirement.

7 Evaluating Futuristic ISTN Research with StarryNet: Case Studies

Next we conduct several case studies to show how StarryNet can be used to advance futuristic ISTN research.

7.1 Exploring the Design Space of Integrating LEO Satellites and Terrestrial Facilities

To realize the promise of low-latency and pervasive accessibility of ISTN, the first step should be interconnecting LEO satellites and terrestrial facilities (*e.g.,* ground stations and user terminals). While many existing studies have proposed different space-ground topology designs, it still lacks a systematically analysis and comparison on these integration paradigms, in terms of their network performance and corresponding cost. We leverage StarryNet to explore how different design choices of space-ground integration (as illustrated in Figure 10) could affect the performance and cost of an ISTN.

(1) Satellite relays for last-mile accessibility (SRLA). Satellites and ground facilities can be integrated based on the classic "bent-pipe" architecture to provide last-mile network accessibility as illustrated in Figure 10a, which is the *status-quo* of many today's satellite constellations like OneWeb. Data from the ground are first transmitted to the satellite, which then sends it right back down again like a bent pipe. The only processing performed by satellites is to retransmit the signals.

(2) Satellite relays for ground station networks (SRGS). Figure 10b depicts another "bent-pipe"-based integration paradigm originally introduced in [57], where geo-distributed ground stations work as routers to construct a network. Each satellite switches packets between two ground stations connected to the satellite. Packets from the sender are routed to the receiver by paths built upon satellites and ground stations.

(3) Ground station gateway for satellite networks (GSSN). Figure 10c shows an ISL-based internetworking approach proposed by [52]. ISL-enabled satellites can build space routes for long-haul communication, without the need of a large number of ground station relays, as well as user-side satellite dishes. Satellites and ground stations build a Layer-3 network. During an end-to-end transmission, packets from the sender are first routed to a ground station via terrestrial Internet, then to the receiver side ground station over ISL-enabled satellites, and finally to the receiver over the terrestrial Internet.

(4) Directly accessed satellite networks (DASN). Figure 10d plots a paradigm where users' satellite dishes directly connect to ISL-enabled satellite networks, and two users can establish long-haul communication without the assistance of geo-distributed ground stations [51, 56]. Satellites work not only as routers, but also as access points/gateways allocating addresses for different terrestrial users.

Experiment setup. We leverage StarryNet to build an ENE for each paradigm, analyze their cost, and evaluate their network performance. Specifically, we establish an ENE based on Starlink's first constellation shell and its ground station distribution. We randomly pick geo-distributed user-pairs and establish communication sessions between these pairs over the satellite network. On each emulated satellite, we load `BIRD` [37] routing software and run `OSPF` as the routing protocol to achieve the shortest path for data transmission.

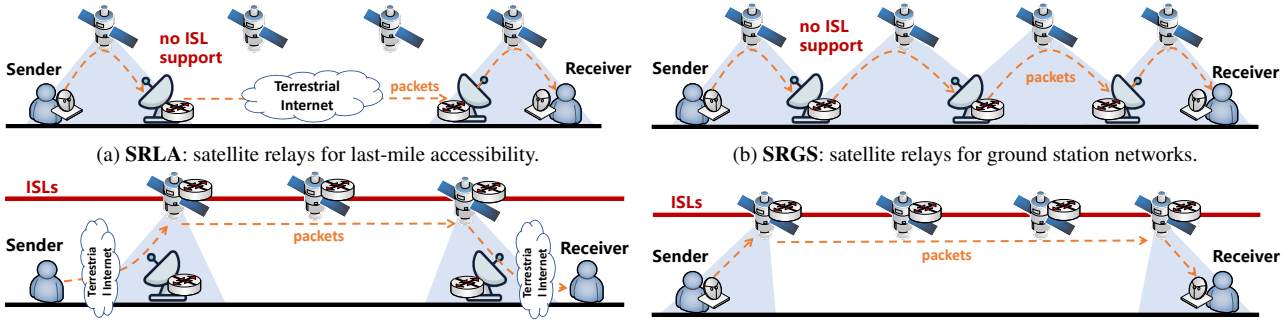
Observations. Table 3 summarizes the average end-to-end latency and the latency breakdown of different space-ground topology designs. We observe an obvious latency reduction accomplished by laser ISLs, and DASN obtains the lowest end-to-end latency on average. Since ground stations typically can not be deployed upon oceans (70% earth surface), SRGS suffers from the highest latency as compared with other schemes due to the insufficient deployment of ground stations.

As satellites move, two main factors affect the end-to-end network reachability. First, users in certain regions may lose available satellite access due to the LEO dynamics. For example, users in high latitude areas may suffer from intermittent satellite access. Second, frequent connectivity changes can trigger *routing re-convergence*. As plotted in Table 3, SRGS suffers from the lowest reachability due to the combination of LEO dynamics and limited coverage of insufficient ground stations. GSSN obtains low reachability because frequent satellite-ground handovers result in continuous re-convergence, during which routes are fluctuating and unstable.

For SRLA, SRGS and GSSN, the IP address of user's satellite dish is allocated by the fixed user-side ground station, and the addresses of senders or receivers do not change during the communication. However, for DASN, each satellite works not only as a router, but also as an access point/a gateway which allocates IP addresses for terrestrial dishes connect to it. Due to the LEO dynamics, terrestrial dishes have to frequently change access satellite as well as their subnet. Consequently, addresses are frequently updated, which may further disrupt high layer transport connections and application sessions.

The above four topologies have different deployment and operating costs in addition to LEO satellites. SRLA and SRGS require a large number of available ground stations near users to guarantee continuous satellite coverage. For SRGS, it also requires sufficient geo-distributed ground stations to ensure stable and low-latency routes over satellites and ground stations. GSSN and DASN require the extra deployment of ISLs. Users in SRLA, SRGS and DASN have to purchase a dedicated dish to access satellites. In GSSN, users connect to the ground station gateway via terrestrial networks, and do not need to install additional satellite dishes.

Implications. As summarized in Table 3, there is no clear win-



(a) **SRLA**: satellite relays for last-mile accessibility.

(b) **SRGS**: satellite relays for ground station networks.

(c) **GSSN**: ground station access for satellite networks.

(d) **DASN**: satellite networks directly accessed by terrestrial users.

Figure 10: The design-space for various space-ground integration methodologies.

Design	Average end-to-end latency and its breakdown (ms)				Reachability	Frequent Address Update	Operating Cost		
	Inter-Satellite	Space-Ground	Ground	Toal			GS	Terminal	ISLs
SRLA	0	76.25	107	183.25	97.00%	✗	✓	✓	✗
SRGS	0	313.39	0	313.39	51.00%	✗	✓	✓	✗
GSSN	48.46	38.45	20	106.91	57.40%	✗	✓	✗	✓
DASN	48.46	37.65	0	86.11	97.50%	✓	✗	✓	✓

Table 3: Comparison for different space-ground integration methodologies.

ner for all four integration methodologies. "Bent-pipe"-based approaches achieve simplicity without ISL requirements, but they fail to fully unleash the low-latency potential of ISTNs. Approaches relying on ISLs can form near-optimal spaces routes to attain wide-area low-latency communication, but they involve extra ISL cost, and suffer from higher route instability and connection disruptions, due to the route re-convergence and address update caused by LEO dynamics. All integration approaches have their limitations, and satellite operators are suggested to choose a proper topology based on their specific performance requirements and cost budgets.

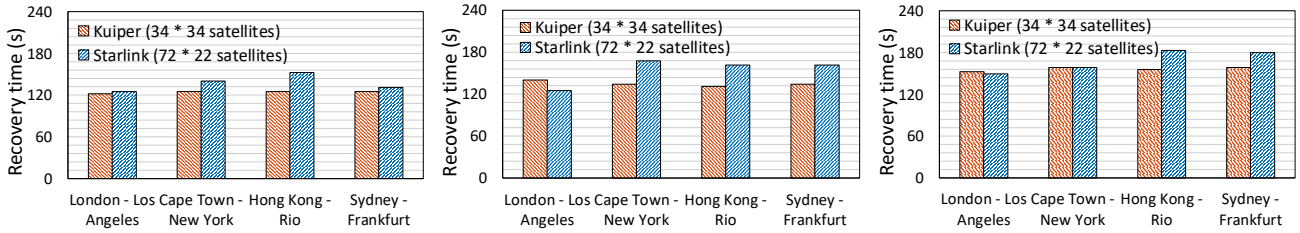
7.2 Evaluating ISTN Resilience

Satellites are operated in complex outer space environments. Small satellites deployed in emerging mega-constellations typically have a short lifetime (*e.g.,* 3-5 years [30]) due to their low manufacturing cost. Many space factors or events, such as space debris [15], geomagnetic storms [12], and single event upset [28], *etc.*, can cause immediate satellite failures. For example, in February 8, 2022, about 40 Starlink satellites are doomed by a geomagnetic storm [1]. Therefore, given the harsh and error-prone space environment, it should be important for satellite operators and researchers to evaluate and analyze how resilient an ISTN is, and what kind of system/network factors affect the resilience.

Experiment setup. We thus conduct an experiment with StarryNet to evaluate the network resilience with different routing configurations. Specifically, we mimic the impact of a space failure (*e.g.,* due to a geomagnetic storm) which makes a fraction of satellites in the constellation inactive and can not forward network traffic. We load `BIRD` [37] in our ENE and run `OSPF` as the routing protocol in this experiment.

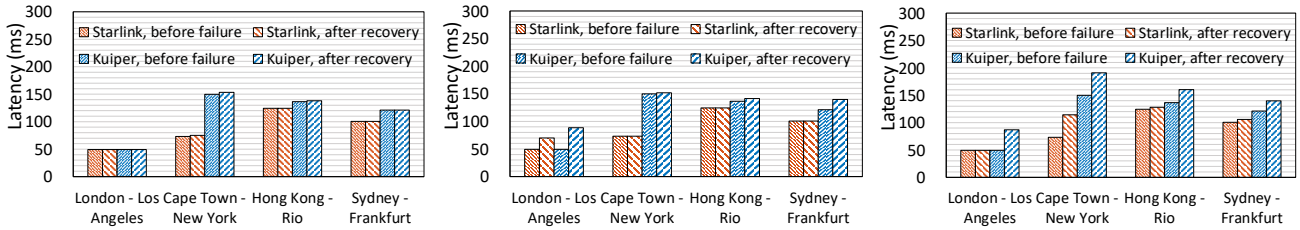
Observations. Figure 11 plots the routing recovery time for a set of representative city-pairs under various failure ratios. An on-path satellite failure can cause a network disruption, and the routing recovery time increases as the constellation size and failure rate increase. Figure 12 plots the comparison for the end-to-end latency before the constellation failure and after the routing re-convergence. We observe that the latency increases slightly under low failure rate, and the latency dramatically increases when the failure rate reaches 30%.

Implications. We summarize three key implications from this experiment. First, we find the mesh-like network based on a large number of satellites can maintain low latency in case of low failure rate. This is because the mesh-like satellite network has high *path diversity*, offering backup routes for communication pairs in case of network failures. Second, the inherent high dynamicity of LEO satellites is a double-edge sword for the service restoration in an ISTN. On one hand, for terrestrial users whose access satellite above them fails, the dynamicity helps because faulty satellites will soon move out of their line-of-sight. On the other hand, the dynamicity hurts, as it spreads the failure globally, and could affect the network accessibility of other users. Finally, while improving redundancy in physical connectivity and applying robust mechanisms in protocol design are two critical directions to improve the ISTN resilience, it is challenging to attain a "win-win" integration of them in practical systems. Increasing the satellite density indeed improves the resilience of satellite accessibility in case of sudden failures, but it also involves much more nodes and links in the network, and thus imposes new challenges and requirements on the protocol scalability and recovery efficiency under various failure scenarios.



(a) Route recovery time under 10% failure ratio. (b) Route recovery time under 20% failure ratio. (c) Route recovery time under 30% failure ratio.

Figure 11: Route recovery time under different constellation-wide failures.



(a) Increased latency under 10% failure ratio. (b) Increased latency under 20% failure ratio. (c) Increased latency after 30% failure ratio.

Figure 12: Increased latency after route reconvergence under various constellation-wide network failures.

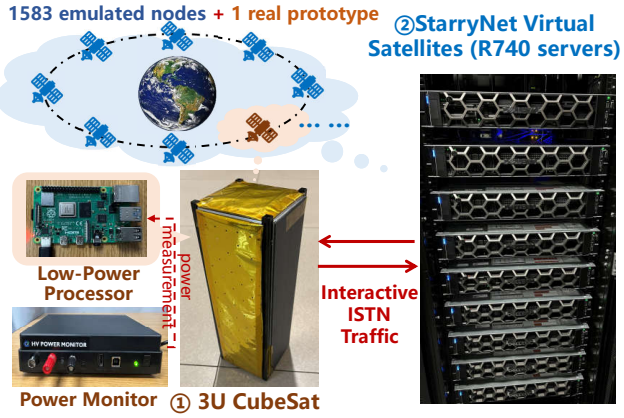


Figure 13: Hardware-in-the-loop testing with StarryNet.

7.3 Hardware-in-the-loop Testing

In real satellite deployments, it is very important to accurately estimate how much energy a new system or network function will consume before the launch. StarryNet enables researchers to conduct *hardware-in-the-loop* testing to accurately evaluate the low-level system effects under various workloads. As a case study to demonstrate StarryNet's ability, we connect a 3U CubeSat prototype, equipped with a low-power processor [23,24] running real routing protocols to the virtual satellite network emulated by StarryNet, as illustrated in Figure 13. Collectively, the 3U CubeSat prototype together with the emulation creates a virtual constellation network with 1584 Starlink satellites. We follow the satellite traffic model proposed in [51] to inject traffic and use a power monitor to measure the satellite prototype. Table 4 summarizes the power consumption in different states (*e.g.,* calculating route convergence and forwarding traffic in various data rates). Our hardware-in-the-loop test demonstrates

State	Idle	Routing convergence	Data transmission rate (Mbps)				
			100	250	500	750	1000
Power consumption (W)	2.83	3.22	4.6	4.99	5.36	5.45	5.46

Table 4: Power consumption under various ISTN workloads.

StarryNet's ability to create a hybrid ENE and evaluate real system effects for user-defined functionalities.

8 Limitation and Future Work

Experimental scope and limitations. Our StarryNet framework mainly targets at various network-level experiments for ISTNs, *e.g.,* evaluating a new routing/transport-layer protocol, or assessing the network performance of a new topology design in a highly-dynamic, resource constrained virtual ISTN environment. The scale of the experiment supported by StarryNet is closely related to the underlying resources provided by physical machines. In its present form, the key limitation of StarryNet can be summarized as follows. First, StarryNet is essentially a data-driven framework combining constellation-relevant modeling and network emulation. Thus its fidelity tightly depends on the availability and accuracy of the public information shared by the satellite ecosystem. For example, in practice, public TLE data may provide inaccurate orbit information, which can have errors up to 12 km, and such errors can affect the calculation of network performance (*e.g.,* inter-satellite visibility and propagation delay). Second, some parameters are hard to obtain from a practical satellite system today, because most mega-constellations are still in their early stage with limited access. For example, it is difficult to obtain the real ISL-enabled Starlink performance right now, since Starlink's laser ISLs are still under internal test. Thus, StarryNet allows researchers

to manually configure the ISL parameters (*e.g.,* link capacity) and customize their experiments based on various experimental requirements. Third, our framework is primarily based on virtualization-based network-level emulation, and thus it has limited ability to emulate physical layer (PHY) characteristics that can be observed in a live network experiment, *e.g.,* spectrum adaptation and multiplexing [42], or the time consumed by a real satellite dish to detect PHY connectivity changes.

Future work. Satellite Internet mega-constellations are still evolving rapidly. New constellation designs are constantly being proposed, and existing constellation schemes are constantly being updated. In our future work, we will follow the evolution and deployment of realistic satellite Internet constellations. In particular, we will track the latest constellation information to update StarryNet's open database, calibrate the constellation models and further improve the fidelity of the StarryNet framework. Moreover, based on these implications obtained from our case studies in §7, we will explore new network techniques tailored for ISTNs, *e.g.,* practical and resilient satellite routing protocols in the future. Our latest research progress on StarryNet will be updated on the website: https://github.com/SpaceNetLab/StarryNet.

9 Related Work

Section 3.2 has discussed most existing efforts relevant to the method of building ENEs. In this section we briefly introduce other ISTN works related to our study in this paper.

The network community has many recent efforts studying on the topology design [40, 58], routing [47, 52, 56, 57, 83], transport-layer congestion control [60, 64], new satellite applications [62] and security issues [51] for emerging ISTNs. For instance, Motif [40] is a recent topology design for LSNs, in which each satellite is dynamically connected to other visible satellites to achieve low latency under various traffic configurations. Works in [40, 56, 57] suggest the use of pre-calculated shortest-path-based routing and traffic engineering schemes for ISTNs. On one hand, these pioneering studies indeed outline the promising network potential of futuristic ISTNs. On the other hand, the above new thoughts are evaluated by simulations with a high-level abstraction. StarryNet can stimulate new research and advance existing ISTN works by evaluating them in a more realistic ENE to obtain practical insights for further optimizations.

The rapid evolution of ISTNs also attracted the attention of the system community. Specifically, orbital edge computing (OEC) [43, 44, 66] is a new computation architecture which leverages computational satellites to pre-process earth observation (EO) data, and save the data download overhead. Studies in [49, 50] explored the feasibility of applying deep neural networks to process on-board satellite data. Since StarryNet creates real system runtime and networking stack in an experimental ISTN environment, it can also help to evaluate system-level effects of these new algorithms, implementations and programming models designed for ISTNs.

10 Conclusion

To advance futuristic research on ISTNs, this paper presents StarryNet, an experimentation framework that empowers researchers to conventionally and flexibly build ENEs for ISTN research. StarryNet simultaneously achieves constellation-consistency, network realism, and flexibility, by integrating real constellation-relevant information, orbit analysis and large-scale emulations to construct ENEs. By comparing StarryNet's results with live network performance and conducting diverse case studies, we demonstrate StarryNet's fidelity and flexibility for various ISTN experiments. We are confident that the open-source StarryNet can help the network and system community to flexibly conduct various ISTN evaluations with more credible results.

Acknowledgments

We thank our shepherd Behnaz Arzani and all anonymous NSDI reviewers for their feedback which greatly improved the paper. This work was partially supported by the National Natural Science Foundation of China (NSFC No. 62132004) and Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute.

References

[1] 40 Starlink satellites doomed by geomagnetic storm. https://earthsky.org/space/40-starlink-satellites-doomed-by-geomagnetic-storm/.

[2] Application of Kuiper Systems LLC for Authority to Launch and Operate a Non-Geostationary Satellite Orbit System in Ka-band Frequencies. https://licensing.fcc.gov/myibfs/download.do?attachment_key=1773885.

[3] Azure Orbital: Satellite ground station and scheduling service connected to Azure for fast downlinking of data. https://azure.microsoft.com/en-us/services/orbital/.

[4] China's megaconstellation project establishes satellite cluster in chongqing. https://spacenews.com/chinas-megaconstellation-project-establishes-satellite-cluster-in-chongqing/.

[5] Coremark: a benchmark designed specifically to test the functionality of a processor core. https://www.eembc.org/coremark/.

[6] Docker: Empowering app development for developers. https://www.docker.com/.

[7] Emulab: a time- and space-shared platform for research, education, and development in distributed systems and networks. https://www.emulab.net/.

[8] FCC authorizes boeing broadband satellite constellation. https://www.fcc.gov/document/fcc-authorizes-boeing-broadband-satellite-constellation.

[9] FCC Authorizes Kuiper Satellite Constellation. https://docs.fcc.gov/public/attachments/FCC-20-102A1.pdf.

[10] FCC International Bureau Filings. https://fcc.report/IBFS/.

[11] General mission analysis tool. https://gmat.atlassian.net/wiki/spaces/GW/overview?mode=global.

[12] Geomagnetic storm. https://en.wikipedia.org/wiki/Geomagnetic_storm.

[13] High performance spaceflight computing (HPSC). https://www.nasa.gov/directorates/spacetech/game_changing_development/projects/HPSC/.

[14] HPE spaceborne computer. https://www.hpe.com/us/en/compute/hpc/supercomputing/spaceborne.html.

[15] Kessler syndrome. https://en.wikipedia.org/wiki/Kessler_syndrome.

[16] Norad two-line element sets current data. https://www.celestrak.com/NORAD/elements/.

[17] NTP: The Network Time Protocol. http://www.ntp.org/.

[18] Nvidia Jetson TX2 Module. https://developer.nvidia.com/embedded/jetson-tx2.

[19] Open vswitch manual. http://www.openvswitch.org/support/dist-docs/ovs-vsctl.8.txt.

[20] PLANETLAB: an open platform for developing, deploying and accessing planetary-scale services. https://planetlab.cs.princeton.edu/.

[21] RAD5545 SpaceVPX single-board computer. Multi-core single-board computer. https://www.baesystems.com/en-media/uploadFile/20210404061759/1434594567983.pdf.

[22] RAD750 family of radiation-hardened products. https://www.baesystems.com/en-media/uploadFile/20210404044504/1434555689265.pdf.

[23] Raspberry pi in space! https://www.raspberrypi.com/news/raspberry-pi-in-space/.

[24] Raspberrypi fundation. https://www.raspberrypi.org/.

[25] Roscosmos for space flights, cosmonautics programs, and aerospace research. http://en.roscosmos.ru/.

[26] SatNOGS – Open Source global network of satellite ground stations. https://network.satnogs.org/.

[27] Scrapy. https://scrapy.org/.

[28] Single-event upset. https://en.wikipedia.org/wiki/Single-event_upset.

[29] Skyfield. https://rhodesmill.org/skyfield/.

[30] SpaceX is Giving the Internet Lift With Starlink. https://subspace.com/resources/spacex-is-giving-the-internet-lift-with-starlink.

[31] SpaceX's Starlink user terminal. https://arstechnica.com/information-technology/2020/12/teardown-of-dishy-mcflatface-the-spacex-starlink-user-terminal/.

[32] Speed check: Starlink performance. https://www.speedcheck.org/starlink-performance-2021/.

[33] Starlink analysis at the carinthia university of applied sciences (CUAS). https://forschung.fh-kaernten.at/roadmap-5g/files/2021/07/Starlink-Analysis.pdf.

[34] Starlink: high-speed, low latency broadband Internet. https://www.starlink.com/.

[35] Systems Tool Kit (STK). https://www.agi.com/products/stk.

[36] Telesat. https://www.telesat.com/.

[37] The BIRD Internet Routing Daemon. https://bird.network.cz/.

[38] UCS satellite database. https://www.ucsusa.org/resources/satellite-database.

[39] Amazon. AWS Ground Station. https://aws.amazon.com/ground-station/.

[40] D. Bhattacherjee and A. Singla. Network topology design at 27,000 km/hour. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, pages 341–354. ACM, 2019.

[41] K. C. Castonguay. Additive manufacture of propulsion systems in low earth orbit. Technical report, Air Command and Staff College, Air University Maxwell AFB United States, 2018.

[42] R. Chen and W. Gao. TransFi: emulating custom wireless physical layer from commodity wifi. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MOBISYS)*, 2022.

[43] B. Denby and B. Lucia. Orbital edge computing: Machine inference in space. *IEEE Computer Architecture Letters*, 18(1):59–62, 2019.

[44] B. Denby and B. Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 939–954. ACM, 2020.

[45] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac. Distrinet: a mininet implementation for the cloud. *ACM SIGCOMM Computer Communication Review*, 51(1):2–9, 2021.

[46] L. Dreyer. Latest developments on SpaceX's Falcon 1 and Falcon 9 launch vehicles and Dragon spacecraft. In *Aerospace conference*. IEEE, 2009.

[47] E. Ekici, I. F. Akyildiz, and M. D. Bender. Datagram routing algorithm for leo satellite networks. In *Proceedings of International Conference on Computer Communications (INFOCOM)*, volume 2, pages 500–508 vol.2. IEEE, 2000.

[48] W. Frick and C. Niederstrasser. Small launch vehicles - A 2018 state of the industry survey.

[49] G. Giuffrida, L. Diana, F. de Gioia, G. Benelli, G. Meoni, M. Donati, and L. Fanucci. Cloudscout: a deep neural network for on-board cloud detection on hyperspectral images. *Remote Sensing*, 12(14):2205, 2020.

[50] G. Giuffrida, L. Fanucci, G. Meoni, M. Batič, L. Buckley, A. Dunne, C. van Dijk, M. Esposito, J. Hefele, N. Vercruyssen, G. Furano, M. Pastena, and J. Aschbacher. The sat-1 mission: The first on-board deep neural network demonstrator for satellite earth observation. *IEEE Transactions on Geoscience and Remote Sensing*, 60:1–14, 2022.

[51] G. Giuliari, T. Ciussani, A. Perrig, and A. Singla. Icarus: Attacking low earth orbit satellite networks. In *USENIX Annual Technical Conference (ATC)*, pages 317–331. USENIX, 2021.

[52] G. Giuliari, T. Klenze, M. Legner, D. Basin, A. Perrig, and A. Singla. Internet backbones in space. *ACM SIGCOMM Computer Communication. Review.*, 50(1):25–37, Mar. 2020.

[53] E. W. Gretok, E. T. Kain, and A. D. George. Comparative benchmarking analysis of next-generation space processors. In *Aerospace Conference*. IEEE, 2019.

[54] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems (TOCS)*, 29(2):1–48, 2011.

[55] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Conference on emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 2012.

[56] M. Handley. Delay is not an option: Low latency routing in space. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets)*, page 85–91, 2018.

[57] M. Handley. Using ground relays for low-latency wide-area routing in megaconstellations. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets)*, page 125–132. ACM, 2019.

[58] Y. Hauri, D. Bhattacherjee, M. Grossmann, and A. Singla. "Internet from Space" without Inter-Satellite Links. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets)*, page 205–211. ACM, 2020.

[59] Internet World Stats. World internet usage and population statistics. https://www.internetworldstats.com/stats.htm, 2021.

[60] S. Kassing, D. Bhattacherjee, A. B. Águas, J. E. Saethre, and A. Singla. Exploring the "Internet from Space" with Hypatia. In *Proceedings of the Internet Measurement Conference (IMC)*, page 214–229. ACM, 2020.

[61] Z. Lai, H. Li, and J. Li. StarPerf: Characterizing Network Performance for Emerging Mega-Constellations. In *28th International Conference on Network Protocols (ICNP)*. IEEE, 2020.

[62] Z. Lai, W. Liu, Q. Wu, H. Li, J. Xu, and J. Wu. SpaceRTC: Unleashing the Low-latency Potential of Mega-constellations for Real-Time Communications. In *Proceedings of International Conference on Computer Communications (INFOCOM)*, pages 1339–1348. IEEE, 2022.

[63] Z. Lai, Q. Wu, H. Li, M. Lv, and J. Wu. OrbitCast: Exploiting Mega-Constellations for Low-Latency Earth Observation. In *29th International Conference on Network Protocols (ICNP)*. IEEE, 2021.

[64] X. Li, F. Tang, J. Liu, L. T. Yang, L. Fu, and L. Chen. AUTO: Adaptive congestion control based on multi-objective reinforcement learning for the satellite-ground integrated network. In *USENIX Annual Technical Conference (ATC)*, pages 611–624. USENIX Association, 2021.

[65] T. M. Lovelly. *Comparative Analysis of Space-Grade Processors*. PhD thesis, University of Florida, 2017.

[66] B. Lucia, B. Denby, Z. Manchester, H. Desai, E. Ruppel, and A. Colin. Computational nanosatellite constellations: Opportunities and challenges. *GetMobile: Mobile Computing and Communications*, 25(1):16–23, 2021.

[67] man7.org. tc(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[68] MININET. An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/.

[69] M. K. Mukerjee, C. Canel, W. Wang, D. Kim, S. Seshan, and A. C. Snoeren. Adapting TCP for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020.

[70] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, 2015.

[71] W. M. Organization. Observing systems capability analysis and review tool. https://space.oscar.wmo.int/satellites.

[72] D. Pediaditakis, C. Rotsos, and A. W. Moore. Faithful reproduction of network experiments. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems (ANCS)*, pages 41–52, 2014.

[73] S. Services. Petition of Starlink Services, LLC for Designation as an Eligible Telecommunications Carrier. https://www.mass.gov/doc/dtc-21-1-starlink-final-order/download, 2021.

[74] V. Singh, A. Prabhakara, D. Zhang, O. Yağan, and S. Kumar. A community-driven approach to democratize access to satellite ground stations. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MOBICOM)*. ACM, 2021.

[75] A. Singla. SatNetLab: a call to arms for the next global internet testbed. *ACM SIGCOMM Computer Communication Review*, 51(2):28–30, 2021.

[76] SNS3. Satellite network simulator 3. https://www.sns3.org/content/home.php.

[77] D. Vasisht and R. Chandra. A Distributed and Hybrid Ground Station Network for Low Earth Orbit Satellites. In *Proceedings of the 19th Workshop on Hot Topics in Networks (HotNets)*, page 190–196. ACM, 2020.

[78] D. Vasisht, J. Shenoy, and R. Chandra. L2D2: low latency distributed downlink for LEO satellites. In *Proceedings of the ACM SIGCOMM Conference*, pages 151–164. ACM, 2021.

[79] E. Weingärtner, F. Schmidt, H. Vom Lehn, T. Heer, and K. Wehrle. SliceTime: A Platform for Scalable and Accurate Network Emulation. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2011.

[80] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl. Maxinet: Distributed emulation of software-defined networks. In *IFIP Networking Conference*, 2014.

[81] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[82] L. Wood. Satellite Constellation Design for Network Interconnection Using Non-Geo Satellites., 2002. https://openresearch.surrey.ac.uk/esploro/outputs/bookChapter/Appendix-A-Satellite-Constellation-Design-for/99513302802346.

[83] Y. Wu, Z. Yang, and Q. Zhang. A Novel DTN Routing Algorithm in the GEO-Relaying Satellite Network. In *The 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, pages 264–269, 2015.

[84] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein. Pantheon: the training ground for Internet congestion-control research. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, 2018.

[85] Y. Zheng and D. M. Nicol. A virtual time system for openvz-based network emulations. In *Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 2011.

POLYCORN: Data-driven Cross-layer Multipath Networking for High-speed Railway through Composable Schedulerlets

Yunzhe Ni[ℙ], *Feng Qian*[𝕄], *Taide Liu*[ℙ], *Yihua Cheng*[ℙ], *Zhiyao Ma*[ℙ], *Jing Wang*[ℙ]
Zhongfeng Wang[𝔾], *Gang Huang*[ℙℍ], *Xuanzhe Liu*[ℙℍ], *Chenren Xu*[ℙℤℍ✉*]
[ℙ]*Peking University* [𝕄]*University of Minnesota – Twin Cities* [𝔾]*China Railway Gecent Technology Co., Ltd*
[ℤ]*Zhongguancun Laboratory* [ℍ]*Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Abstract – Modern high-speed railway (HSR) systems offer a speed of more than 250 km/h, making on-board Internet access through track-side cellular base stations extremely challenging. We conduct extensive measurements on commercial HSR trains, and collect a massive 1.79 TB GPS-labeled TCP-LTE dataset covering a total travel distance of 28,800 km. Leveraging the new insights from the measurement, we design, implement, and evaluate POLYCORN, a first-of-its-kind networking system that can significantly boost Internet performance for HSR passengers. The core design of POLYCORN consists of a suite of composable multipath schedulerlets that intelligently determine what, when, and how to schedule user traffic over multiple highly fluctuating cellular links between HSR and track-side base stations. POLYCORN is specially designed for HSR environments through a cross-layer and data-driven proactive approach. We deploy POLYCORN on the operational LTE gateway of the popular Beijing-Shanghai HSR route at 300 km/h. Real-world experiments demonstrate that POLYCORN outperforms the state-of-the-art multipath schedulers by up to 242% in goodput, and reduces the delivery time by 45% for instant messaging applications.

1 Introduction

High-speed railway (HSR) systems, which offer a speed of 250+ km/h, revolutionize inter-city travel. Internet services on HSR are typically provided by track-side cellular base stations and an on-board proxy relaying data between WiFi APs and the cellular infrastructure [1–7]. However, the ultra-fast speed of HSR poses unprecedented challenges in bringing seamless Internet service to passengers because of the intermittent link connectivity characteristic – handover happens every less than 10 seconds [8] and the handover failure may cause a "blackout" period of up to 10 seconds [9], as reported by previous studies.

It is known that MPTCP [10] (or multipath transport in general) can leverage path diversity (with each path associated with a cellular carrier or mobile network operator) to improve link/connection robustness, as demonstrated in low mobility scenarios [11–14]. However, applying multipath transport to HSR networking is very challenging. Under such extreme mobility, the network performance fluctuates subsecond level [15], leading to 1636x higher variance in RTT than in static or low mobility scenarios [16] – this is a sharp contrast to the common assumption that a relatively stable network condition may last for several RTTs, which is leveraged by the state-of-the-art MPTCP schedulers [17, 18] for making scheduling decisions. Indeed, as to be demonstrated in §6.2, the inaccurate link quality estimation and scheduling decision informed by inaccurate throughput and RTT observations often lead to poor performance under extreme mobility.

In this paper, we argue that in contrast to the state-of-art multipath schedulers that rely on instantaneous performance measurement for making reactive scheduling decisions, we should carefully mine the networking features specific to the extreme mobility scenario, identify the main events leading to predictable failures, and design proactive scheduling strategies accordingly. For this purpose, we conduct real-world measurements on the popular Beijing-Shanghai route in China traveling at an average speed of 300 km/h. Over 24 trips spanning three weeks, we collected 1.79 TB data covering a total travel distance of 28,800 km. Our study differs from all prior HSR networking measurement studies [8, 15, 19, 20]: through collaborating with the on-board HSR ISP, we obtain precise location (from the GPS receiver mounted on the carriage roof) for every collected network performance sample. This allows us to statistically correlate the train's physical location with various network performance events, enabling many key analyses and henceforth the design of our system.

Leveraging our unique dataset, we identify three key aspects that guides our system design. First, handover failures, which incur several seconds of link disconnection, can be reasonably predicted from the train's moving trajectory. Second, the multipath heterogeneity (the relative performance ranking across paths, *i.e.,* carriers) is highly dynamic, oftentimes changing on a per-RTT basis. Third, transport-layer packet retransmission is much more common than typical cellular links. We find that on HSR, 1.8% of the TCP packets experience retransmission timeout (RTO) – among them, 24% are retransmitted more than once.

Inspired by the above findings, we develop POLYCORN, a practical networking system that significantly boosts the Internet performance for HSR passengers. It is to our best knowledge the first full-fledged system that specifically optimizes for Internet services on extreme mobility ground trans-

*✉: chenren@pku.edu.cn

portation. Leveraging multipath heterogeneity, POLYCORN distributes passengers' traffic over multiple cellular carriers for bandwidth aggregation, reduced delay, and improved reliability. In its core, POLYCORN is equipped with a novel multipath scheduler called HSRSCH, which judiciously determines transmitting which data over which path(s) in real time despite the highly dynamic network conditions. HSRSCH is tailored to HSR environments through a proactive and cross-layer approach. Its design consists of four optimizations that could be either individually or jointly applied to existing "baseline" schedulers such as the minRTT shipped with MPTCP.

• **Handover-failure-aware Path Rejection** (§4.3) has an intuitive idea: once an imminent *handover failure event* is predicted, HSRSCH temporarily disables the corresponding path so packets will not be scheduled over it to avoid the blackout period (and inter-subflow out-of-order delay). To realize this idea, we apply robust and lightweight machine learning to predict handover failures. We use two features carefully derived from our measurements: location and cell ID, which lead to an overall prediction accuracy of 80.6%.

• **Tail-aware Path Rejection** (§4.4) determines whether to use path(s) when their link conditions deteriorate. Its basic idea is to avoid scheduling tail packets, which belong to the end of a flow (*i.e., upon an end-of-flow indicator*) on slow path(s). Since HSR traffic is dominated by short-lived flows (*e.g.,* web browsing, instant messaging, mini videos), this optimization can significantly accelerate short flows' completion time. We instantiate the idea by modeling the queuing and transmission process of tail packets to guide path selection.

• **Extended Reinjection** (§4.5) detects vulnerable packets when losses occur, and retransmit them early over other paths in a batched manner. The idea stems from the bursty nature of wireless losses and consequent *excessive RTO events*, which we find to be even more prominent on HSR: one single packet loss will introduce more subsequent ones. Our approach differs from MPTCP's default packet-by-packet reinjection mechanism that is far too conservative for HSR. We carefully determine the reinjection aggressiveness based on real data to avoid putting too much burden on other paths.

• **Opportunistic Redundant Traffic Injection** (§4.6) proactively leverages idle path(s) to transmit redundant data. It not only provides extra resiliency to link quality fluctuation, but also enables the transport layer to continuously probe the path for important metrics such as RTT and RTO, which are highly dynamic when probed from HSR.

We integrate the above components through a composable scheduling framework, which treats a complex multipath scheduler as a pipeline of modularized *schedulerlets* as described above. Compared to a monolithic scheduler, our schedulerlet-based approach decouples the multipath scheduling logic, and thus significantly reduces the system complexity and development overhead, through the unified interface of schedulerlets designed by us. It also makes the system extensible and open to future optimizations (§4.2). The scheduling framework is then integrated with a multi-user/multi-path data transport mechanism (also developed by us), leading to the full-fledged POLYCORN system (§4.7).

Utilizing various system-level techniques including multipipe multiplexing [21, 22] and user-level packet interception, our implementation runs completely in the user space while maintaining full user/server transparency and high packet I/O performance. It is deployed as two proxy modules, one running on the HSR train and the other running on a cloud server, that schedule uplink and downlink traffic respectively over multiple cellular paths. POLYCORN requires no hardware or firmware modifications, and is orthogonal to HSR-specific PHY/MAC layer innovations [23–26] for cellular networks. Our implementation consists of 24K lines of code.

Through our three-year collaboration efforts with the HSR ISP's operational department, we managed to deploy POLYCORN on real HSR trains by instrumenting their onboard LTE gateways. We evaluate POLYCORN on the popular Beijing-Shanghai route at 300 km/h, with the key results as follows.

• On HSR, POLYCORN outperforms state-of-art multipath schedulers (*e.g.,* ECF [27], STMS [18], MuSher [17], BLEST [28] and MPTCP' default scheduler [10]) by 43% to 242% when downloading files with different sizes.

• POLYCORN consistently outperforms MPTCP by 61.5%, 30.6%, 64.2% on the three HSR route segments (Beijing-Jinan, 406 km; Jinan-Nanjing, 617 km; Nanjing-Shanghai, 301 km) respectively, in terms of the file download time. This indicates that POLYCORN could boost the networking performance under different HSR track-side environments.

• POLYCORN reduces the delivery time by 45% for an instant messaging application in a multi-user setting, compared to the current operational deployment of HSR Internet access.

Note that although the above results are obtained from LTE, POLYCORN is compatible with 5G networks that are being deployed along the HSR tracks [29]. We elaborate the applicability of POLYCORN on 5G in §7.

The Contributions of this paper is summarized as follows.

• New insights of extreme mobility networking characteristics derived from a massive, GPS-labeled TCP-LTE dataset covering 28,800 km travel distance.

• The design of cross-layer, proactive multipath scheduling algorithms tailored to extreme mobility networking, and their integration through a composable scheduling pipeline.

• The development of POLYCORN, the first-of-its-kind network system boosting the mobile Internet performance for all the HSR passengers.

• Deployment and extensive evaluations of POLYCORN on commercial HSR trains in the wild.

This work does not raise any ethical issues.

2 Networking Performance Measurement

To motivate the design of POLYCORN, we conduct real-world measurements of mobile networking performance on HSR. Our study is unique in two aspects. First, all our measured network performance samples have precise GPS coordinates, as opposed to coarse-grained cell IDs used in previous studies [8, 15]. Second, leveraging the unique GPS-labeled data, we offer new insights such as the predictability of failed LTE handovers on HSR.

2.1 Data Collection Methodology

A major challenge of collecting fine-grained location on HSR is a lack of GPS signal in the carriage due to electromagnetic shielding. To overcome it, we collaborate with the China Railway Gecent Technology operating China's HSR WiFi platform. We next describe our data collection setup in detail.

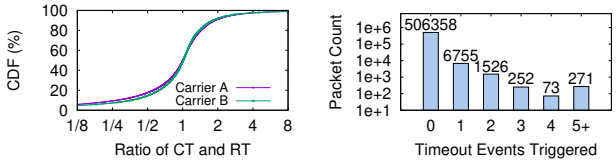
Onboard LTE Gateway. It is deployed by China's HSR WiFi carrier in the server room on each train. This gateway serves two purposes: in the upstream, it connects to track-side LTE base stations for Internet access; in the downstream, it connects to the WiFi access points (802.11ac APs) serving passengers in each carriage through a wired local area network (LAN). The gateway is equipped with multiple SIM cards of two major cellular carriers, as well as a 2×2 MIMO antenna mounted on top of the server room carriage. The GPS receiver is also mounted on the carriage roof, allowing precisely tracking the location and speed of the train. We are permitted to access the GPS data and use two of the LTE interfaces exclusively (*i.e.,* there is no other user traffic over these interfaces during data collection). We conduct data collection using iPerf [30], tshark [31], and Quectel LTE QLog[1] to measure the available bandwidth, capture packet traces, and record LTE control-plane messages, respectively. No prior study to our knowledge has leveraged such a unique infrastructure for HSR network measurement and optimization.

Measurement Server. We deploy two co-located servers (4× Intel Xeon Skylake 6133 2.5 GHz CPU with 8 GB RAM) in a major cloud service provider in China. Each server serves measurement requests for one LTE carrier. The servers are located in Shanghai that is 20 to 1,300 km away from our studied HSR route. We conduct wired experiments from several hosts near the HSR route to the two servers, and the throughput (RTT) are measured to be ≥50 Mbps (≤ 33 ms), which is far above (below) the corresponding metric measured on HSR. This indicates that the Internet is not the performance bottleneck for the end-to-end (*i.e.,* HSR-to-server) path.

Route and Duration. We carried out experiments on the "Fuxing" high-speed trains between Beijing and Shanghai, the top-two cities in China. This 1,318 km route is one of the busiest railway routes in China, with an annual passenger volume of 215 million. The average train speed is 300 km/h. Over 24 trips spanning three weeks, we collected 1.79 TB data covering a total travel distance of 28,800 km. Our dataset consists of the following cross-layer records: (1) The *GPS location* of the train updated every second; (2) The *packet traces and downlink TCP Throughput* collected by a long-lived iPerf session running on the LTE gateway; the server uses the BBR congestion control [32] that is known to yield a more accurate bandwidth estimation compared to widely deployed TCP CUBIC [33]; (3) The *LTE lower-layer information* including cell ID, signal strength (RSRP), and the LTE signaling messages collected by Quectel LTE QLog and parsed by MobileInsight [34]. We made the dataset publicly available in: https://soar.group/projects/hsrnet/dataset.html.

2.2 Throughput & Latency Characterization

Leveraging our large dataset, we begin with basic characterizations of throughput and latency. Across the entire dataset, the 25th, 50th, and 75th-percentile downlink TCP throughput of Carrier A (B)[2] are measured to be 2.56 (4.60), 6.48 (10.67), and 12.42 (19.73) Mbps, respectively. Regarding the latency, the 25th, 50th, and 75th-percentile RTT of carrier A (B) are 123 (147), 185 (191), and 315 (348) ms, respectively. We observe that both throughput and latency exhibit high temporal fluctuations. To quantify them, we compute the ratio between the average throughput in the current time window $[t_0 - \Delta t, t_0]$, denoted as CT, and the average throughput in the previous window $[t_0 - 6\Delta t, t_0 - \Delta t]$, denoted as RT, where t_0 is the current time. Δt is empirically chosen as 0.2 seconds, the median RTT when HSR travels at 300 km/h. Fig. 1a plots the distribution of the throughput ratio defined above. As shown, in 26.4% (Carrier A) or 22.6% (Carrier B) of the cases, CT/RT is lower than 50% or higher than 200%, confirming the high throughput fluctuation. The latency variation is also prominent (figure not shown). This leads to frequent TCP Retransmission Timeout events (RTOs), which are experienced by 1.8% of the packets. We even observe that the transmissions of many packets experience more than one RTO, as shown in Fig. 1b ("0" indicates no RTO is triggered).



(a) Throughput. (b) RTT.

Figure 1: TCP performance temporal variation.

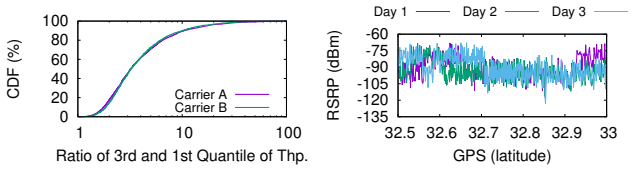
2.3 Predictability of Networking Performance

Since trains move along fixed rail tracks, it is anticipated that the networking performance is predictable, similar to what has been reported for lower-speed vehicles [35]. On HSR, however, the predictability may be affected by the ultra-high speed. No prior study to our knowledge has studied the predictability of HSR networking performance.

[1] A proprietary tool offered by the gateway vendor for collecting LTE log data from their LTE modems.

[2] China Mobile and China Unicom respectively.

We begin with exploring a straightforward approach of predicting the TCP throughput using the train's trajectory. For each <location, direction> pair, we compute the average throughput near the location ($\pm$ 1 km) in all trips to smooth out the temporal variation, aggregate all those samples and calculate the ratio between the 75-th and the 25-th percentile values. As shown in Fig. 2a, the median ratio is 3.21 and 3.24 for Carrier A and B, respectively, and it may reach up to 100[3]. The results indicate that, unlike low-speed transportation, throughput prediction in HSR is very challenging. The main reason is that the extreme mobility introduces complex stochastic channel fading, which can cause significant temporal variation in received signal strength and henceforth high data rate fluctuation at the same location. This is exemplified in Fig. 2b, which plots the RSRP values of Carrier A over a 40 km route measured on three different days. As shown, the link quality not only exhibits randomness across the three trips but also lacks spatial locality during the same trip. We also would to note that this unpredictable pattern is jointly determined by the highly dynamic channel condition and its complex interaction with TCP congestion control – a small difference in wireless channel condition may result in a big difference in future TCP performance. In addition, the prediction result also depends on a specific congestion control (CC) algorithm, which makes the design space for throughput prediction even more challenging.



(a) Variability of Throughput. (b) Location vs. RSRP.

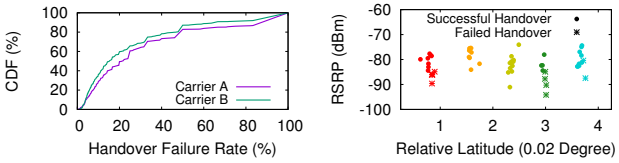
Figure 2: Measurement study of location-aware TCP throughput repeatability and predictability.

Given the difficulty of throughput prediction for HSR, POLYCORN takes a unique approach of predicting *handover failures*. Due to its high speed, HSR experiences much more frequent handovers compared to low-speed vehicles. More importantly, in HSR, handovers are more likely to fail. A *failed handover* occurs when a UE disconnects from or loses connection to the current base station but is not yet connected to the new base station. Failed handovers bring negative performance impact, including packet losses and their incurred retransmission timeout (RTO) that force TCP to enter a slow start. In our dataset, the performance impact is measured to be at least 1 second and can last as long as 10 seconds. In contrast, a successful handover usually incurs shorter than 100 ms of TCP throughput disruption.

We next explore the predictability of failed handovers given their importance. Our dataset records 32,231 and 45,656 handovers for Carrier A and B, respectively. Among them, the fraction of failed handovers is 6.22% and 5.47%, respectively, significantly higher than those experienced by low-speed vehicles. We observe that 47.33% (40.35%) of the source cells (from which the handover initiates) of Carrier A (Carrier B) experiences at least one handover failure in our three-week measurement. Fig. 3a plots the handover failure rate, defined as the ratio of failed handovers, across the cells[4] experienced at least one handover failure. As shown, for both carriers, more than 30% of the cells have a failure rate between 20% and 80%, indicating that it is infeasible to predict failed handovers only using cell ID. Nevertheless, we identify two noticeable features that can facilitate prediction of handover failures. First, the RSRP and failure rate are found to be negatively correlated, for handover messages are more likely to be lost under lower SNR. Second, a handover that is triggered late (compared to historically recorded handovers at the same location) is more likely to fail: due to HSR's high speed, there is simply not enough time for a late handover to complete. This is confirmed by the statistics shown in Tab. 1, which plots the successful rates across all handovers under four scenarios. (1) Handovers starting within 200 m of the Earliest Handover Positions (EHP) of their source cells. (2) Handovers starting at least 200 m beyond EHP; (3) Handovers with $\geq$-95dB RSRP when they start; (4) Handovers with $<$-95dB RSRP when they start. Here the EHP of a cell s is defined as the earliest geographic location (w.r.t. the train's moving direction) of all the handovers with a source cell s when they start. As shown in Tab. 1, handovers that start early or have high RSRP values have higher chances of success compared to late or low-RSRP handovers, respectively. We also plot some individual handovers versus signal strength and location in Fig. 3b where a dot (star) represents a successful (failed) handover, and the colors correspond to different cells.

Scenario	Carrier A	Carrier B
Distance to EHP < 200m	75.4%	83.7%
Distance to EHP $\geq$ 200m	65.9%	71.8%
RSRP $\geq$ -95dBm	89.2%	83.8%
RSRP < -95dBm	52.9%	67.7%

Table 1: Handover success rate.



(a) Failure rate. (b) Pattern snapshot.

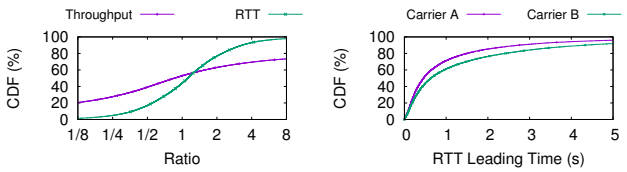
Figure 3: Handover pattern study.

[3]The ratio could be even higher with finer-grained location granularity due to the bursty TCP performance on HSR.

[4]Unless explicitly mentioned, *cell* means <cell, direction> pair.

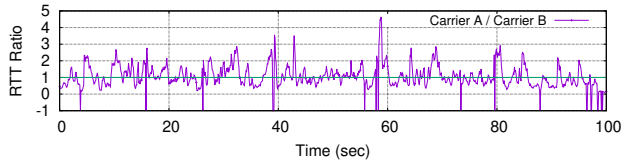
2.4 Multipath Heterogeneity

Recall that the on-board LTE gateway (§2.1) is equipped with SIM cards of two carriers. We next explore whether the performance of the two carriers are correlated or not. Specifically, we compute the throughput and RTT ratios between the two carriers in a synchronous manner. We find that 39.2% (16.9%) of the computed throughput (RTT) ratios are lower than 0.5, and 37.0% (23.2%) of the throughput (RTT) ratios are higher than 2.0, as plotted in Fig. 4a. This suggests that the two carriers' performance is indeed heterogeneous when accessed from HSR, and the two carriers can performance-wise complement each other. We further quantify at what time granularity one carrier can consistently outperform the other. Specifically, we define the *RTT Leading Time* as the longest consecutive period during which one carrier always has a lower RTT than the other. As shown in Fig. 4b, the median RTT leading time for Carrier A and B are 457 ms and 632 ms respectively. This indicates that the multipath heterogeneity on HSR is highly dynamic, changing every 2 to 4 RTTs (see also Fig. 4c), attributed to HSR's extreme mobility.



(a) Throughput/RTT ratio.

(b) RTT leading time.



(c) Snapshot of RTT ratio time series.

Figure 4: Measurement study of path diversity.

2.5 Implications on System Design

We summarize key findings of our measurements and their implications on system design.

• As the mobile networking performance is highly fluctuating on HSR, one needs to continuously probe the cellular link to get fresher link quality estimations and use it strategically;

• One may tackle the link dynamics by proactive reinjection, as the RTO-based retransmission is often inefficient (§2.2);

• One could leverage the predictability of handover failures to take early actions before losing the connectivity (§2.3);

• One can further leverage the path heterogeneity to mitigate the volatility on individual paths. The path selection requires judicious decisions based on traffic patterns, real-time link quality monitoring, and handover failure prediction (§2.4).

3 Handover Failure Prediction

In this section, we present how to leverage the available yet reliable information to predict handover failures, which is crucial to improving networking performance in our frequently disconnected networking environment.

Handover Success/Failure Determination Methodology. Our measurement in §2.3 provides evidence that HSR handover failures, which disrupt TCP performance for several seconds, are potentially predictable – they are more likely to fail if happened at a latter location in the overlap zone (for handover) and/or if RSRP is lower. Herein, we formulate handover success/failure determination task as a classification problem, and adopt SVM, a lightweight supervised machine learning algorithm fed with location (*i.e.,* longitude, latitude) and RSRP values when UE disconnects from the source cell as features and handover result as labels. We log all these relevant information into a database called LinkDB deployed on both mobile relay and remote proxy (see Fig. 7) and train the SVM with linear kernel function and L2 loss function for each source cell. By using location, the percentage of the linearly separable (successful and failed) handovers is 72.8% and 71.4% for carrier A and B respectively; by using RSRP, this number is reported as 73.5% and 67.1%. When jointly use location and RSRP, this number raises up to 92.6% and 89.2%. This data shows that for most cells, the handover result could be accurately inferred from the location and/or RSRP when the handover starts. Another implication is that if a fresh handover event is close to the historical handover failure data in the feature space, it is very likely to fail.

Handover Failure Prediction in POLYCORN. Although the aforementioned offline analysis shows promising results in determining whether the handover is successful or failed based on location and RSRP data, it is not straightforward to turn it into a practical online handover failure predictor. The main challenge is that the handover failure has to be predicted *in advance* so as to be useful for guiding interface scheduling, especially for downlink traffic. In other words, the time advance needs to take into account the tens or hundreds of milliseconds of delay for mobile relay to deliver the handover failure precaution signal to the remote proxy. In our data analysis, we also find that the LTE chipset can delay the RSRP log reporting to userspace for up to 200 ms. This fact together with RSRP's highly fluctuating nature (Fig. 2b) makes RSRP an error-prone feature for SVM to use for online handover failure prediction. Therefore, POLYCORN has to rely on location information only since it is truly predictable given the reliable train speed. In practice, the mobile relay sends its associated cell ID, train's speed and location, and current time to the remote proxy. On the remote proxy, LinkDB provides information about all historical handovers from this cell and calculates the predicted location of handover failure $\hat{L_{HOF}}$, defined as the average location of all handover failures. Then, LinkDB predicts $\hat{t_{HOF}}$, the time that train passes $\hat{L_{HOF}}$. If the current moment is approaching $\hat{t_{HOF}}$ (to be elaborated in §4.3), POLYCORN predicts that the handover in this cell would fail because it is too late to initiate it even from now.

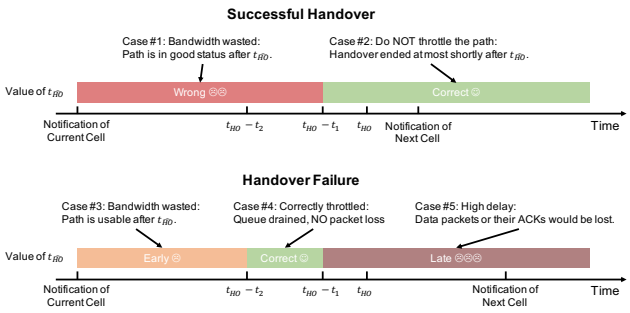


Figure 5: Types of Handover Prediction Results.

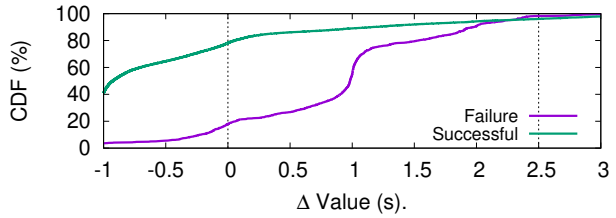


Figure 6: Handover Prediction Validation.

Prediction Validation. We consider the case where we throttle the path from $t_{\hat{HOF}}$ to the time when the UE connects to the next cell. The point is, if the handover failed, we should throttle the path right before the handover to drain the queue on it. Otherwise, we should only throttle the path for a very short period, or optimally do not throttle it. As illustrated in Fig. 5, let t_{HO} be the groundtruth value of handover time, $t_1 < t_2$ where t_1 and t_2 are two non-negative numbers, and hence there are five combinations of $\Delta = t_{HO} - t_{\hat{HOF}}$ and handover results. We set t_1 to 0 seconds where $t_{\hat{HOF}}$ exactly matches t_{HO}, and set t_2 to 2.5 seconds, beyond which the side effect brought by early prediction overweighs its benefit. Fig. 6 plots the distribution of Δ, where $t_{\hat{HOF}}$ is predicted using leave-one-trip-out cross-validation over the entire dataset. For handover failure, 80.6%, 1.3%, and 18.1% of the prediction results are correct, late, and early, respectively. For successful handovers, the correct rate is 78.2%. Here we note that POLYCORN seeks for a conservative approach towards handovers prediction and path throttling decision – it prioritizes avoiding the penalty of a handover failure misprediction and considers it acceptable to waste available bandwidth in the cases that successful handover predicted as failure (Case 1) or successfully predicting the handover failure but in a earlier moment (Case 3) – in both cases the networking performance already starts to degrade when approaching the handover point anyway. As to be shown in §6.2, such coarse-grained results is adequate for our system given the high speed and the GPS system errors.

4 System Design of POLYCORN

We now present the design of POLYCORN, a software solution for high-performance Internet access for ultra-high-speed transportation. The design goals of POLYCORN include the following: (1) *Be resilient to extreme mobility environment.*

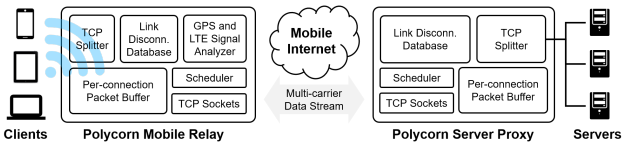


Figure 7: POLYCORN Architecture Overview.

POLYCORN should survive highly fluctuating network performance and inaccurate link quality estimations. (2) *Faster flow completion.* As opposed to bandwidth-intensive such as bulk data transfer [8], networked applications used by passengers, such as instant messaging and web browsing, typically have short or small sessions. It is therefore important to reduce the flow completion time. (3) *Effectively use multiple cellular carriers*. The multipath heterogeneity revealed in §2.4 should be leveraged for robust traffic delivery. (4) *Be practical for real-world deployment.* POLYCORN should be easy to deploy and ideally, be transparent to client and server applications and require no infrastructure modifications.

4.1 Overall Architecture

The high-level architecture of POLYCORN is illustrated in Fig. 7. As shown, POLYCORN leverages the dual-proxy architecture [22]. One proxy deployed at the on-board mobile relay (*i.e.,* cellular gateway) multiplexes passengers' uplink traffic over the paths of multiple cellular carriers; another proxy deployed at a cloud server performs the reverse operation of demultiplexing the traffic and delivering them to the destination servers. Each in-cloud proxy is paired with one on-board mobile relay, and more pairs can be flexibly set up for scaling up the service for larger HSR network. Downlink traffic is handled in a symmetric way: the cloud-side and on-board proxy perform multiplexing and demultiplexing, transparently. The two proxies are essential for providing transparent transport-layer multipath to off-the-shelf hosts.

The dual proxies offer a centralized place for multipath scheduling, *i.e.,* deciding which traffic should be transmitted over which path(s). One path is one subflow that exclusively uses one network interface (*i.e.,* SIM card). The scheduler for uplink and downlink traffic resides on the on-board LTE gateway and the in-cloud proxy, respectively. Multipath scheduling is one of the most critical components of a multipath transport system, in particular for HSR networking where the paths' performance is individually fluctuating and collectively heterogeneous. We are unaware of any multipath scheduler specifically designed for extreme mobility transportation, and POLYCORN's design fills this gap.

4.2 Composable Multipath Scheduler

Our measurement study in §2.5 suggests that multipath transport for HSR networking needs to consider multiple dimensions: performance fluctuation, predictable handover failures, and path heterogeneity, *etc.* To tackle such complexity, we adopt a novel framework that treats a multipath scheduler as a pipeline of modularized *schedulerlets*. Each schedulerlet encapsulates a multipath scheduling functionality that ma-

Performance Issue	Mitigation Strategy	Schedulerlet
Disruption due to handover failure	Filter paths facing imminent handover failures based on prediction	§4.3
Suboptimal scheduling due to path heterogeneity	Filter paths that lengthen TCP flow completion due to tail packets	§4.4
Single packet experiencing multiple RTOs	Proactively reinject packet clusters facing excessive retransmissions	§4.5
Stale performance metrics on idle paths	Opportunistically deliver redundant data over unselected paths	§4.6

Table 2: Logic flow from performance issues to designs.

nipulates three sets of paths: a *selected set* $\mathbb{S}$ containing the currently selected path(s) for data transmission, a *candidate set* $\mathbb{C}$ containing the candidate paths that can be selected, and an *unavailable set* $\mathbb{U}$ containing the unavailable paths that by default cannot be selected. The purpose of having $\mathbb{U}$ is to restrict path selection to only a subset of all paths. Initially, all the paths belong to $\mathbb{C}$. Depending on which set(s) to manipulate, we classify the schedulers into different categories: (1) a *candidate filter* that moves path(s) from $\mathbb{C}$ to $\mathbb{U}$; (2) a *selection filter* that moves paths from $\mathbb{S}$ to $\mathbb{C}$; and (3) a *soft selector* that moves path(s) from $\mathbb{C}$ to $\mathbb{S}$; (4) a *hard selector* that moves path(s) from $\mathbb{C}$ or $\mathbb{U}$ back to $\mathbb{S}$. The purpose of having a hard selector is to provide a mechanism that can "revive" any path. This is useful when paths' conditions are highly dynamic; it also ensures the completeness of the framework.

The schedulerlets are then strategically arranged to form the overall scheduling pipeline. Compared to a monolithic scheduler, our schedulerlet-based approach decouples the multipath scheduling logic, and thus significantly reduces the system complexity and development overhead, through the unified interface of schedulerlets (modifying $\mathbb{S}$, $\mathbb{C}$, and/or $\mathbb{U}$). Note that we do not claim that our design can achieve any optimality, since the "local" optimalities achieved by individual schedulerlets do not necessarily translate into a "global" optimality. Nevertheless, from a practical perspective, formulating a global optimization problem and solving it through a monolithic scheduler is extremely challenging due to the large solution space, real-time requirement, and volatile network dynamics. Therefore, we believe our "decouple-then-integrate" design achieves a right balance among practicality, simplicity, and performance, as to be thoroughly evaluated in §6.

We now consider how to instantiate the above generic framework into the concrete design of HSRSCH, the multipath transport scheduler for POLYCORN. The high-level design principle is to identify scenarios where vanilla MPTCP performs poorly, based on our extensive field studies, and improve them through judiciously designed schedulerlets. Specifically, Tab. 2 lists our identified performance issues, mitigation strategies, and the corresponding schedulerlets of HSRSCH, which will be detailed in the rest of §4. Here we describe the high-level scheduling pipeline. As shown in Fig. 8, the pipeline begins with a *candidate filter* schedulerlet that removes "bad" paths facing an imminent handover failure (§4.3), followed by a *soft selector* that performs initial, rough path selection. We find that the default minRTT scheduler

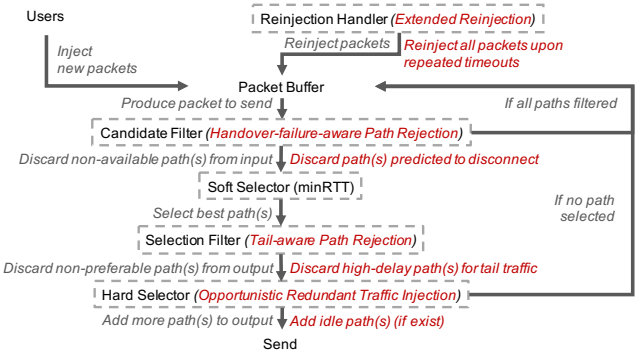


Figure 8: Composable scheduler in POLYCORN.

used by MPTCP can be properly leveraged as a soft selector, because favoring a low-latency path (when congestion window permits) is also desirable in HSR networking. Subsequently, we employ a *selection filter* to further remove certain selected paths, shortening the flow completion time (§4.4). Next, we apply a *hard selector* to make use of the remaining paths (in $\mathbb{C}$ and $\mathbb{U}$) – we use them for delivering redundant data and probing the bandwidth (§4.6). Finally, due to the high network condition volatility, performance degradation or even outage may still appear on a selected path despite the above schedulerlets. To tackle this, we also introduce a *reinjection handler* that dynamically redistributes scheduled packets to other paths. This mechanism tolerates other schedulerlets' errors and further improves the overall robustness (§4.5). Note that HSRSCH is designed to be scalable, *i.e.,* they could work with any number of available paths.

The aforementioned taxonomy of schedulerlets based on which sets ($\mathbb{S}$, $\mathbb{C}$, $\mathbb{U}$) they manipulate also helps decide the order of the schedulerlets. For example, the candidate filter is invoked first since it does not depend on other schedulerlets; the selection filter needs to examine the output of the soft selector so the former comes after the latter in the pipeline. The hard selector tries to make use of any unselected paths; it is therefore situated at the end of the pipeline.

4.3 Handover-failure-aware Path Rejection

Recall from §2.3 that handover failures incur considerable performance impact. Our first optimization has an intuitive idea: predict imminent handover failures using LinkDB (§3), and apply a schedulerlet (candidate filter) to disable the path(s) facing handover failure(s). The rationale is that, sending traffic over a path experiencing a blackout of several seconds caused by a failed handover will significantly lengthen the flow com-

pletion time. Therefore, it should be avoided at all costs. In our design, for downlink traffic scheduling[5], the gateway continuously sends the collected features (GPS reading and cell ID) to the in-cloud proxy where LinkDB runs. Those features are sent with top priority using a redundant scheduler to ensure that the proxy would receive the features in time. The proxy then predicts t_{HOF}, the interval between the current time and the next expected handover (§3). If t_{HOF} is predicted to be less than a threshold L, the proxy-side scheduler disables the path. The path will be re-enabled when the connectivity to the new cell is established. The threshold L incurs a tradeoff between bandwidth waste (occurs when disabling the path early) and performance degradation (occurs when disabling the path late). We configure L as: $L = RTT + \frac{E_{GPS}}{V_{HSR}}$. The first term is the estimated RTT between the in-cloud proxy and the LTE gateway. RTT is the lower bound of L because at least one round trip is needed for any in-flight data to be delivered with confirmation; if we send any data after $t + t_{HOF} - RTT$ (t is the current time), then we run into risks where the data/ACK delivery is affected by a failed handover. The second term $\frac{E_{GPS}}{V_{HSR}}$ accounts for the GPS localization inaccuracy, where E_{GPS} and V_{HSR} are the maximum GPS error (in meter) and the train's current speed (in m/s) respectively. Due to potentially erroneous GPS reading, the train's actual location may be ahead of the reported GPS location. Therefore, we need the second term for an additional safety margin. We conservatively set E_{GPS} to 20 m, and V_{HSR} is estimated from the recent GPS trajectory. We would also like to mention that in the corner case when all paths are predicted to experience handover failure soon, all of them would be disabled – this would effectively block all pending traffic until the mobile relay notifies the remote proxy of a new cell.

4.4 Tail-aware Path Rejection

The HSR networking performance is not only affected by handover failures, but also by the highly fluctuating channel quality due to HSR's high speed. When a path's quality deteriorates, HSRSCH needs to make a key decision: should the path be temporarily disabled? Here the tradeoff is bandwidth utilization vs. latency: skipping the path misses the opportunity of utilizing its (albeit low) bandwidth, while using the path can possibly lengthen the flow completion time compared to sending the data over a faster path.

To balance the above tradeoff, since our goal is to accelerate short flows dominating the traffic pattern on HSR, HSRSCH detects scenarios where a flow is about to end, and employs a schedulerlet (selection filter) that rejects sending *tail packets* over low-performance paths. Here tail packets reside at the end of a flow, whereas non-tail packets are at the beginning or in the middle of a flow. The rationale is that, sending a tail packet over a low-performance path is very likely to delay the flow completion. In contrast, transmitting a non-tail packet over a poor-quality path usually only incurs packet out-of-order with a negligible or small impact on the flow completion time, provided that (1) all the paths are fully utilized (*i.e.,* there is no idle period), and (2) the receiver has a large enough buffer to accommodate out-of-order packets (which we can ensure as we have control over both multiplexing proxies). To transparently identify tail packets without the knowledge of flow size, HSRSCH keeps monitoring TCP FIN or RST packets. Once a TCP FIN or RST is observed, HSRSCH marks all the packets in the send buffer and all future outgoing packets of the same flow as tail packets. We leave more sophisticated tail packet identification methods (*e.g.,* based on application semantics) as future work.

To cope with highly fluctuating network conditions, HSRSCH employs a new way of deciding whether flow f has low performance over path i. Specifically, HSRSCH compares two scenarios. In the first scenario, we schedule some packets of f on path i. The packet delivery time and henceforth the flow completion time of f is *at least*:

$$T^{-}_{i,f} = owd_i + \frac{buf_i}{bw_i}$$

where owd_i, buf_i, and bw_i are the estimated one-way delay, the send buffer occupancy level, and the estimated bandwidth of path i, respectively. $T^{-}_{i,f}$ estimates the time taken to drain the FIFO send buffer of path i prior to sending any new packet belonging to f. It is therefore a *lower bound* of f's completion time if any of its packets are scheduled on path i. In the second scenario, we do not use path i at all to schedule f. In this scenario, the optimal flow completion time of f is *at most*:

$$T^{+}_{i,f} = \min_{j \neq i} \left\{ \left(owd_j + \frac{buf_j + remain_f}{bw_j} \right) (1 + \eta_j) \right\}$$

where $remain_f$ denotes the remaining bytes of f yet to be sent, η_j is a relaxation parameter empirically defined as the ratio between path j's RTT variance and RTT, and j iterates over all the paths except path i. $T^{+}_{i,f}$ quantifies the time of transmitting all the remaining bytes of f over the best path (except i), considering the paths' bandwidth and current send buffer occupancy levels. It is a loose *upper bound* of the optimal (single-path) completion time of f, which can in fact be further reduced (albeit difficult to quantify) by distributing f over multipath. If $T^{-}_{i,f} > T^{+}_{i,f}$, it implies that there exists a better scheduling strategy of not using path i compared to any strategy of using path i; we thus determine that path i is too slow to schedule tail packets of f. See example in §A.

4.5 Extended Reinjection

Multipath transport needs to simultaneously manage multiple paths. To handle individual paths' failure, MPTCP has a built-in reinjection mechanism (also known as Opportunistic Retransmission [36]): upon RTO events, the oldest unacknowledged packet on the same path will be retransmitted (reinjected) over another path as determined by performing

[5]For brevity, we only describe downlink traffic scheduling. Uplink traffic scheduling is performed at the on-board gateway in a similar manner.

the scheduling again. We find that such a reinjection policy is too conservative since it handles reinjection on the basis of *individual packets*. In contrast, in HSR networks, packet delivery failures often occur in a *bursty* manner: if a packet experiences an RTO, the probability that the subsequent packets are delayed or lost becomes much higher. Bursty losses are common in wireless networks in general. Nevertheless, on HSR, the bursty pattern of packet losses is much more prominent as the UE is frequently disconnected from base stations due to failed handovers or low signal strength.

Inspired by the above observation, we design an Extended Reinjection mechanism for HSRSCH, whose basic idea is to detect scenarios where packets are undergoing multiple RTOs, and reinject packets in a *proactive and batched* manner to match the bursty packet loss pattern for HSR cellular access. Specifically, in our approach, when any packet experiences the k-th RTO, HSRSCH reinjects *all* the unacknowledged (*i.e.,* in-flight) packets on the same path to other path(s). Instead of executing the reinjection(s) *immediately* by invoking the scheduling algorithm multiple times (similar to what MPTCP does when executing each single reinjection operation), HSRSCH performs *lazy* reinjection: the to-be-reinjected packets are thrown back to their origin connection-level send buffers, and the actual reinjection will take place later when these packets are (re)scheduled according to their connection- and user-level priorities (at that time that destination path will also be determined[6]). The purpose of lazy reinjection is to maintain priority and fairness, *i.e.,* the reinjected packets are regarded as newly arrived so they do not bear an unfairly high priority over the reinjected path. This is particularly important when many packets belonging to the same connection are reinjected. Also note that for each reinjection, the receiver will receive at least two identical copies: the original packet and at least one reinjected packet; only the first arrived copy will be consumed by the receiver.

In the above algorithm, the parameter k incurs a tradeoff: a large k provides fewer reinjection opportunities, potentially worsening the performance on the current path where losses occur, whereas a small k makes reinjection more aggressive, adding more traffic burden on other paths. We take a data-driven approach to select the appropriate k value: in our dataset, setting k to 1, 2, and 3 incurs 47%, 15%, and 0.3% more redundant traffic, respectively. We therefore set $k = 3$ in our evaluation (§6.2).

4.6 Opportunistic Redundant Traffic Injection

The soft selector along with both filters intend to find the best path for each packet, leaving the remaining unselected path(s) idle. This will cause two major issues: (1) the available bandwidth on idle paths, despite their high latency, is wasted; (2) no performance measurement can be carried out on idle paths without traffic, and previous measurement quickly becomes stale due to the high path dynamics. To address both issues, we design a schedulerlet (hard selector) that opportunistically schedules redundant data over idle paths. This not only allows passive measurement to be conducted, but also provides extra resiliency to channel quality fluctuation, leading to further reduced flow completion time, *i.e.,* when one path experiences unexpected performance degradation, the receiver can still receive extra copies of the data delivered over other path(s). In HSRSCH, the idleness of a path is determined when no traffic has been scheduled over it for either α seconds, or β bytes worth of data, whatever occurs first. Once a path becomes idle, HSRSCH duplicates the next τ scheduled packets and transmits their duplicated copies over the idle path. If multiple idle paths are available, the duplicate copies will be transmitted over all the idle paths. We empirically set α = 1 second, β = 8 KB, and τ = 16, which were found to well balance the tradeoff between the performance and bandwidth overhead based on our on-board controlled experiments.

Besides fostering reliability under performance degradation, injecting traffic over an idle path brings another benefit: it allows the transport layer to keep the path performance statistics up-to-date. As shown in Fig. 1a, on HSR, a cellular link's quality changes almost every RTT. If no packet is sent over an idle path, TCP's built-in probing mechanism, which is piggybacked with user traffic, will be paused, and TCP will thus lose track of important metrics such as RTT and RTO. Our proactive injection design addresses this issue.

4.7 Putting Everything Together

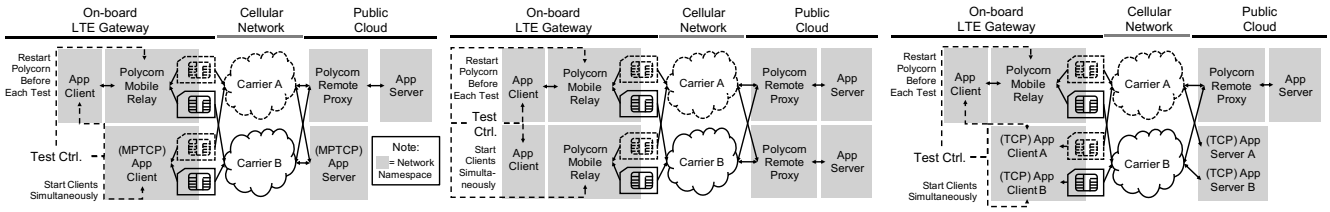
We integrate the above four schedulerlets into the composable framework introduced in §4.2. We next describe the detailed scheduling logic on both in-cloud proxy (for downlink traffic) and on-board LTE gateway (for uplink traffic).

Recall that POLYCORN is a multi-user system. In each scheduling round, HSRSCH begins with selecting a user to serve, and then picking a flow belong to the selected user. Our current implementation uses the standard proportional fair (PF) scheduling [37] for user selection, and round-robin for flow selection. More sophisticated scheduling algorithms can be plugged into our framework. Once the to-be-served flow is determined, HSRSCH schedules the flow's next packet, which is the untransmitted packet (including to-be-reinjected packets) with the smallest sequence number, as described in Fig. 8. Note that the reinjection handler runs in parallel with the scheduling thread that invokes the candidate/selection filters and soft/hard selectors.

5 Implementation

This section details the implementation of POLYCORN. Our high-level design goals consist of the following. First, POLYCORN should be practical. The required changes on clients' mobile devices should be minimized, or ideally none. Also, POLYCORN should be able to deploy on the HSR LTE gateway and keep its running components unmodified. Second, the data transport scheme should be able to schedule traffic

[6] We use a bitmap field to ensure that the same packet is not reinjected to its previously scheduled path.



(a) POLYCORN vs. MPTCP. (b) POLYCORN microbenchmark. (c) POLYCORN vs. TCP.

Figure 9: Experimental setup for different comparative evaluation.

with awareness of multiple users and connections.

Working with Unmodified Users and LTE Gateway. Based on TM^3 [21] and MPFlex [22] frameworks, POLYCORN uses TCP splitting to achieve transparency towards both clients and servers. A brief summary of TCP splitting is: when communicating with servers, POLYCORN acts as a forward proxy; when communicating with clients, POLYCORN acts as a reverse proxy. To avoid kernel modification which is not allowed by the LTE gateway vendor, we use raw socket instead of `netfilter`-based [38] kernel modules (which is used by MPFlex and TM^3) or high-performance packet I/O frameworks like DPDK [39] to capture user traffic. After capturing user traffic, we drop all user packets using `iptables` [40] to prevent the kernel from forwarding them. In this way, we implement POLYCORN completely in userspace. As for the well-known performance issue of raw sockets, our experiments show that raw sockets could operate at 300 Mbps on the LTE gateway, which is significantly higher than the peak aggregated throughput of the LTE interfaces. Moreover, POLYCORN works in a separated network namespace (netns) to avoid conflicts with runtime kernel configuration used by other programs and mitigate potential security issues. For instance, POLYCORN disables reverse-path filtering in its own netns to forward packets generated by POLYCORN with any source IP. Our design allows sensitive system configurations to be preserved in the original netns, thus isolate the security risk from normal runtime programs.

Multiuser Multipath Traffic Multiplexing. Similar to TM^3 and MPFlex, POLYCORN multiplexes user traffic onto off-the-shelf sockets to implement multipath data transfer. POLYCORN uses TCP sockets as its subflows. Although QUIC is better than TCP in terms of Head-of-Line blocking mitigation, especially in multiuser settings, we choose the "fallback" TCP sockets primarily because we encountered extensive rate limitation cases when launching QUIC flows in our measurements. This observation agrees with the findings in [41] to reveal that UDP traffic will most likely be treated as malicious flow by cellular carriers when sending in large volume, which situation might not disappear soon in most developing and under-developed countries. Therefore, we believe our choice is better for long-term real-world deployment. As for multiuser traffic scheduling, POLYCORN maintains a separated send/reinject buffer and a metadata set including user source IP, amount of sent traffic, etc, for each flow. With those metadata, in operation POLYCORN first determines which user/flow to serve, then checks the flow's send buffer to choose a packet to send. Finally, POLYCORN runs HSRSCH to choose interfaces to send the packet, as described in §4.2.

6 Evaluation

6.1 Experimental Setup

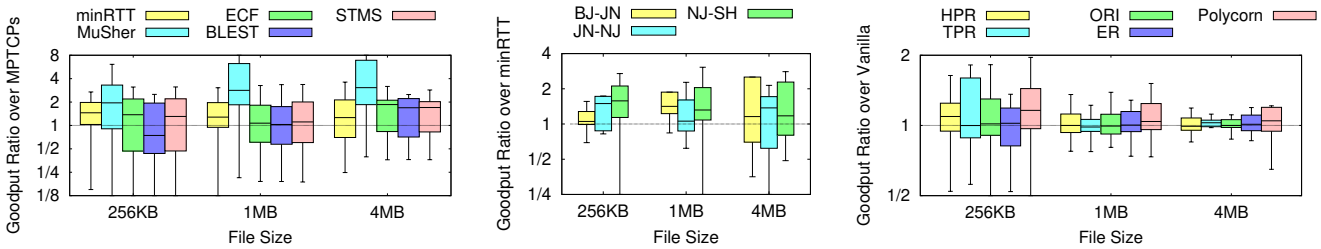
We carried out the experiments on Beijing-Shanghai HSR route, the one carrying most HSR passengers in the country.

Deployment on Operational System. We deployed POLYCORN mobile relay and server proxy on the high-speed train LTE gateways and public cloud servers respectively, with the same hardware configuration described in §2.1. More specifically, the mobile relay runs CentOS 7.3 with MPTCP kernel 0.94 (Linux 4.14) – we adhere to CentOS for POLYCORN deployment on the LTE gateway because it runs other mission-critical train-ground communication services developed by the operators and third-party IT service providers.

Fairness in Comparative Study. We made the following efforts to improve fairness in evaluating POLYCORN:

• *MPTCP Baseline* is configured in decoupled rather than default coupled congestion control mode (used in [15]) because the relay-proxy suite acting as an end-user traffic aggregation and delegation point should harness more wireless bandwidth from multiple cellular carriers instead of treating itself as a single user or session – it makes the baseline stronger and comparative evaluation fairer.

• *Pairwise study.* We carried all the experiments in the side-by-side concurrent test setting for 50 times with different software configuration tailored for fairness in different scenario, including comparing POLYCORN with MPTCP variants (Fig. 9a) and its own variants for microbenchmark (Fig. 9b) in single session bulk data download (§6.2), and comparing POLYCORN with SPTCP (Fig. 9c) and MPTCP (Fig. 9a) in multi-user instance messaging settings (§6.3). Specifically, we managed to obtain exclusive access to four SIM cards with two for each carrier respectively from the HSR WiFi service division, pair each transport software solution (*e.g.,* SPTCP, MPTCP and POLYCORN) with two cards from different carriers, and perform all the experiments on the operational HSR with a speed of 300+ km/h. Note that the scheduler and system design of POLYCORN can easily scale to more than two



(a) Comparing with state-of-art MPTCPs. (b) On different route segments. (c) Microbenchmark.

Figure 10: Bulk data downloading time comparative evaluation.

interfaces in LTE, 5G, *etc.* We choose to perform evaluations with two cards primarily because that is the maximum number we can obtain permission to access presently.

6.2 Bulk Data Download Performance

We first evaluate the performance of POLYCORN in comparison to the state-of-the-art MPTCP solutions and its own variants for microbenchmark from a single session perspective. We use fixed-size flows of 256 KB, 1 MB, 4 MB to examine POLYCORN's performance in a pairwise manner. Specifically, we choose to use average goodput ratio of test object to its opponent as the primary metric instead of showing their absolute value. This is because the cellular link quality and the associated mobile networking performance differs significantly from one trackside location to another – the large variance for a single performance profile prevents comparative quantitative illustration and analysis between the two solution in the same testing environment.

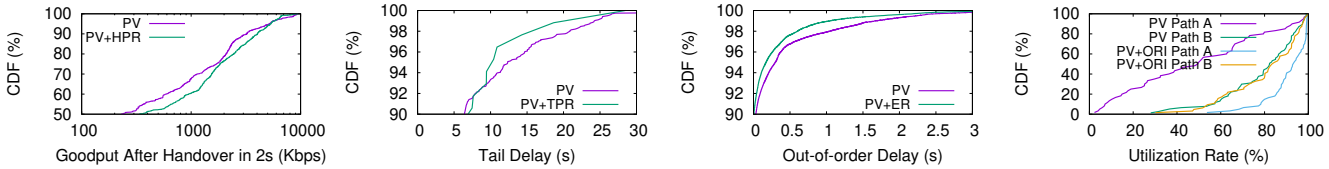
Comparison with MPTCP Schedulers. We examine the efficacy of POLYCORN by showing its goodput ratio relative to the state-of-the-art MPTCP schedulers in Fig. 10a. We make three key observations: First, POLYCORN wins in almost all the cases, demonstrating that the four data-driven scheduler designs collectively and successfully improve the networking performance under different corner cases unique to HSR that are not well handled by all the state-of-the-art MPTCP scheduling strategies. Second, minRTT (as the default MPTCP scheduler) performs similarly compared with BLEST [28] (1 MB and 4 MB), ECF [27], and STMS [18], and outperforms MuSher [17]. This indicates that the strategies assuming predictable path heterogeneity are not advantageous over the simplest (and generic) one when encountering the highly dynamic networking environment. Specifically, POLYCORN outperforms minRTT by 1.45x, 1.28x, and 1.26x for 256 KB, 1 MB, and 4 MB respectively with overall 1.31x. In general, all the state-of-the-art schedulers trust and exclusively rely on their estimations of network condition to make interface scheduling decisions accordingly, while the fluctuating network nature on HSR makes the estimations error-prone and degrade the accuracy of the scheduling decisions. This is also why MuSher performs worse than others in our case: it assigns traffic to interfaces according to the quickly varying ratio of throughput on each interface and failed to catch up with the changes; others who rely more on RTT performed better simply because RTT is relatively less variable. POLYCORN chooses to employ coarse-grained but more reliable event information and achieves better performance. Third, POLYCORN performs worse than BLEST in the shortest flow. Unlike POLYCORN that tries to improve bandwidth utilization (*i.e.,* Tail-aware Path Rejection), BLEST does not schedule packets on the path that would potentially cause head-of-line blocking, and hence achieves zero tail delay. This benefit comes at the cost of reduced bandwidth utilization, and will not continue to stay in a long flow.

Different HSR Route. We also examine the robustness of POLYCORN in different segments on the Beijing-Shanghai HSR route with different cellular coverage and terrain patterns of different channel characteristics [42]. As shown in Fig. 10b, POLYCORN consistently outperforms minRTT – the performance gain of POLYCORN on Beijing-Jinan (plain/rural), Jinan-Nanjing (hills/rural), Nanjing-Shanghai routes (urban/plain) are 19.4%, 25.2%, 44.9%, respectively.

Microbenchmark. We further study the performance gain of HSRSCH and our four individual scheduler designs over POLYCORN Vanilla (*i.e.,* POLYCORN with minRTT scheduler). We plot the goodput ratio of the aforementioned five multipath transmission schemes and POLYCORN Vanilla in Fig. 10c. The four proposals all positively improve POLYCORN's performance by 7.0%, 18.8%, 4.2%, and 2.9% on average for the three different file sizes, and they cumulatively contribute to 16% goodput gain.

• *Handover-failure-aware Path Rejection* is designed to mask the impact of packet loss during the disconnected period and the consequent RTO to the TCP (*e.g.,* slow start) by receiving or predicting handover from explicit signals from LTE real-time analytic and/or our LinkDB information and take action accordingly. Specifically, by sending redundant cross-flow copies during the period any interface encountering disconnectivity, as shown in Fig. 11a, POLYCORN can recover from the disconnection much faster, *i.e.,* achieve 1.2x and 1.5x as mean and median values within 2 seconds after handover failure, which typically lasts a few seconds or longer.

• *Tail-aware Path Rejection* is used to avoid the out-of-order delay caused by the slow paths by refusing to inject data on the interface that may increase the flow completion time. As shown in Fig. 11b, the tail delay was reduced by 5.6% on average and 15.6% in 95 percentile compared to POLYCORN



(a) HPR. (b) TPR. (c) ER. (d) ORI.

Figure 11: POLYCORN microbenchmark comparative evaluation.
(PV: Polycorn Vanilla; HPR: Handover-failure-aware Path Rejection; TPR: Tail-aware Path Rejection;
ER: Extended Reinjection; ORI: Opportunistic Redundant Traffic Injection.)

Vanilla. This mechanism is useful especially for those tail delay greater than 10 seconds, which is due to the high packet loss rate and prolonged retransmission time of the interfaces with abnormal high RTT.

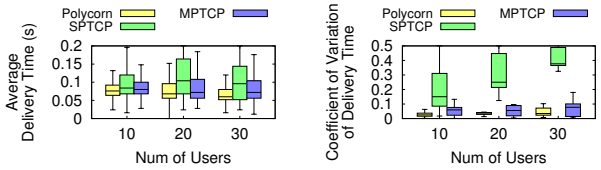
• *Extended Reinjection* mainly focuses on reducing extremely high retransmission time and leads to a significant reduction of out-of-order delay, which is shown in Fig. 11c. It reduces out-of-order delay by 23% among all the out-of-order packets, and 4% out-of-order delay among all the packets.

• *Opportunistic Redundant Traffic Injection* aims to proactively update the performance metrics of interfaces that have been idle for a while due to higher measured RTT. This helps HSRSCH more quickly discover recovered paths and improve path utilization. As shown in Fig. 11d, POLYCORN Vanilla simply ignores the path with much higher RTT. By employing opportunistic probing mechanism, the utilization rate of the path appears to be worse is increased by more than 60%, which allows better bandwidth utilization from all the paths.

Remarks. We note that POLYCORN exhibits non-trivial variation in its performance, and sometimes it falls behind the counterpart solutions. There are two major reasons: 1) It is difficult to exactly *repeat* tests on HSR: minor difference in test location results large difference in network condition – given the fluctuating network delay, the remote proxy (*i.e.,* sender) cannot accurately learn about the location of the train; 2) POLYCORN works with inaccurate handover failure predictions and TCP performance metrics. Our schedulerlets could tolerant minor errors in the context data, *e.g.,* comparative operators tolerates minor error in RTT. However, there exist cases where other solution has the proper information to make right scheduling decisions while POLYCORN does not.

6.3 Multi-user Instant Messaging Performance

We next evaluate POLYCORN in a multi-user setting, and choose instant messaging, the most popular application of HSR passengers as a representative use case for a case study. To best emulate the application behavior, we establish a long-lived TCP connection (adopted by many instant messaging application including WeChat, the most popular one in China) between POLYCORN mobile relay and server proxy for each user. We let each user sends 100 messages concurrently with pre-generated intervals following the exponential distribution. Each messaging event includes a 100-byte uplink message and an immediate 4-byte downlink one. Note that POLYCORN adopts a symmetric scheduler design for both downlink and uplink, which makes our data-driven interface scheduling work with uplink without extra effort. We perform concurrent pairwise experiments for POLYCORN vs. SPTCP and POLYCORN vs. MPTCP-minRTT respectively – SPTCP with round-robin scheduling cross different SIM card is the current operational solution used by the HSR WiFi systems due to its simplicity and interface-level fairness.



(a) Performance. (b) User fairness.

Figure 12: Multi-user instant messaging evaluation.

Experimental Results. Benefiting from the multi-stage data-driven scheduler design, POLYCORN outperforms SPTCP and MPTCP in both performance and user fairness. As shown in Fig. 12a, POLYCORN consistently improves aggregated instant messaging performance when the number of users ranges from 10 to 30. On average, POLYCORN reduces delivery time by 45% and 16% in comparison to SPTCP and MPTCP respectively, and tail delay, *e.g.,* 90 percentile, by 34% and 14%. In terms of user fairness, we use the coefficient of variation to quantify the variance of message delivery time regardless of the mean value across tests on different route segments with diverse networking conditions. As we can see in Fig. 12b, POLYCORN reduces the coefficient of variation by 86% and 49% on average when compared with SPTCP and MPTCP respectively, which significantly improves fairness across different on-board users.

7 Discussion

POLYCORN for 5G. Our evaluation does not cover 5G because the 5G CPE (Customer Premise Equipment) is not available on our HSR WiFi system yet. However, we believe POLYCORN's techniques remain applicable to 5G. For example, a recent measurement reveals that on 5G HSR, the handover failure rate is comparable to LTE [20], and multiple studies suggest that 5G suffers from higher bandwidth fluctuation and packet losses compared to 4G [20, 26].

Fairness. We discuss two fairness issues here. First, the fairness among POLYCORN users is ensured by POLYCORN's multi-user scheduling algorithm (we use proportional fair scheduling, see §4.7). Second, the fairness between POLYCORN users and non-POLYCORN users (who use their own cellular data plan) is typically guaranteed by the LTE base station. Also, POLYCORN uses unmodified TCP congestion control for each multiplexed long-lived TCP connection established for each sim card. This further minimizes the risk of POLYCORN being overly aggressive.

Other Mobile Applications. We have evaluated POLYCORN on bulk download (with different sizes) and instant messaging (with different number of users). Other applications popular on HSR include web browsing and short videos. Short video traffic may resemble bulk download that POLYCORN can effectively handle, whereas web browsing further involves client-side processing overhead, which may reduce the effectiveness of POLYCORN that only optimizes content delivery.

Scalability. The POLYCORN architecture natively supports adding more wireless interfaces (*e.g.,* SIM cards) and pairs of onboard mobile relay & in-network server proxy to meet the scalability requirement. We leave larger-scale evaluations of POLYCORN as our future work.

Head-of-Line (HoL) Blocking Issue in TCP Reuse. POLYCORN uses one multipath connection formed with TCP subflows to transmit all user traffic. The use of TCP will inevitably introduces the HoL blocking at both intra-connection and inter-user level due to due to its byte-level ordering guarantee, which is an overkill in POLYCORN's multi-user multiplexing context. We envision that this problem will be solved by (MP)QUIC with its support of out-of-order delivery and cross-path acknowledgment [43, 44].

8 Related Work

Mobile Networking Performance Improvement. A plethora of research efforts have been devoted to improve network performance (under high mobility) through developing robust handover schemes [26], simplifying cellular control plane [45, 46], and fixing base station-side policy configuration bugs [47, 48]. Unlike POLYCORN, all the above approaches require modifications to the cellular infrastructure. There are also studies at upper layers, *e.g.,* designing customized single-path transport protocol [49] and optimizing congestion control algorithms [50–54]. POLYCORN instead proposes a holistic multipath solution with new optimization dimensions.

Performance-enhancement Proxy (PEP). In vehicular systems, PEPs are often deployed on mobile relays, to leverage carrier diversity and UDP encapsulation for bandwidth aggregation and mitigating link failure, *e.g.,* through striping [11, 55], opportunistic erasure coding [12], and flow splicing [13]. Specifically, the work [35, 55] present the idea of location-aware link characteristics (*e.g.,* throughput and availability) prediction and packet scheduling. PEPs can also be deployed in fixed locations in the Internet [56–58]. POLYCORN synthesizes all the ideas above and presents four multipath scheduling strategies dedicated to addressing the unique networking challenges in extreme mobility.

Multipath Transport Architecture. Transmitting data over multiple paths can be realized at different layers, *e.g.,* WNIC driver [59, 60], in-kernel transport layer [36, 61], light kernel modification [21, 22], and UDP encapsulation [62–64]. Differing from the above, POLYCORN is an entire userspace solution reusing Linux TCP for the benefits of OS/middlebox compatibility, application transparency, and good runtime performance, with multipath, multi-user multiplexing support.

Scheduling over Heterogeneous Paths. Several generic multipath transport schedulers have been proposed to mitigate the head-of-line blocking and out-of-order delay incurred by imbalanced subflows, such as opportunistic declining [28] and migrating [27], intra-chunk opposite scheduling [65], out-of-order transmission [18], and reactive bandwidth probing [17]. None of them considers HSR-specific aspects, and many of them [27, 28, 65] only work for two paths. Horde [66] and miDRR [67] allows user/app to specify their QoS requirement and perform packet scheduling accordingly. RAVEN [14] achieves low latency by extensively leveraging redundant transmission over multiple paths. HSRSCH brings new optimization dimensions integrated through a composable schedulerlet pipeline, and strikes a balance where overall throughput is not harmed by large amount of redundant traffic and latency of short flows are preserved.

9 Conclusion

The popularity of HSR systems brings the requirement of high-performance data networking under extreme mobility more tangible than ever. In this work, we have addressed the challenge of bringing seamless Internet service to passengers on HSR by synthesizing multipath transmission and data-driven scheduling techniques into a practical and readily deployable system design. Extensive experimental results have demonstrated the effectiveness of our system design dedicated to extreme-mobility. We believe that our upper layer optimization solution can seamlessly cooperate with the ongoing 5G/NextG(-Unlicensed) evolution.

Acknowledgment

We are grateful to the reviewers for their constructive critique, and our shepherd Keith Winstein in particular, for his valuable comments, all of which have helped us greatly improve this paper. We also thank Dina Katabi, Songwu Lu, Kun Tan and Yong Cui for their thoughtful input based on an early version of the work. This work was supported by National Key Research and Development Plan, China (Grant No. 2020YFB1710900), National Natural Science Foundation of China (Grant No. 62022005 and 62172008) and Microsoft Research Asia. Chenren Xu is the corresponding author.

References

[1] China launches upgraded high-speed trains, with wi-fi. https://gbtimes.com/china-launches-upgraded-high-speed-trains-wi-fi.

[2] Jr to launch free wi-fi on bullet trains from may. https://mainichi.jp/english/articles/20180303/p2a/00m/0na/009000c.

[3] South korea's brand-new olymic bullet train will make americans jealous. https://mic.com/articles/187809/south-koreas-brand-new-olympic-bullet-train-will-make-americans-jealous.

[4] Spain's high speed trains introduce high speed wifi. https://www.thelocal.es/20161104/spains-high-speed-trains-introduce-high-speed-wifi.

[5] Eurostar - on-board entertainment server launched. https://nomad-digital.com/customer-story/eurostar-on-board-entertainment-server-launched.

[6] Enjoy the standard experience. https://www.thalys.com/nl/en/info-services/enjoy-the-standard-experience.

[7] Deutsche bahn launches 'wifi @ db' wlan network. https://www.globalrailwayreview.com/news/109948/deutsche-bahn-launches-wifi-db-wlan-network.

[8] Jing Wang, Yufan Zheng, Yunzhe Ni, Chenren Xu, Feng Qian, Wangyang Li, Wantong Jiang, Yihua Cheng, Zhuo Cheng, Yuanjie Li, Xie Xiufeng, Yi Sun, and Zhongfeng Wang. An active-passive measurement study of tcp performance over lte on high-speed rails. In *ACM MobiCom*, 2019.

[9] Chenren Xu, Jing Wang, Zhiyao Ma, Yihua Cheng, Yunzhe Ni, Wangyang Li, Feng Qian, and Yuanjie Li. A first look at disconnection-centric tcp performance on high-speed railways. *IEEE Journal on Selected Areas in Communications*, 38(12), 2020.

[10] Multipath tcp - linux kernel implementation. https://multipath-tcp.org.

[11] Pablo Rodriguez, Rajiv Chakravorty, Julian Chesterfield, Ian Pratt, and Suman Banerjee. Mar: A commuter router infrastructure for the mobile internet. In *ACM MobiSys*, 2004.

[12] Ratul Mahajan Jitendra Padhye Sharad Agarwal and Brian Zill. High performance vehicular connectivity with opportunistic erasure coding. In *USENIX ATC*, 2012.

[13] Joshua Hare, Lance Hartung, and Suman Banerjee. Transparent flow migration through splicing for multi-homed vehicular internet gateways. In *IEEE VNC*, 2013.

[14] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. Raven: Improving interactive latency for the connected car. In *ACM MobiCom*, 2018.

[15] Li Li, Ke Xu, Tong Li, Kai Zheng, Chunyi Peng, Dan Wang, Xiangxiang Wang, Meng Shen, and Rashid Mijumbi. A measurement study on multi-path tcp with multiple cellular carriers on high speed rails. In *ACM SIGCOMM*, 2018.

[16] Qingyang Xiao, Ke Xu, Dan Wang, Li Li, and Yifeng Zhong. Tcp performance over mobile networks in high-speed mobility scenarios. In *IEEE ICNP*, 2014.

[17] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. Musher: An agile multipath-tcp scheduler for dual-band 802.11 ad/ac wireless lans. In *ACM MobiCom*, 2019.

[18] Hang Shi, Yong Cui, Xin Wang, Yuming Hu, Minglong Dai, Fanzhao Wang, and Kai Zheng. Stms: Improving mptcp throughput under heterogeneous networks. In *USENIX ATC*, 2018.

[19] Li Li, Ke Xu, Dan Wang, Chunyi Peng, Kai Zheng, Rashid Mijumbi, and Qingyang Xiao. A longitudinal measurement study of tcp performance and behavior in 3g/4g networks over high speed rails. *IEEE/ACM Transactions on Networking*, 25(4), 2017.

[20] Yueyang Pan, Ruihan Li, and Chenren Xu. The first 5g-lte comparative study in extreme mobility. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1), 2022.

[21] Feng Qian, Vijay Gopalakrishnan, Emir Halepovic, Subhabrata Sen, and Oliver Spatscheck. Tm 3: flexible transport-layer multi-pipe multiplexing middlebox without head-of-line blocking. In *ACM CoNEXT*, 2015.

[22] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z Morley Mao, and Subhabrata Sen. An in-depth understanding of multipath tcp on mobile devices: measurement and system design. In *ACM MobiCom*, 2016.

[23] Fumihiro Hasegawa, Akinori Taira, Gosan Noh, Bing Hui, Hiroshi Nishimoto, Akihiro Okazaki, Atsushi Okamura, Junhwan Lee, and Ilgyu Kim. High-speed train communications standardization in 3gpp 5g nr. *IEEE Communications Standards Magazine*, 2(1), 2018.

[24] Bo Ai, Andreas F Molisch, Markus Rupp, and Zhang-Dui Zhong. 5g key technologies for smart railways. *Proceedings of the IEEE*, 108(6), 2020.

[25] Study on international mobile telecommunications (imt) parameters for 6.425 - 7.025 ghz, 7.025 - 7.125 ghz and 10.0 - 10.5 ghz. https://www.3gpp.org/DynaReport/38921.htm.

[26] Yuanjie Li, Qianru Li, Zhehui Zhang, Ghufran Baig, Lili Qiu, and Songwu Lu. Beyond 5g: Reliable extreme mobility management. In *ACM SIGCOMM*, 2020.

[27] Yeon-sup Lim, Erich M Nahum, Don Towsley, and Richard J Gibbens. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *ACM CoNEXT*, 2017.

[28] Simone Ferlin, Özgü Alay, Olivier Mehani, and Roksana Boreli. Blest: Blocking estimation-based mptcp scheduler for heterogeneous networks. In *IFIP Networking*, 2016.

[29] China's high-speed rail links winter olympics cities. http://english.cctv.com/2019/12/30/ARTIITvoMUF29MZmtX5y4t9m191230.shtml.

[30] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The tcp/udp bandwidth measurement tool. http://dast.nlanr.net/Projects.

[31] Gerald Combs. Tshark-the wireshark network analyser. http://www.wireshark.org.

[32] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, et al. Bbr: congestion-based congestion control. *Communications of the ACM*, 60(2), 2017.

[33] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5), 2008.

[34] Yuanjie Li, Chunyi Peng, Zengwen Yuan, Jiayao Li, Haotian Deng, and Tao Wang. Mobileinsight: Extracting and analyzing cellular network information on smartphones. In *ACM MobiCom*, 2016.

[35] Jun Yao, Salil S Kanhere, and Mahbub Hassan. Improving qos in high-speed mobility using bandwidth maps. *IEEE Transactions on Mobile Computing*, 11(4), 2011.

[36] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *USENIX NSDI*, 2012.

[37] Raymond Kwan, Cyril Leung, and Jie Zhang. Proportional fair multiuser scheduling in lte. *IEEE Signal Processing Letters*, 16(6), 2009.

[38] The netfilter.org project. https://www.netfilter.org/.

[39] Linux Foundation. Data plane development kit (DPDK), 2015.

[40] The netfilter.org "iptables" project. https://www.netfilter.org/projects/iptables/index.html.

[41] Korian Edeline, Mirja Kühlewind, Brian Trammell, and Benoit Donnet. copycat: Testing differential treatment of new transport protocols in the wild. In *ACM ANRW*, 2017.

[42] Cheng-Xiang Wang, Ammar Ghazal, Bo Ai, Yu Liu, and Pingzhi Fan. Channel measurements and models for high-speed train communication systems: A survey. *IEEE communications surveys & tutorials*, 18(2), 2015.

[43] J Iyengar and M Thomson. Rfc 9000 quic: A udp-based multiplexed and secure transport. *Omtermet Emgomeeromg Task Force*, 2021.

[44] https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/.

[45] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A high performance packet core for next generation cellular networks. In *ACM SIGCOMM*, 2017.

[46] Yuanjie Li, Zengwen Yuan, and Chunyi Peng. A control-plane perspective on reducing data access latency in lte networks. In *ACM MobiCom*, 2017.

[47] Yuanjie Li, Haotian Deng, Jiayao Li, Chunyi Peng, and Songwu Lu. Instability in distributed mobility management: Revisiting configuration management in 3g/4g mobile networks. In *ACM SIGMETRICS*, 2016.

[48] Zengwen Yuan, Qianru Li, Yuanjie Li, Songwu Lu, Chunyi Peng, and George Varghese. Resolving policy conflicts in multi-carrier cellular access. In *ACM MobiCom*, 2018.

[49] Hongke Zhang, Wei Quan, Jiayang Song, Zhongbai Jiang, and Shui Yu. Link state prediction-based reliable transmission for high-speed railway networks. *IEEE Transactions on Vehicular Technology*, 65(12), 2016.

[50] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM*, 2015.

[51] Wai Kay Leong, Zixiao Wang, and Ben Leong. Tcp congestion control beyond bandwidth-delay product for mobile cellular networks. In *ACM CoNEXT*, 2017.

[52] Shinik Park, Jinsung Lee, Junseon Kim, Jihoon Lee, Sangtae Ha, and Kyunghan Lee. Exll: An extremely low-latency congestion control for mobile cellular networks. In *ACM CoNEXT*, 2018.

[53] Ke Liu, Zhongbin Zha, Wenkai Wan, Vaneet Aggarwal, Binzhang Fu, and Mingyu Chen. Optimizing tcp loss recovery performance over mobile data networks. *IEEE Transactions on Mobile Computing*, 19(6), 2019.

[54] Soheil Abbasloo, Yang Xu, and H Jonathan Chao. C2tcp: A flexible cellular tcp to meet stringent delay requirements. *IEEE Journal on Selected Areas in Communications*, 37(4), 2019.

[55] Joshua Hare, Lance Hartung, and Suman Banerjee. Beyond deployments and testbeds: experiences with public usage on vehicular wifi hotspots. In *ACM MobiSys*, 2012.

[56] Rajiv Chakravorty, Sachin Katti, Ian Pratt, and Jon Crowcroft. Using tcp flow-aggregation to enhance data experience of cellular wireless users. *IEEE Journal on Selected Areas in Communications*, 23(6), 2005.

[57] Kyu-Han Kim and Kang G Shin. Prism: Improving the performance of inverse-multiplexed tcp in wireless networks. *IEEE Transactions on Mobile Computing*, 6(12), 2007.

[58] Jiasi Chen, Rajesh Mahindra, Mohammad Amir Khojastepour, Sampath Rangarajan, and Mung Chiang. A scheduling framework for adaptive video delivery over cellular networks. In *ACM MobiCom*, 2013.

[59] Srikanth Kandula, Kate Ching-Ju Lin, Tural Badirkhanli, and Dina Katabi. Fatvap: Aggregating ap backhaul capacity to maximize throughput. In *USENIX NSDI*, 2008.

[60] Anthony J Nicholson, Scott Wolchok, and Brian D Noble. Juggler: Virtual networks for fun and profit. *IEEE Transactions on Mobile Computing*, 9(1), 2010.

[61] Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. A programming model for application-defined multipath tcp scheduling. In *ACM Middleware*, 2017.

[62] Luiz Magalhaes and Robin Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *IEEE ICNP*, 2001.

[63] Cheng-Lin Tsao and Raghupathy Sivakumar. On effectively exploiting multiple wireless interfaces in mobile hosts. In *ACM CoNEXT*, 2009.

[64] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *ACM CoNEXT*, 2017.

[65] Yihua Ethan Guo, Ashkan Nikravesh, Z Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating multipath transport through balanced subflow completion. In *ACM MobiCom*, 2017.

[66] Asfandyar Qureshi and John Guttag. Horde: separating network striping policy from mechanism. In *ACM MobiSys*, 2005.

[67] Kok-Kiong Yap, Te-Yuan Huang, Yiannis Yiakoumis, Sandeep Chinchali, Nick McKeown, and Sachin Katti. Scheduling packets over multiple interfaces while respecting user preferences. In *ACM CoNext*, 2013.

A Example for Tail-aware Path Rejection

Fig. 13 exemplifies how tail-aware path rejection works on HSR. In this example, the traffic consists of a flow whose FIN packet was received by POLYCORN (so all the subsequent packets are tail packets). There are two paths A and B, whose estimated RTT and send buffer occupancy levels are plotted in the top and bottom subfigure, respectively. Path A has a low RTT and its send buffer is almost full, whereas Path B has a high RTT and its buffer occupancy level is low. MPTCP's default minRTT scheduler frequently schedules tail packets to Path B because Path A is busy (congestion window being full). However, our algorithm usually rejects Path B because the flow completion time will reduce if we wait for Path A to become available and send tail packets over A. This results in a large number of path rejection instances.

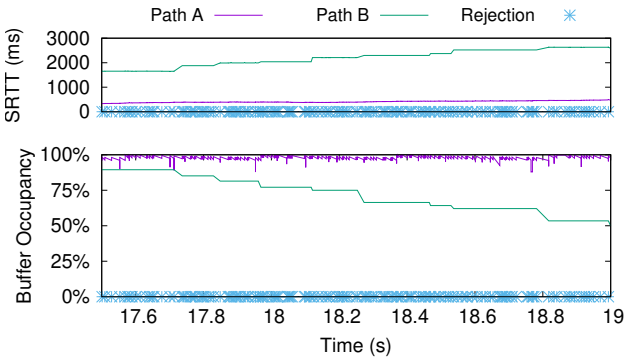


Figure 13: Reject Trace.

Augmenting Augmented Reality with Non-Line-of-Sight Perception

Tara Boroushaki [1], Maisy Lam [1], Laura Dodds [1], Aline Eid [1,2], Fadel Adib [1]
[1] *Massachusetts Institute of Technology,* [2] *University of Michigan*
tarab@mit.edu, mllam@mit.edu, ldodds@mit.edu, alineeid@umich.edu, fadel@mit.edu

Abstract – We present the design, implementation, and evaluation of X-AR, an augmented reality (AR) system with non-line-of-sight perception. X-AR augments AR headsets with RF sensing to enable users to see things that are otherwise invisible to the human eye or to state-of-the-art AR systems. Our design introduces three main innovations: the first is an AR-conformal antenna that tightly matches the shape of the AR headset visor while providing excellent radiation and bandwidth capabilities for RF sensing. The second is an RF-visual synthetic aperture localization algorithm that leverages natural human mobility to localize RF-tagged objects in line-of-sight and non-line-of-sight settings. Finally, the third is an RF-visual verification primitive that fuses RF and vision to deliver actionable tasks to end users such as picking verification. We built an end-to-end prototype of our design by integrating it into a Microsoft Hololens 2 AR headset and evaluated it in line-of-sight and non-line-of-sight environments. Our results demonstrate that X-AR achieves decimeter-level RF localization (median of 9.8 cm) of fully-occluded items and can perform RF-visual picking verification with over 95% accuracy (F-Score) when extracting RFID-tagged items. These results show that X-AR is successful in extending AR systems to non-line-of-sight perception, with important implications to manufacturing, warehousing, and smart home applications. Demo video: y2u.be/bdUN21ft7G0

1 Introduction

The past few years have witnessed an increasing interest in augmented reality (AR) systems. Major tech companies - including Microsoft, Meta, Apple, and Google - have invested billions of dollars in developing AR technologies [7, 25, 52, 51]. A significant driver for these investments is the role that AR systems are expected to play in boosting efficiency across Industry 4.0 sectors including manufacturing, warehousing, logistics, and retail. For example, in e-commerce warehouses, AR headsets can boost labor efficiency by guiding workers in picking, sorting, and packing orders and returns [26]. Similarly, in manufacturing settings, AR headsets can guide employees by visualizing assembly tasks, automatically labeling tools in the environment, and helping users find parts they need [28]. More generally, AR headsets are expected to make workers more efficient by annotating their environments, visualizing their next tasks, and guiding them in executing these tasks [27, 30].

To realize their full potential, AR headsets need to deliver the above capabilities in real-world industrial environments, which are typically dense and highly cluttered. For example, a typical warehouse or dark store is dense with packages, and a standard manufacturing plant is dense with materials and compartments. In these environments, the majority of items are occluded due to being inside a box, under a pile, or behind other packages. Such occlusions make it difficult for existing headsets to perceive these items, which in turn prevents them from identifying and locating the items or guiding workers towards them. This limitation stems from the fact that today's AR headsets perceive their environment through cameras or other vision-based sensing systems which are inherently limited to line-of-sight (LOS) [6, 38]. Such line-of-sight restriction hinders AR systems from boosting worker efficiency where it is most needed, namely in cluttered and dense industrial environments.

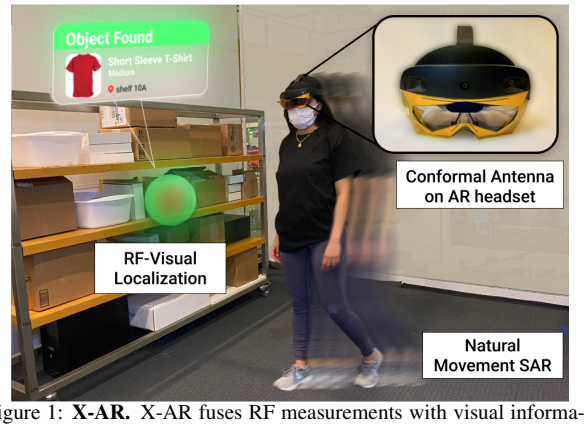


Figure 1: **X-AR.** X-AR fuses RF measurements with visual information and leverages natural human motion to localize RFID tagged items in the environment. The system uses a custom-designed, conformal, light-weight antenna mounted on an augmented reality headset and displays information to the user.

In this paper, we ask the following question: *Can we design and build an augmented reality system that can sense fully-occluded objects and expand the perception of humans beyond the line of sight?* With this capability, augmented reality would go beyond any natural human ability and truly augment the way we interact with the world, enabling significant advances in warehouse logistics, manufacturing, retail, and more. For example, AR headsets with non-line-of-sight (NLOS) perception could identify and localize specific items (e.g., customer orders, tools, materials) even when they are fully occluded, helping workers avoid a lengthy search process. Additionally, such AR headsets could be used to automate inventory control of items in warehouses or retail stores without needing to see all objects, and can alert

workers to misplaced items hidden behind occlusions.

To realize this vision, our approach is to leverage Radio Frequency (RF) signals, which, unlike visible light, can traverse everyday occlusions such as cardboard boxes, plastic containers, wooden dividers, and clothing fabric. Indeed, recent advances in RF sensing have demonstrated the potential to use RF signals to sense and accurately localize items in non-line-of-sight and highly cluttered environments [17, 40, 39]. Among existing RF sensing technologies, we are particularly interested in leveraging UHF RFID (Radio Frequency IDentification) tags due to their widespread adoption in supply chain industries (for example, over 93% of US retailers have adopted UHF RFIDs [5]). Our vision is to bring RFID sensing and localization to AR headsets to provide them with non-line-of-sight perception and augment human visual abilities for applications in warehouse automation, e-commerce fulfillment, and manufacturing.

We would like to build a system that realizes the above vision while satisfying the following requirements:

1. *AR-compatibility:* The system must seamlessly integrate with an AR headset without impacting the performance of its existing sensors and displays (i.e., without obstructing the headset cameras or the user field of view).
2. *Seamless Mobility:* The system must operate correctly with natural human mobility. Specifically, it must be able to accurately localize RFIDs (in LOS and NLOS settings) without requiring the user to perform unnatural movement patterns, which may hinder their productivity.
3. *Actionability:* The system should provide users with actionable tasks (e.g., guide the user where to search) and inform users of task success (e.g., verify to a warehouse picker whether or not they picked the right order).
4. *User-friendliness:* The system needs to be compact and lightweight, so that the user can easily wear the AR headset and move around to complete their tasks.

Satisfying the above requirements is challenging and cannot be realized by simply integrating a state-of-the-art RFID localization system with an AR headset. In particular, the majority of accurate RFID localization solutions require multiple antennas that are separated by meter-scale distances [39, 40], making them too bulky to mount onto an AR headset. Solutions that don't require such antenna arrays typically rely on robot-mounted antennas that need to be moved on predefined trajectories [58, 17, 15], making them incompatible with natural human mobility. In addition to these challenges, delivering AR-actionable tasks goes beyond simple RF localization and requires new mechanisms to fuse RF and vision perception under natural mobility and display the output on the headset.

In this paper, we present X-AR, an augmented reality headset with a built-in RF sensing system. A user wearing X-AR can freely walk in their environment (e.g., a warehouse or manufacturing plant), and the headset automatically identifies and localizes items in the environment, even when they are not the in line-of-sight. Using this information, X-AR guides the worker towards items of interest (tools, packages, etc.) and verifies whether or not they have picked up the correct item. Our system introduces multiple innovations that together allow it to satisfy the above requirements:

1. AR-Conformal Wideband Antenna: X-AR introduces the design of an ultra-lightweight and wideband antenna that is conformal to the headset (described in §3). Our unique antenna design matches the shape of the AR glasses visor, as depicted in Fig. 1, and does not block the user's view or any sensors. The antenna also achieves the radiation, bandwidth (BW), and gain properties required to perform accurate RFID localization.

2. RF-Visual Synthetic Aperture Radar: X-AR does not make restrictive assumptions about the user's motion pattern when localizing the RFID tags in the environment, and opportunistically leverages natural human mobility. To do this, X-AR first uses the visual information from the AR headset camera to self-localize in the environment. It then uses the RFID measurements collected during the user's motion to create a synthetic aperture radar (SAR) and localize RFID tagged items with high accuracy. In addition, X-AR introduces a number of techniques to handle localization artifacts and constraints that arise from natural human motions such as natural head tilts and RFID backscatter radiation properties. We describe this localization method in detail in §4 and show how the system guides the user to the item's location and displays it on the AR headset.

3. RF-Visual Verification: The final component of X-AR's design is a mechanism that verifies when the user has picked up their desired RFID-tagged item. Such verification is important to avoid costly errors such as picking and shipping the wrong order to a customer in e-commerce warehouses. One might assume that such a capability can be simply realized by localizing the RFID-tagged target item to within a user's hand once they've picked it up (i.e., using the same localization mechanism described up). In practice, doing so is challenging because, unlike the above scenario where the user's walking emulates a synthetic aperture, a user picking an item stays in a relatively fixed location. To address this challenge, X-AR leverages the RFID tag's mobility instead. Specifically, it performs a *reverse* SAR to localize the headset with respect to the picked item's trajectory. In §5, we describe this technique in detail and show how X-AR fuses the AR-headset's hand-tracking capability with reverse SAR to perform the verification with high accuracy.

We implemented an end-to-end prototype of X-AR. We mounted a custom-designed conformal antenna on

Microsoft Hololens 2 headset [9]. The antenna is connected to bladeRF software defined radios [47] that communicate with the AR headset through an edge server. Our algorithms are written in the C driver and operate in real-time, and we program the Hololens through Unity to display item locations and labels, guide the user to the target items, and show the verification results.

We evaluated X-AR's performance over 230 experimental trials. Our evaluation demonstrated that:

1. X-AR's conformal antenna achieves all desired specifications in terms of weight ($<$1g), size (conformal), BW (200 MHz), and gain (around 0 dB).

2. X-AR accurately localizes RFID tagged objects in line of sight and non line of sight scenarios with a median accuracy of 9.8 cm. Even the 90th percentile accuracy remains within a foot and a half (45 cm). In contrast, a standard SAR-based baseline has more than double the error, achieving a median accuracy of 24.8 cm and a 90th percentile accuracy of 99.1 cm.

3. X-AR tracks the movement of the RFID tag and the user's hand to automatically verify what object has been picked up with over 95% accuracy (F-Score). Even if the user picks up a box with the RFID-tagged item inside it (rather than picking up the item itself), the F-Score remains over 91%.

Contributions: X-AR is the first augmented reality headset that can sense through occlusions and perceive fully occluded objects. This system introduces three key innovations: 1) A custom-designed, conformal, wideband, and light-weight antenna that can be integrated with a commercial AR headset, enabling RFID localization without obstructing the user's or cameras' view, 2) An RFID localization system that opportunistically leverages natural user motion to create a non uniform RF-Visual synthetic aperture radar and to localize and visualize the RFID tagged objects in 3D, 3) A verification mechanism that performs reverse RF-Visual localization to verify whether the user has picked the target item, in line-of-sight, non-line-of-sight, or occluded settings.

Although X-AR enables non-line-of-sight perception for augmented reality headsets, our current implementation still has a few limitations. First, X-AR is currently designed to operate on a single headset, and still has no mechanisms to extend to multiple coordinated headsets. Second, the range of the RF measurements are limited to 3m; however, future antenna design iterations can achieve an even longer range. Finally, X-AR only demonstrates two actionable tasks: guiding the user toward a target item, and verifying the target item is in the user's hand. As research evolves, it would be interesting to extend this system to other tasks. Despite these limitations, X-AR marks an important step in bringing RF sensing to AR and opens the door to future works bridging these fields.

2 System Overview

X-AR is a next-generation augmented reality system capable of perceiving objects in both LOS and NLOS conditions. The system can identify, locate, and label RFID-tagged items in the environment. It leverages an RF sensing module to read passive, off-the-shelf UHF RFID tags attached to items of interest. By combining this information with visual data from the AR headset's camera sensors, it locates RFID-tagged items with high accuracy.

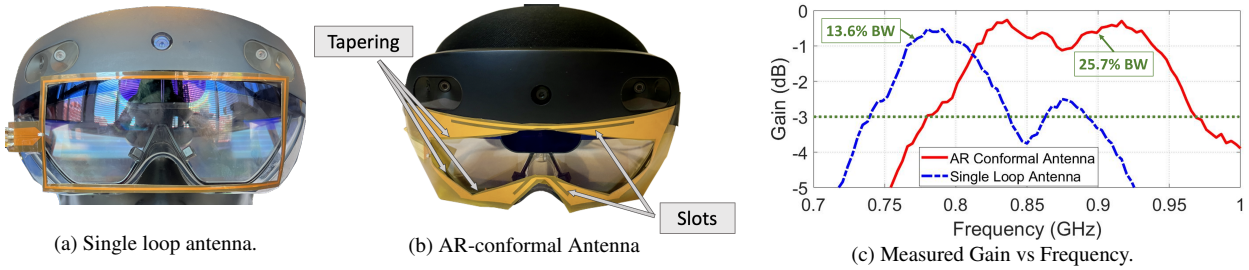
X-AR is designed to be used in practical environments, such as warehouses, manufacturing plants, and retail stores. It opportunistically leverages human motion (i.e., as the user walks around and picks up items) in order to localize tagged items in the environment, guide the user towards them, and verify when the user has picked them up. For simplicity, the remainder of this paper discusses the system in a single tag scenario. sHowever, X-AR can easily extend to multiple RFID tags in the environment. Using the EPC Gen 2 protocol, X-AR can read each RFID tag separately, and perform the same localization and verification algorithms for each tag.

3 AR-Conformal Antenna

X-AR introduces a conformal antenna that can be mounted on the headset to identify, locate, and verify UHF RFID tags, without interfering with the headset's operation or constraining the user. Here, we describe our AR-conformal antenna design, its requirements, challenges, and the path describing its evolution from a conventional antenna structure to one satisfying all the desired needs. To perform RFID localization from the headset in the field of view of the user, the antenna needs to satisfy the following requirements:

- **Wideband operation around 900 MHz:** The antenna needs to maintain a matched operation and a good gain over a BW of at least 200 MHz to match the bandwidth requirements of state-of-the-art RFID localization systems [39, 40].
- **Conformal and unobstructive:** The antenna must be designed on a flexible substrate to easily conform to the Hololens' visor without obstructing a user's field of view or the cameras mounted on the front of the headset.
- **Lightweight and small form-factor:** The antenna must maintain an ultra-light weight ($<$ 1g) and be simple and easy to mount on the AR's visor.

Existing solutions in state-of-the-art wideband RFID localization systems do not satisfy these properties [16, 17, 40, 39, 14]. In particular, they rely on rigid, relatively-large, and often bulky antennas. For example, the majority of these systems leverage large patch antennas that are 26 $cm \times 26$ $cm \times 3.3$ cm and weigh approximately 1.04 kg, while others rely on log-periodic antennas that measure 15 $cm \times 13$ $cm \times 0.01$ cm. These solutions are too bulky and would obstruct the field of



(a) Single loop antenna. (b) AR-conformal Antenna (c) Measured Gain vs Frequency.

Figure 2: **Conformal Antenna Design.** (a) Fabricated single-loop antenna mounted on the headset (dimensions 122×51mm). (b) Fabricated conformal antenna (dimensions 165×64mm). (c) These plots show the measured gains of the single loop (blue) and AR conformal (red) antennas vs frequency while mounted on the headset and worn by the user. The horizontal green line is used to highlight the 3-dB bandwidth.

view of the AR headset, thus are ill-suited for our use case. While some RFID localization systems utilize compact and lightweight antennas [58, 57, 59], these antennas have a narrow band of operation, which makes them unsuitable for our use case.[1]

Below, we describe our investigation in designing the AR-conformal antenna to satisfy the above requirements.

3.1 Investigating a Single Loop Design

We first investigated whether we could achieve the above properties using a single loop antenna design. Our choice of a single loop was motivated by the fact that a loop can wrap around the outline of the visor, delivering a small form factor not obstructing the field of view. Also, the loop is a simple antenna that does not require a ground plane, making it easy to mount on an AR headset.

Fig. 2a shows the picture of our initial design of a loop antenna, fabricated on a 100 µm thin polyimide substrate, and mounted on the Hololens. Notice how our antenna (almost) follows the perimeter of the visor, thus not obstructing its view. In order to identify the appropriate dimensions corresponding to an operation around 900 MHz, we performed our antenna simulations in Ansys High Frequency Simulation Software (HFSS). In designing these antennas, we leveraged polyimide films because of their good electromagnetic properties, their common use for applications requiring flexible electronics, and their wide availability at low-cost. This antenna also weighs less than 1g, thus it satisfies our requirements of a small form factor while maintaining a light weight.

To investigate the bandwidth requirement, we mounted the antenna on the headset, worn by the user, and measured its gain over the frequency of interest. This was done by illuminating it with a transmitter antenna of a known gain and using a vector network analyzer (VNA) to extract the S parameters of the loop antenna (specifically the S21 parameter).

Fig. 2c plots the gain of the loop with respect to frequency, showing 3 dB BW of approximately 100 MHz around 780 MHz. This shows the loop antenna design would not allow us to achieve the desired 200 MHz of BW. It should be noted the loop antenna delivers a resonant frequency of 900 MHz and a gain of 3.8 dB when tested in air. However, its gain degraded by 3 dB and frequency detuned by 120 MHz when placed on the headset visor and worn by the user. This behavior is often observed with wearable antennas [32, 41], where the frequency of operation and antenna radiation properties degrade when mounted on a new material. Thus, while the loop antenna is conformal, unobstructive, lightweight, and small, it did not satisfy the BW requirements.

[1] As mentioned in §1, past systems that leverage these antennas require either bulky arrays or a robot to move the antenna on a pre-defined trajectory, neither of which are suitable for an AR localization system.

3.2 Wideband AR-Conformal Antenna

Motivated by a desire to increase the bandwidth of the single-loop conformal antenna, we investigated how strategies such as tapering (i.e., gradually changing the width of the loop) and slotting (i.e., adding slotted gaps in the loop) can help us achieve the desired bandwidth. Through an iterative design and simulation processing (whereby the simulation was performed using Ansys HFSS), we reached the design shown in Fig. 2b. Notice how we carefully chose the dimensions of the antenna to perfectly match the shape of the visor, without blocking any of the cameras. We also added tapers to the outline of the antenna and integrated slots on the top and bottom lines around the nose to achieve a wideband operation.

Similar to the loop antenna, we conducted gain measurements to assess the 3-dB bandwidth of our conformal antenna while mounted on the headset and worn by the user. The red plot in Fig. 2c shows the gain of our new antenna as a function of frequency. Notice how the 3 dB bandwidth of the gain is now 200 MHz - from 775 MHz to 975 MHz. This shows that the antenna achieves the desired gain pattern in the frequency range of interest. Note that the negative gain realized by these wearable antennas is normal with ultra-thin substrates due to close proximity with lossy material such as the headset and human tissues [48, 19]. In principle, it is possible to further optimize the gain of the antenna, however, the negative gain could be easily overcome by transmitting at a higher power, thus maintaining a constant effective radiation pattern (typically referred to as EIRP). It should be also noted that the detuned frequency observed with

the measured loop due to placement on headset was accounted for in the HFSS simulations for the new antenna, by simulating the structure on plexiglass that mimics the headset's visor, and thus resulted in the proper resonant frequency during measurements. Finally, we also simulated the radiation pattern of the conformal AR antenna on the headset as well as measured its gains across frequencies and elevation angles (see appendix).

It should be noted that while this antenna was designed to match this headset, the design could be adapted for different visor shapes, depending on the location of the cameras and other components that cannot be blocked.

4 RF-Visual Synthetic Aperture Radar

In the previous section, we described the custom, conformal, and lightweight antenna that X-AR uses to sense RFID tags in the environment. In this section, we describe how X-AR uses these RFID measurements, along with visual information from the AR headset's camera to locate RFID tags with high accuracy through RF-Visual Synthetic Aperture Radar (SAR). For ease of exposition, we discuss localizing a single tag, but the same approach generalizes to any number of tags in the environment.

4.1 Background on SAR

X-AR's localization builds on a technique called Synthetic Aperture Radar (SAR). At a high level, SAR leverages the same localization principle as antenna arrays, where measurements from multiple antenna locations are combined to localize a wireless device in 2D or 3D space. SAR differs from standard antenna arrays in that it moves a single antenna, collecting measurements from different physical locations to emulate an antenna array. Formally, we can estimate the power P received from every point in space using the following equation:

$$P(x,y,z) = \left|\left| \frac{1}{M}\frac{1}{N}\sum_{j=1}^{M}\sum_{i=1}^{N} h_{i,j} e^{\frac{4\pi d_i(x,y,z)}{\lambda_j}} \right|\right| \quad (1)$$

where M is the number of frequencies used, $h_{i,j}$ is the channel measurement of the ith location with the jth frequency, d_i is the distance from (x,y,z) to the ith location, and λ_j is the wavelength of jth frequency.

To localize the tag, we find the (x,y,z) location with the highest power. Formally, the location of the tag, p_{tag}:

$$p_{tag} = \texttt{argmax}_{(x,y,z)}(P(x,y,z)) \quad (2)$$

For more details on SAR please refer to the Appendix.

4.2 AR-Based SAR

Since it is infeasible to mount an antenna array on an AR headset, X-AR builds on SAR-based RFID localization. Specifically, X-AR opportunistically leverages natural human motion to collect wideband measurements from different locations and uses them to construct a synthetic aperture radar to localize RFID tagged items.

However, bringing SAR to an AR headset faces a number of challenges. Unlike prior systems that leverage SAR (e.g., robots or airplanes), X-AR cannot rely on a constant velocity or predictable trajectory. For example, humans naturally accelerate and decelerate and move slightly side-to-side as they walk, making it difficult to predict the exact antenna location. Moreover, recall that X-AR aims to opportunistically leverage human motion as opposed to controlling the user's trajectory, making it even more challenging to control the antenna's location.

Self-Tracking. To address these challenges and localize the antenna over time, X-AR leverages the AR headset's built-in self-tracking capability. Existing AR headsets can self localize by extracting feature points from their cameras' visual data and performing visual-inertial odometry (VIO). They then track these points over time to build a map of the environment and derive their 6D pose (i.e., location and rotation) within this map [38, 42].

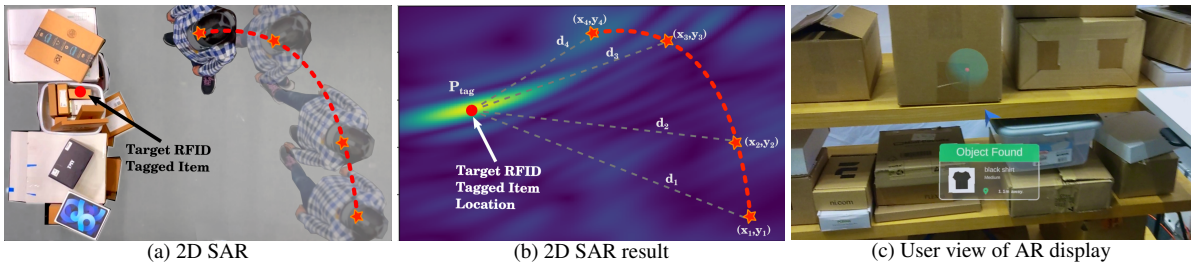
To leverage this built-in localization, X-AR requires an additional transformation. Specifically, the headset tracks its location as the center of the user's head, but the antenna is mounted on the front of the visor. This transform is essential since SAR relies on small changes in the RFID channel and therefore requires precise locations. This transform can be formulated as [55]:

$$\begin{aligned} {}^{W}P^{A} &= {}^{W}R^{H} \times {}^{H}P^{A} + {}^{W}P^{H} \\ {}^{W}R^{A} &= {}^{W}R^{H} \end{aligned} \quad (3)$$

where ${}^{W}P^{A}$ and ${}^{W}R^{A}$ are the position (x,y,z) and quaternion rotation of the antenna in the world frame W; ${}^{W}P^{H}$, ${}^{W}R^{H}$ are the position and quaternion rotation of the Hololens in the world frame. The x,y,z translation from the Hololens H to the antenna A is defined as ${}^{H}P^{A}$. The position and rotation of the Hololens are obtained from the vision-based AR self-tracking. We empirically measure the translation from the Hololens's center to antenna (${}^{H}P^{A}$) since this translation is fixed and results from mounting the antenna on the headset.

After applying the transformation, X-AR uses them as the antenna array locations. This allows it to then exploit wideband measurements as per Eq. 1 to opportunistically apply SAR along the user's trajectory.

RFID Localization. Fig. 3 shows an example of X-AR's RFID localization (shown in 2D for simplicity). Fig. 3a shows an overhead view of a user walking through the environment. RFID measurements are taken during the user's trajectory, resulting in the measurement locations shown by the red stars. These measurements are then used to compute the power at each point in the workspace using Eq. 1 to estimate the tag's location. Fig. 3b shows this power as a heatmap with yellow indicating areas of higher power and blue indicating areas of lower power. The tag's location (red dot) overlaps with the area of highest power, showing that the localization was successful. While the above description focused on 2D localiza-



(a) 2D SAR (b) 2D SAR result (c) User view of AR display

Figure 3: **AR-Based SAR.** (a) As the user moves naturally, X-AR collects RF measurements. (b) Using RF-Visual SAR, X-AR creates a heatmap of the RFID tag's possible location. The target RFID location overlaps with the area of highest power (yellow), indicating a successful localization. (c) The user's view from the Hololens application. The sphere shows the estimated tag location and the arrow points to it.

tion, the same method extends to 3D as per Eq. 2, enabling X-AR to localize items in 3D space.

Holographic Visualization. Once the item has been localized, X-AR leverages holographic visualization to display its location to the user and guide them towards it. To do this, X-AR leverages the transforms described in Eq.3 to compute the tag's location in the world frame. Fig. 3c shows an example from the user's perspective as displayed on the AR headset. In this example, X-AR places a spherical hologram around the estimated location, and a floating arrow appears in order to guide the user towards the localized tag for object retrieval. The arrow is programmed to float slightly above the user's eye level at a fixed distance in front of them. For every frame update, the application queries the location and rotation of the user in the world space. It then computes their directionality to update the pointing vector of the arrow to properly guide the user towards the target object.

4.3 Practical Considerations

Standard wideband SAR systems typically design their antennas to have uniform gain across the entire frequency band. However, off-the-shelf UHF RFID tags are not designed to be wideband and therefore have significant variability in their antenna gain across frequency. In general, measurements with frequencies further from the tag's resonant frequency (typically 900MHz) will be weaker and therefore more susceptible to noise. These weak measurements can introduce significant error in the location estimate. To overcome this, we introduce a weighted SAR formulation that biases the estimation towards confident measurements to improve the accuracy.

To do this, X-AR starts by quantifying its confidence in each of its measurements using the signal-to-noise ratio (SNR). For any wideband measurement with an average SNR below a certain threshold, X-AR is unlikely to be able to accurately estimate the RFID channel and it therefore removes the measurement from the SAR formulation entirely. The remaining measurements all contain useful information, however, as described above, certain frequencies in each wideband measurement may have weaker responses due to the tag's frequency dependent response. To prioritize frequencies with stronger responses, X-AR applies an SNR-based weighting function to each frequency in a measurement.

This is formalized in the following equation:

$$P(x,y,z) = \left|\left| \sum_{j=1}^{M} \sum_{i=1}^{N} \begin{cases} w_{i,j} h_{i,j} e^{\frac{4\pi d_i}{\lambda_j}} & \overline{SNR_i} > \tau \\ 0 & \overline{SNR_i} < \tau \end{cases} \right|\right| \quad (4)$$

$$w_{i,j} = \frac{SNR_{i,j}}{\max_{k\in[1,M]}(SNR_{i,k})} \quad (5)$$

where $w_{i,j}$ is the weight for the ith location and jth frequency, and τ is the SNR threshold for removing poor measurements. $SNR_{i,j}$ is the SNR of the ith location with the jth frequency, and $\overline{SNR_i}$ is the average SNR across all frequencies for the ith location. [2]

A few additional points are worth noting:

• In practice, the self-localization frame rate is different from that of the RFID channel measurements. To overcome this, X-AR linearly interpolates between Hololens self-tracked locations to find the corresponding location of the mounted antenna for any given measurement.

• X-AR continues to collect measurements until it has become confident in the tag's location. To determine its confidence, it finds all (x,y,z) locations whose power is within 0.75dB of the peak power.[3] It then computes a bounding box around these locations. When this bounding box's size falls below a threshold, X-AR declares the localization complete and visualizes the location.

5 RF-Visual Verification

So far, we explained how X-AR opportunistically leverages human motion to localize RFID-tagged target items and visualize them on the AR headset for retrieval. In principle, this visualization should be sufficient to indicate to the user to pick up the item within the holographic sphere shown in Fig. 3c. In practice, however, the user may still pick up an incorrect item. For example, multiple items may lie within the glowing sphere.[4] Even if the user knows what they're looking for (e.g., red shirt), there might be several items that are visually similar to each other or in similar packaging in the region. More generally, the picked item may be incorrect because the

[2] When computing $w_{i,j}$ in our implementation, we offset all of the SNR values and clip them at 0 to avoid negative weights.

[3] In practice, other thresholds are possible, but a looser threshold would reduce the confidence and hence the localization accuracy.

[4] The size of the sphere is determined by the confidence interval from RFID localization accuracy which is around 10-20 cm.

picker is prone to human error.

To ensure that the user has picked up the correct item, X-AR incorporates a mechanism for picking verification. We describe *RF-Visual Verification*, which enables an accurate and seamless verification of grasped items.

5.1 RF-Visual R-SAR

At the most basic level, the goal of X-AR's RF-Visual verification primitive is to verify whether the correct item is *in* the user's hand after they have picked up an object. Said differently, it aims to localize the RFID-tagged target item to within the user's palm. At first blush, one might assume that such a capability is trivial given that X-AR already has a mechanism to localize RFIDs, as described in the previous section on RF-Visual SAR. However, the two localization problems are fundamentally different. Unlike the earlier scenario where the user's walking emulates synthetic aperture, a user picking an item is in a relatively fixed location. Hence, one cannot leverage the user's movements to localize the item.

To localize the item despite the user's stationary position, X-AR leverages the RFID tag's mobility instead. Fig. 4a shows a sample scenario, demonstrating how the tag itself traces an antenna array. X-AR leverages this emulated array in order to localize the AR-headset (more specifically, the antenna on the headset) with respect to the array. This formulation is the reverse of the SAR described in §4, where the RFID tag was stationary, and the AR conformal antenna on the headset was moving with the user. Notably, in §4, we could leverage the AR headset's self-tracking capability to track the antenna locations. Here, we still need a mechanism to track the RFID locations in order to properly apply the antenna array equations.[5] To track the RFID's location as it moves, our idea is to leverage the *hand-tracking capability* of the AR headset. Specifically, AR headsets like the Microsoft Hololens 2 can detect and track multiple feature points on a user's hand, including their palm [8]. Thus, if the user picks up the correct RFID-tagged item, then the RFID traces a similar trajectory to the user's palm as shown in Fig. 4a.

X-AR leverages the above observation and applies the antenna array equations on the hand's trajectory in order to localize the headset. If the headset's estimated location using this method coincides with the headset's visual-inertial odometry-based location, that indicates that the target RFID tagged item was accurately retrieved and is indeed in the user's hand. On the other hand, if the headset localization fails, the failure indicates that the target RFID tag is not in the user's hand. Below, we formalize the above intuition by describing scenarios where the user picks the correct item and compare it to a scenario where the user picks an incorrect item.

[5] In principle, one could use ISAR [11]. It is less desirable than SAR because the former suffers from a larger direction location ambiguity.

(a) Scenario where the User Picks the Correct Item. Fig. 4a shows an example where the target item is in the user's hand. Here, the palm location (P_{palm}) and the tag location (P_{tag}) are similar. As the user's hand moves, P_{palm} and P_{tag} change similarly together. As a result, the target tag's location can be accurately approximated with the palm location over time for applying SAR and estimating the AR conformal antenna's location according to the following equation:

$$P(x,y,z) = || \sum_{j=1}^{M} \sum_{i=1}^{N_v} h_{ij} e^{\frac{4\pi d(t_i)}{\lambda_j}} || \quad (6)$$

$$d(t_i) = |(x,y,z) - P_{\text{palm}}(t_i)| \quad (7)$$

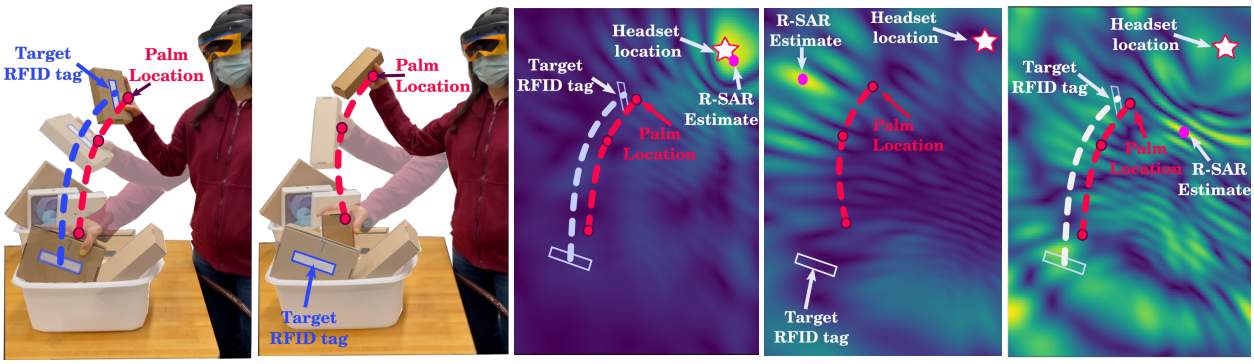
$$(x_h, y_h, z_h) = \max_{x,y,z} P(x,y,z) \quad (8)$$

where N_v is the number of measurements, t_i is the time of ith measurement, $d(t_i)$ represents the distance at time t_i from the (x,y,z) position to the user's palm location, $P_{\text{palm}}(t_i)$. X-AR obtains $P_{palm}(t_i)$ through vision based hand tracking. The SAR estimated headset location, (x_h, y_h, z_h), is the position that emanated the maximum power. Remember that when the user has the target item in their hand, $P_{\text{palm}}(t_i)$ is similar to the target RFID location at time t_i.

Fig. 4c shows the result of applying SAR to localize the headset in the form of a 2D heatmap from a side view. For simplicity, the result of antenna array projections is sliced in the plane that coincides with the real-world plane containing the user's body and the RFID-tagged item. In this heatmap, yellow indicates higher probability of the headset location, while navy blue indicates low probability. As the figure shows, the location of highest power (the pink dot) is very close to the actual location of the headset antenna (the white star), indicating that the headset has been accurately localized.

(b) Scenario where the User Picks an Incorrect Item. Next, consider a scenario where the user picks up an incorrect item, as shown in Fig. 4b. Here, the user's palm location (P_{palm}) changes as the user's hand moves, but the target RFID tag location (P_{tag}) does not change. In this case, when X-AR uses the user's palm location to estimate the tag location for the SAR, it will fail to accurately locate the AR conformal antenna location.

Figure 4d shows the result of applying SAR in this scenario. Notice how the heatmap displays multiple high probability regions that are far from the actual headset location. In this case, the highest probability location (depicted by the pink dot) which corresponds to the SAR-based estimate of AR conformal antenna's location is far from the actual location of the headset antenna (depicted by the white star). Thus, the SAR based headset localization fails because of large error. Since the headset knows its actual location (using the self-tracking via visual-inertial odometry as described in §4), it can determine that the reverse localization has failed, and use this information to determine that the target RFID tag is not



(a) Target is in hand (b) Target is not in hand (c) In-hand heatmap (d) Not in-hand heatmap (e) Without compensation

Figure 4: **RF-Visual In-Hand Verification.**(a) The RFID trajectory (blue dashed line) is similar to the palm trajectory (red dashed line) when it is in-hand. (b) The RFID's location (blue rectangle) differs significantly from palms trajectory(red line) when not in-hand. (c) When the tag is in hand, RF-Visual R-SAR accurately estimates the headset location (pink dot) relative to the actual headset location (white star). (d) The R-SAR estimation of the headset location (pink dot) is not accurate when the tag is not in hand. (e) Without compensating for natural head movement, RF-Visual R-SAR cannot locate the headset accurately even when the target RFID is in the user's hand.

in the user's hand.

The criteria for declaring that the target tag is in the user's hand is that the headset localization error should be within an acceptable range and can be formulated as follows:

$$||(x_h, y_h, z_h) - (x_G, y_G, z_G)|| < \tau \quad (9)$$

where (x_G, y_G, z_G) is the headset's location based on built-in odometry, and τ is threshold for localization error.[6]

5.2 Compensating for Natural Tilts

Our above description assumes that the user's head is perfectly still as they are picking an item. In practice, a user's head naturally tilts during picking, and its important to compensate for these tilts in the reverse SAR localization.[7]As a result of head movement throughout the retrieval process, the distance of user's palm to headset's initial location can be different from the actual distance from the user's palm to the headset's antenna location.

In Fig. 4a, we had shown the result of SAR after compensation. For comparison, Fig. 4e shows the result of applying SAR without compensating for the user's natural head movements. Multiple high probability regions are visible in the heatmap showing that if the natural head movements are not accounted for, the SAR estimated head location may have a large error and the item in the user's hand may be incorrectly classified.

To address this issue, X-AR tracks these natural head movements through the visual-inertial odometry and compensates for them in the RF-Visual SAR formulation. Specifically, X-AR translates the palm position from current headset coordinate to the initial headset coordinate. This can be formulated by replacing $d(t_i)$ in Eq. 6 with $\hat{d}(t_i)$ as follows:

$$\hat{d}(t_i) = |(x,y,z) - (P_{palm}(t_i) - [P_{head}(0) - P_{head}(t_i)])|$$

where $P_{head}(t_i)$ is the visual-inertial odometry-based head location at time t_i. In this new formulation, $\hat{d}(t_i)$ represents the compensated distance from head's initial position to the palm location at time t_i. The headset's estimated initial location, $P_{head}(0)$, is the same as (x_G, y_G, z_G) in Eq. 9. X-AR uses the same criteria as Eq.9 for the headset's initial location to determine if the target item is accurately retrieved by the user. In the system evaluation, we demonstrate how much this compensation is critical for RF-Visual Verification. We also note:

- X-AR can also use the camera visual data to determine if and when the user grasps an item by tracking her hands and fingers. It can use this information to trigger the RF-Visual verification module.
- The retrieval process often includes grasping and removing items to declutter the surroundings of the target item before the user actually grasp the target item. As a result, X-AR uses the latest received N_v RFID measurements[8] at each point of time for the RF-Visual verification. When the latest N_v satisfy the Eq.9's criteria, X-AR notifies the user that the target item is in her hand by showing text stating that target item is retrieved.

6 Implementation & Evaluation

Physical Setup: We implemented X-AR on a Microsoft Hololens 2. We mounted our custom conformal antenna on the front visor of the AR headset and connected it to two Nuand BladeRF 2.0 Micrsoftware radios [47]. Our device was tested using standard off-the-shelf UHF RFID tags [4](3-5 cent each). We tagged common items such as office supplies or clothes and placed them in boxes of different arrangements.

RFID Reader: To obtain wideband RFID channel measurements for localization, we implemented the EPC Gen 2 protocol [31] on a wideband RFID reader design similar to [17]. In order to transmit and receive signals from a single antenna, we introduced a CS-0.900 circulator to the reader. To cancel self-interference and extend

[6]In our implementation, τ is 0.3m. Note that the length of the AR-conformal antenna is 0.165m. We experimented with different τ's and found this achieves a good balance between precision and recall.

[7]Note that these tilts remain too subtle to perform SAR on the head movement itself, but are sufficiently large to make reverse SAR inaccurate if they are not accounted for.

[8]In our implementation, N_v is 35.

the range, we implemented over-the-wire nulling through the BladeRF's MIMO capability[47] and a ZAPD-2-21-3W-S+ 2-Way Pass DC Splitter. We connected the reader to a Raspberry Pi to collect and process RFID measurements from the software defined radios.

Software: We implemented the processing described in §4 and §5 on an edge server running Ubuntu 20.04 on an Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz. The code is developed in Python and C++ and uses ROS [50] to enable multicore processing. We developed our own application for the Hololens to stream device transforms and tracked hand locations to the edge server via TCP protocol and present the designed UI to the user. The application was developed in C# in Unity3D [56] and Visual Studio IDE [43]. On the Rasberry Pi, we implemented code in Python to stream the processed RFID channel estimates to the edge server.

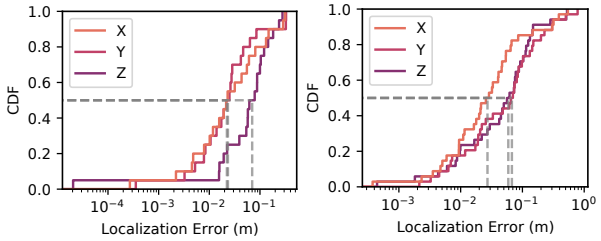
Evaluation Environment: We evaluated X-AR in multipath-rich indoor settings that mimic warehouses, retail stockrooms, and dark stores, which are cluttered with boxes. Fig. 3c shows a sample evaluation environment. Across experimental trials, we changed the arrangement of stacked boxes, moving them near metal shelving and/or wooden bench tops. Since our evaluation setups were created in a lab, they were also surrounded by furniture including chairs, desks, and computers. These environments also had typical wireless interference from various technologies, as well as multipath interference from building occupants who walked around the environment while going about their daily activities. Across our experimental trials, a user wears the X-AR headset and walks around to find and pick up an RFID-tagged target item. To evaluate localization with various human trajectories, we asked users to walk in several different patterns. These patterns included walking towards the target object, in a diagonal path approaching the target, and in 2D "L" or "V" shaped trajectories with respect to the target. We tested objects of different sizes/shapes across both LOS and NLOS settings. In LOS, the tagged object was not occluded from the AR headset's field of view. For NLOS settings, the RFID-tagged target was hidden inside a box or behind clutter. Across trials, we varied the target RFID-tagged object's location to cover various potential scenarios.

Baselines. We implemented 2 state-of-the-art baselines:

SAR baseline: Our first baseline is SAR-based localization algorithm (similar to [58, 65]). In this baseline, we used the AR-based VIO similar to X-AR to obtain antenna locations. However, we limited the localization to only frequencies within the UHF ISM band (around 22 MHz), and did not implement X-AR's weighting optimizations as described in §4.3.

Time-of-Flight baseline: Our second baseline implements state-of-the-art time-of-flight(ToF) estimation using wideband RFID measurements (similar to [40, 17]). For this baseline, we selected 6 measurements from the user's trajectory (similar to [17], spaced evenly in time), computed the ToF-based distance estimates (using the algorithm from [40]), then performed robust trilateration to compute the final 3D location (as in [17]).

For fairness of comparison, in both baselines, we applied the same initial SNR filter as X-AR to remove low-confidence measurements.



(a) Line-of-sight (LOS) (b) Non-line-of-sight (NLOS)

Figure 5: **3D Localization Accuracy.** CDF plots of X-AR's RF-visual SAR localization accuracy in the x/y/z dimensions for LOS and NLOS.

Ground Truth: To measure the localization accuracy of our system, we used the AR headset's built-in spatial awareness to determine the origin of the coordinate system in each trial. In each experimental trial, we aligned the tag's location with the Hololens's origin. This was done by manually moving the RFID tag to the origin (displayed as a hologram by the AR advice) so that it aligns with the Hololens origin at the beginning of each trial. Subsequently, the localization accuracy was computed as the difference between X-AR's RFID tag estimated location and the Hololens' origin. This was repeated for each experimental trial.

7 Results

We ran 234 trials to evaluate the performance of X-AR.

7.1 3D Localization Accuracy

We first evaluated the accuracy of our system in localizing target RFID-tagged items in the environment. We define the localization error to be the euclidean distance between the system's estimated location and the ground-truth location. We ran 54 experimental trials to measure the performance of RFID localization. In each trial, the user walked in a different motion pattern and X-AR automatically localized the target item via RF-visual SAR as described in §4.

Fig. 5 plots the CDF of the localization error across the experimental trials in both line-of-sight and non-line-of-sight scenarios. We plot the localization error along the x(orange), y(pink), and z(purple) dimensions. We note:

• In LOS settings, the median errors are 2.1 cm, 2.1 cm, and 8.4 cm along the x, y, and z dimensions, respectively. In NLOS settings, the median errors are 1.9 cm, 6 cm, and 7.7 cm along the x, y, and z dimensions, respectively. These results demonstrate that X-AR can achieve

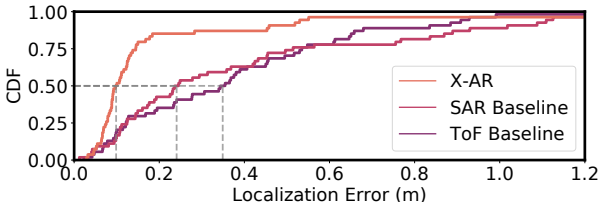


Figure 6: **Comparison to Baseline Localization Accuracy.** CDF plots of L2-norm error for X-AR(orange), *SAR*(pink), and *ToF*(purple).

centimeter-level localization accuracy in each dimension while opportunistically leveraging human motion that is not known a priori or directed in a particular way.

• The median L2 norm of localization error for the LOS and NLOS scenario are 9.6 cm and 10.6 cm. Therefore, there is no significant difference between localization error for NLOS and LOS, showing that X-AR's is able to augment the AR device with perception capabilities for both LOS and NLOS conditions.

• The localization accuracy along the x-axis is generally better than along y and z (especially in the NLOS scenario). This is because in our experimental setup, the object is located on a shelf against a wall. The user walks toward the shelf but not past it, meaning the RFID measurements are only on one side of the RFID in the y direction. On the other hand, the user walks parallel to the shelf and measurements are taken on both sides of the RFID along the x-axis leading to a better accuracy in the x direction than in the y direction. Note that the aperture in z direction (vertical direction) is very small since the user's head does not move vertically.

Baseline Comparison: We compare the performance of our system to the two baselines described in §6. We used the same experimental trials for X-AR and the baselines.

Fig. 6 plots the CDF of the total localization error for X-AR (orange), *SAR Baseline* (pink), and *Time-of-Flight Baseline* (purple). For simplicity, we show the L2-norm distance error (rather than the error along each of the x/y/z dimensions). We make the following remarks:

• For X-AR, the median and 90th percentile localization errors are 9.8 cm and 45 cm, respectively. These results are in-line with those reported above (as L2-norm in 3D).

• For *SAR Baseline*, the median and 90th percentile localization errors are 24.8 cm and 99.1 cm, respectively. This shows that by leveraging our system's custom wideband antenna and wideband RF-visual SAR techniques, X-AR can achieve over $2\times$ performance improvement in both the median, and 90th percentile over a system that is limited to the UHF ISM band, thus demonstrating the value of our customized wideband conformal antenna design and RF-visual SAR localization scheme.

• The *Time-of-Flight* baseline has a median and 90th percentile localization errors of 34.9 cm and 78.8 cm. This shows that X-AR has an improvement of over $3\times$ in the median and almost $2\times$ in the 90th percentile. We note that the baseline's performance is worse than that reported in prior work [40, 17]. This is because that prior work had control over the aperture of measurements (i.e., through physical antenna placement or controlling robotic motion). In contrast, when applying these techniques to an AR system with natural human motion, the aperture cannot be optimized and the resulting accuracy is poor. This demonstrates the benefit of our AR-based SAR techniques when utilizing natural human motion.

Impact of Walking Pattern: Next, we investigated the impact of different walking patterns on X-AR's localization accuracy. Recall that we asked users to walk in different patterns: vertically toward the tag's plane, diagonally toward the tag, as well as L-shaped and V-shaped trajectories. To understand the impact of different motion patterns on localization accuracy, we measured the 10th, median and 90th percentile for each of these patterns and reported them in Table 1.We note:

	Vertical	**Diagonal**	**L-shape**	**V-shape**
10th percentile	5.7 cm	3.8 cm	7.9 cm	6.3 cm
50th percentile	10.8 cm	12.5 cm	9.8 cm	8.4 cm
90th percentile	47.7 cm	51.0 cm	14.9 cm	13.3 cm

Table 1: **Trajectory Impact.** Location error for different trajectories.

• All walking patterns have a similar median localization error, between 8.4 to 12.5 cm. This shows that X-AR works well in different motion patterns and is generally robust to different trajectories. It also suggests that X-AR does not need to constrain the user to a pre-defined 2D trajectory to achieve good localization performance.

• Interestingly, we noticed that 90th percentile accuracy is markedly different across these motion patterns. In particular, while the L-shaped and V-shaped patterns have a 90th percentile around 15 cm, this error increases to around 50 cm for linear motion patterns (diagonal & vertical). This is likely due to the differences in spatial diversity and aperture variability across these motion patterns. In particular, L-shaped and-V-shaped trajectories involve independent mobility in two dimensions, while the diagonal and vertical trajectories involve mostly linear motion patterns, giving less overall aperture.

Impact of Aperture. We investigated the impact of the trajectory's aperture size on localization accuracy through a micro-benchmark evaluation. To do this, rather than providing the RF-visual SAR algorithm with the entire trajectory for localization, we trimmed the trajectory of each trial to a certain maximum aperture. For example, to evaluate an aperture of 0.6 m, we only provided the first 0.6 m of the user's trajectory to the localization algorithm.[9] We repeated the same process for apertures of different lengths, and computed the localization accuracy for each of them across all the experimental trials.

[9]The aperture of a trajectory is defined by the diagonal of the bounding box encompassing the measurements in that trajectory.

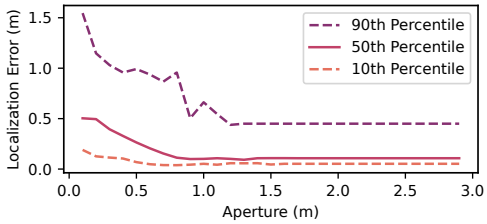


Figure 7: Impact of Aperture. Localization error vs the aperture of the user's trajectory. The plots show the 10th, 50th, and 90th percentiles.

Fig. 7 plots the L2-norm localization error as a function of the aperture. The plot shows the 10th (orange), 50th (pink), and 90th (purple) percentile errors across all experimental trials. We make the following remarks:

• When limiting the aperture to 0.1 m, the 50th and 90th percentile errors are 0.5 m and 1.5 m respectively. As the aperture increases to 0.8 m, these errors drop to 0.11 m and 0.96 m. This shows that for small apertures, X-AR's performance greatly improves as the user walks further.

• After the aperture reaches 0.8 m, the median errors become relatively constant. For example, expanding the aperture to 1.2 m only decreases the median error by 2 cm. This shows that increasing aperture after 0.8 m does not improve the median localization accuracy.

• The 90th percentile continues to improve as the aperture is increased from 0.8 m to 1.2 m, dropping by 0.43m. This shows that larger apertures improve reliability.

• X-AR visualizes the RFID tag on the AR device once the user's walking trajectory allows for adequate RF measurement aperture, such that X-AR is confident about the RFID tag's location, as described in §4.3. As a result, the time it takes X-AR to find the requested item is dependent on the user's walking speed and trajectory.

Impact of SNR-based Weighting Function. We investigated how weighting measurements based on the received SNR impacts the localization accuracy of X-AR. We processed the experiments with SNR-based weighting (Eq.4) and with uniform weighting (Eq.1) and calculated the L2 norm of RFID localization error. Our results showed that the SNR-based weighting improves the robustness of the system, specifically in the 90th percentile localization accuracy. While the uniform weighting and SNR-based weighting have a similar median errors (around 10 cm), the 90th percentile in our SNR-based weighting approach is 45 cm, while the uniform weighting approach has 71 cm error.

7.2 In-Hand Verification

Next, we evaluated X-AR's ability to successfully determine if the correct RFID tagged object was retrieved by the user. We conducted 180 trials in total. In each trial, the user grasped a tagged or non-tagged item and moved their hand in a pick and place motion. In each trial, X-AR predicted whether the RFID tag was in the user's hand or not, (i.e., the correct item being picked or not). We define a successful trial as one in which X-AR correctly determines whether or not the tag was in the user's hand.

	Precision	Recall	F-score
Extracting RFID-tagged item (LOS)	98%	100%	98.9%
(without compensation)	98%	98%	98%
Picking boxed RFID-tagged item (NLOS)	100%	85.1%	91.9%
(without compensation)	100%	74.3%	85%
Large Object (LOS+NLOS)	100%	87.5%	93%
Small Object (LOS+NLOS)	98%	93%	95.4%

Table 2: **In-hand Verification Accuracy.** The table reports the results for in-hand verification across different evaluation scenarios. The results are reported as percentages for precision, recall, and F-measure.

Table 2 reports the results for X-AR's in-hand verification algorithm. Here, *Precision* indicates the number of trials where the target item was correctly classified in-hand divided by the overall number of trials that systems classified the target item as in-hand. *Recall* indicates the number of trials where target item was correctly classified in-hand divided by the overall number of trials where the target RFID tagged item was actually in the user's hand. We make the following remarks:

• X-AR achieves a 98% precision rate, and 100% recall rate. These values demonstrate that when the user retrieves an item, X-AR can reliably and correctly predict whether the target item has been picked up.

• The system has 98% precision rate, which indicates 2% of the trials when the system registered it as a potential retrieval, the user has picked up an incorrect item (e.g., non-tagged item, or potentially an item that is tagged with a different RFID). We suspect the reason for some trials being mistakenly registered as positive arises from multipath. Specifically, even though the user did not pick up the target RFID-tagged item in these trials, the wireless signal reflecting off the user's hand during motion creates an array of the multipath reflections. Such multipath arrays may have inadvertently allowed localizing the headset, resulting in false positives.

Picking Boxed RFID-tagged items (NLOS): We also evaluated whether X-AR can accurately verify when a user picks up an RFID-tagged item that remains *inside* a box during the picking process. While such scenarios are less likely in practice (e.g., in warehousing or retail), they may arise and serve to test the limit of our system in performing RF-visual verification of RFIDs in NLOS.

The results for this experiment are shown in the third row in Table 2. The results show that even though the recall rate drops, X-AR remains largely successful in performing the verification, achieving a precision and recall rate of 100% and 85.1%. This change in performance can be attributed to the fact that when target tags are not in line-of-sight (and are inside a box) their distance to the user's palm is markedly higher. This offset between the tag location and visually extracted palm location impacts the reverse SAR calculation. In the future, this may

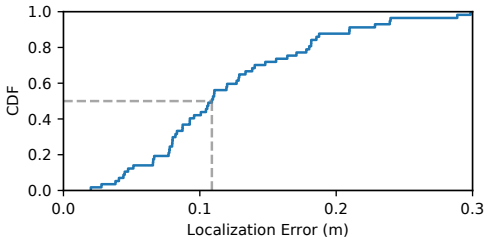


Figure 8: **CDF of Reverse SAR.** CDF plots of the headset's localization accuracy by applying R-SAR on the trajectory of the target item.

be compensated for by estimating the potential location of the RFID inside the box and/or investigating different features from the AR headset's built-in hand-tracking.

Impact of Motion Compensation: Recall from §5 that X-AR's RF-visual verification primitive compensates for head tilts in the reverse SAR localization process. To investigate the impact of such compensation, we processed the same experimental trials as above (for both LOS and NLOS scenarios) without performing motion compensation and reported the results in Table 2. The table shows that the recall rate drops for both LOS (from 100% to 98%) and NLOS scenarios (from 85.1% to 74.3%). This demonstrates that by accounting for head tilts, X-AR's accuracy in prediction markedly improves.

Impact of Target Object Size: Next, we investigated the impact of target object size on the verification accuracy. Specifically, we divided our experimental trials between objects of smaller and larger sizes. The object size was determined by their longest dimension, using 10 cm as a divider between objects that we referred to as small and large - i.e, objects with largest dimension $<$ 10 cm classified as small, and those with a dimension $>$10 cm classified as large. In practice, choosing a different threshold does not make a significant difference, as the primary goal of this experiment was to micro-benchmark the impact of object size on verification accuracy.

The last two rows of Table 2 show the results comparing the accuracy for different object sizes, covering both LOS and NLOS scenarios. The table shows that smaller items have higher recall rate (93%) than larger items (87.5%). This can be attributed to the fact that for larger items, there is a larger offset between the tag location and palm location. Specifically, recall from §5 that X-AR approximates an RFID's location as the user's palm location (extracted from the AR-headset's hand-tracking module). As a result, the smaller the object is, the more accurate this approximation is, leading to higher accuracy for smaller objects. In the future, it would be interesting to explore mechanisms that adapt the threshold to the object size, or alternatively leverage the RFID location inside the box and apply a transformation to the user's palm to compensate for these differences and achieve higher accuracy for larger objects.

Reverse SAR Localization Accuracy. Our final result looks into the reverse SAR localization accuracy. Recall from §5 that X-AR's verification component relies on the ability to correctly localize the headset (specifically the AR-conformal antenna) by applying SAR on the mobile tag. To investigate this primitive, we evaluated the method's ability to correctly locate the position of the user's head. Here, we defined the ground truth of the location of the user's head to be the visual-inertial odometry-based location and estimated the error by calculating the euclidean distance between the ground truth and X-AR's predicted location. We computed the localization error for all scenarios where the user picks up an RFID tagged item.[10] Here, we included experimental trials from LOS scenarios described above.

The CDF of the localization error is plotted in Fig. 8. The figure shows that the method allows localizing the headset using SAR with a median accuracy of 11 cm and a 90th percentile accuracy of 19.6 cm. These results show that even with simple pick and place movements, X-AR can accurately locate a user's head using reverse SAR techniques, while compensating for head movements. This high localization accuracy is why the system can accurately verify picking RFID-tagged items.

8 Related Work

RFID Localization. RFID localization is a well-studied problem in the networking community with researchers exploring various techniques including received signal strength (RSS) [20, 46], angle of arrival (AOA) [13, 36, 71], and wide-band sensing [40, 16, 39]. The closest to X-AR is past work that leverages motion for RFID localization, which falls in two main categories. The first places an antenna on robots that move along *predefined* trajectories and leverage these trajectories to localize the tags [57, 29, 45, 53, 17, 16, 44, 70, 14]. Our system does not require users to move along specific (unnatural/robotic) trajectories, yet can still localize accurately by leveraging natural movement. The second category tracks RFIDs that are already in motion, e.g., for gesture recognition [59, 65, 21, 66]. Our work differs from these in that it can also localize stationary tags by using an AR mounted antenna. Thus, our work is the first to bring fine-grained RFID localization to AR headsets, addressing challenges that span antenna design, natural human mobility, and various localization artifacts.

Augmented Reality. Augmented Reality (AR) refers to systems that overlay a virtual world on top of the physical world to enable new experiences and interactions [12, 35]. Most prior work that leverages RF in AR systems does not involve headsets altogether and simply visualizes tagged items on a screen or a smartphone [49, 37, 62, 63]. This includes past work that

[10] Note that the error for non-tagged items is much higher since the formulation does not hold. Empirically, the median localization error for those scenarios is over a meter.

deploys an RFID localization infrastructure in the environment and uses it to localize tags and visualize their locations on a screen [49, 37]. It also includes robotic systems mounted with RFID readers and cameras to scan the environment and send the result for visualization on a screen[62, 63]. X-AR builds on this area and brings RFID localization to AR headsets, addressing the associated challenges in antenna design, human mobility, and headset-based localization. X-AR is also related to past work that involves users wearing RFID readers on their hands or in their backpacks to detect objects in the environment [69, 54] or self-localize [64, 37]. X-AR differs from these systems in directly integrating the localization and sensor fusion into the headset itself, resulting in a more natural and seamless AR experience.

Conformal Antennas. Antenna design is a mature field that targets satisfying multiple requirements such as compactness, robustness to flexing, radiation pattern, and weight. The closest to our work are Bluetooth headset antennas desired to radiate outwards while close to a human head, and designed to be mounted around the ear or on glasses handles [23, 22, 34, 33]. These past designs differ from our work in their bandwidth requirements, desired radiation pattern, and form factor. Other wearable antennas were designed for safety helmets [24] or smart glasses [60], but were either too bulky and obstructive or lacking the wideband operation desired for wideband RFID localization. Loop antennas are simple, and do not require a ground plane, but are inherently narrowband. Past techniques such as tapering and slots help improve their bandwidth, but none of the existing wideband loop antenna designs can simultaneously operate at the desired frequency range while matching the dimensions of the visor [68, 61, 67]. Our proposed design takes advantage of wideband antenna techniques to deliver a broadband, compact, and conformal loop antenna that perfectly fits on the headset's visor without covering the cameras or blocking the user's view.

9 Discussion and Limitations

Antenna Placement: In principle, one could design alternate versions of our X-AR system by placing the antenna on top of the headset, on the user's shoulder, or even in the user's hand. However, these alternative approaches are suboptimal in most scenarios compared to X-AR's design. For example, in warehouses, pickers are more efficient when they can use both hands (rather than carring an antenna with one hand all the time). Similarly, mounting large and heavy antennas on their shoulders or heads would create undesirable additional weight which may impact their balance. That said, it is possible that such alternate implementations may be useful in certain use-cases and can be explored as the research evolves.

Transmission Power: X-AR's transmit power is lower than that of existing wrist-worn RFID readers [10] since bladeRF software-defined radios transmit less than 8dBm. This is also lower than the power transmitted by Apple AirMax headphones, which use Bluetooth 5.0 technology and have a maximum transmission power of 20 dBm [1, 2]. In production systems, X-AR could leverage a deployed RFID reader infrastructure to power RFID tags in the environment, and an X-AR headset for wideband measurements for localization, UI, etc.

RFID reliability: Our implementation of X-AR inherits the typical limitations of RF/RFID signals. For example, it cannot detect or localize items inside closed metallic boxes. However, it can still read RFIDs on metal or liquid bottles if proper tags are used. Moreover, due to its wideband sensing capabilities, it can work in multipath-rich environments, including those with metal shelving, as demonstrated in our evaluation.

Form factor: As X-AR moves closer to commercial deployments, we envision that the entire RF sensing hardware can be integrated into the headset. In particular, while our proof-of-concept prototype was implemented using software radios and a Raspberry Pi, future versions may be designed in form factors similar to existing RFID reader chips (e.g., Lepton3 [18] that are around 1"x1"x0.1"), thus small enough to fit into AR headsets.

Range: The operation range of X-AR is approximately 3-4 meters which is similar to mobile (portable) handheld RFID readers on the market [3]. While this range is lower than stationary readers (which can reach around 10 m), that is primarily because stationary ones typically transmit much higher power. In contrast, handheld readers usually transmit lower power to conserve their battery life, and we envision the same would be desired for future readers integrated in headsets like X-AR.

10 Conclusion

The past few years have witnessed remarkable advances in augmented reality and its metaverse applications. Motivated by these advances, this paper brings a new sensing modality to AR systems through networked RF sensing, giving them the ability to perceive what used to be invisible to the human eye and to existing AR headsets. In doing so, the paper opens the door to more exciting capabilites and applications at the intersection of RF sensing and AR systems. As the research evolves it would be interesting to explore how various networked wireless sensing modalities and sensor fusion techniques - spanning RFID, WiFi, mmWave, and THz - can further augment augmented reality and open new possibilities in visualization and interaction.

Acknowledgments We thank the anonymous reviewers, our shepherd Dr. Behnaz Arzani, and the Signal Kinetics group for their help and feedback. We also thank Yuechen Wang for her help with UI design and implementation. This research is sponsored by NSF (Awards #1844280 and #2044711), the Sloan Research Fellowship, and MIT Media Lab.

References

[1] Apple air max pro specifications. https://www.apple.com/airpods-max/specs. Apple Inc. .

[2] BLUETOOTH SPECIFICATION Version 5.0. https://www.bluetooth.org/en-us/specification/adopted-specifications. Vol 6, Part A, Page 2536.

[3] RFD40 UHF RFID STANDARD SLED . https://www.zebra.com/us/en/products/spec-sheets/rfid/rfid-handhelds/rfd40.htmll. Zebra Technologies.

[4] ALN-9640 Squiggle Inlay, 2014. Alien Technology Inc.

[5] A New Era for RFID in Retail. https://www.accenture.com/_acnmedia/PDF-155/Accenture-RFID-In-Retail.pdf, 2021. Accenture.

[6] About HoloLens 2. Microsoft, 2022. https://docs.microsoft.com/en-us/hololens/hololens2-hardware.

[7] From Apple to Google, big tech is building VR and AR headsets. The Economist, 2022. https://www.economist.com/business/2022/04/09/from-apple-to-google-big-tech-is-building-vr-and-ar-headsets.

[8] Hand tracking. Microsoft, 2022. https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/mrtk2/features/input/hand-tracking?view=mrtkunity-2022-05.

[9] HoloLens 2. Microsoft, 2022. https://www.microsoft.com/en-us/hololense.

[10] 1153 Bluetooth Wearable UHF RFID Reader, Technology Solutions Ltd. Microsoft, 2023. www.tsl.com/products/1153-bluetooth-wearable-uhf-rfid-reader/.

[11] Fadel Adib and Dina Katabi. See through walls with Wi-Fi! In *ACM SIGCOMM*, 2013.

[12] Ronald T. Azuma. A Survey of Augmented Reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, 08 1997.

[13] Salah Azzouzi, Markus Cremer, Uwe Dettmar, Rainer Kronberger, and Thomas Knie. New measurement results for the localization of uhf rfid transponders using an angle of arrival (aoa) approach. In *2011 IEEE International Conference on RFID*, pages 91–97, 2011.

[14] Tara Boroushaki, Laura Dodds, Nazish Naeem, and Fadel Adib. Fusebot: Rf-visual mechanical search. *Robotics: Science and Systems 2022*, 2022.

[15] Tara Boroushaki, Laura Dodds, Nazish Naeem, and Fadel Adib. Fusebot: Mechanical search of rigid and deformable objects via multi-modal perception. 2023.

[16] Tara Boroushaki, Junshan Leng, Ian Clester, Alberto Rodriguez, and Fadel Adib. Robotic grasping of fully-occluded objects using rf perception. In *2021 International Conference on Robotics and Automation (ICRA)*. IEEE, 2021.

[17] Tara Boroushaki, Isaac Perper, Mergen Nachin, Alberto Rodriguez, and Fadel Adib. Rfusion: Robotic grasping via rf-visual sensing and learning. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 192–205, 2021.

[18] CAEN RFID. https://www.caenrfid.com/en/products/r3100-lepton3/, 2023.

[19] A Cai, TSP See, and Zhi Ning Chen. Study of human head effects on uwb antenna. In *IWAT 2005. IEEE International Workshop on Antenna Technology: Small Antennas and Novel Metamaterials, 2005.*, pages 310–313. IEEE, 2005.

[20] Kirti Chawla, Christopher McFarland, Gabriel Robins, and Connor Shope. Real-time rfid localization using rss. In *2013 International Conference on Localization and GNSS (ICL-GNSS)*, pages 1–6. IEEE, 2013.

[21] Kang Cheng, Ning Ye, Reza Malekian, and Ruchuan Wang. In-air gesture interaction: Real time hand posture recognition using passive rfid tags. *IEEE Access*, 7:94460–94472, 2019.

[22] Aykut Cihangir, Chinthana J Panagamuwa, Will G Whittow, Gilles Jacquemod, Frédéric Gianesello, Romain Pilard, and Cyril Luxey. Dual-band 4g eyewear antenna and sar implications. *IEEE Transactions on Antennas and Propagation*, 65(4):2085–2089, 2017.

[23] Aykut Cihangir, Will G Whittow, Chinthana J Panagamuwa, Fabien Ferrero, Gilles Jacquemod, Frédéric Gianesello, and Cyril Luxey. Feasibility study of 4g cellular antennas for eyewear communicating devices. *IEEE Antennas and Wireless Propagation Letters*, 12:1704–1707, 2013.

[24] Estefania Crespo-Bardera, Aaron Garrido Martin, Alfonso Fernandez-Duran, and Matilde Sanchez-Fernandez. Design and analysis of conformal antenna for future public safety communications: Enabling future public safety communication infrastructure. *IEEE Antennas and Propagation Magazine*, 62(4):94–102, 2020.

[25] Elizabeth Culliford and Nivedita Balu. Facebook invests billions in metaverse efforts as ad business slows. Reuters, 2021. https://www.reuters.com/technology/facebook-revenue-misses-estimates-apples-privacy-rules-bite-2021-10-25/.

[26] Till Dengel. Disruption Denied: How Next-Generation Logistics Create A Resilient Supply Chain. Forbes, 2022. www.forbes.com/sites/sap/2022/06/01/disruption-denied-how-next-generation-logistics-creates-a-resilient-supply-chain/?sh=7152e93744b9.

[27] Heidi Fillmore and Tony Storr. AR and VR in the workplacel. Forbes, 2020. https://www.ibm.com/thought-leadership/institute-business-value/report/ar-vr-workplace.

[28] Susan Galer. Virtual Reality Emerges As Powerful Employee Training Tool. Forbes, 2022. https://www.forbes.com/sites/sap/2022/05/24/virtual-reality-emerges-as-powerful-employee-training-tool/?sh=895d9166969b.

[29] Matthias Gareis, Christian Carlowitz, and Martin Vossiek. A mimo uhf-rfid sar 3d locating system for autonomous inventory robots. In *2020 IEEE MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM)*, pages 1–4, 2020.

[30] Lana Gates. How Augmented Reality Can Improve Employee Performance. Insight, 2018. www.insight.com/en_US/content-and-resources/2018/08062018 -how-augmented-reality -can-improve- employee- performance.html.

[31] GS1 EPC global Inc., 2015.

[32] Peter S Hall and Yang Hao. *Antennas and propagation for body-centric wireless communications*. Artech house, 2012.

[33] W Haydar, Sally AlSayah, and R Sarkis. Design and analysis of conformal antennas for smart glasses. In *12th European Conference on Antennas and Propagation (EuCAP 2018)*, pages 1–5. IET, 2018.

[34] Henrik Jidhage and Anders Stjernman. Hooked loop antenna concept for bluetooth headset applications. In *IEEE Antennas and Propagation Society Symposium, 2004.*, volume 4, pages 3521–3524. IEEE, 2004.

[35] Gregory Kipper and Joseph Rampolla. *Augmented reality: An emerging technologies guide to AR*. Elsevier, 2012.

[36] Rainer Kronberger, Thomas Knie, Roberto Leonardi, Uwe Dettmar, Markus Cremer, and Salah Azzouzi. Uhf rfid localization system based on a phased array antenna. *2011 IEEE International Symposium on Antennas and Propagation (APSURSI)*, pages 525–528, 2011.

[37] Sebastian Kunkel, Robert Bieber, Ming-Shih Huang, and Martin Vossiek. A concept for infrastructure independent localization and augmented reality visualization of rfid tags. In *2009 IEEE MTT-S International Microwave Workshop on Wireless Sensing, Local Positioning, and RFID*, pages 1–4, 2009.

[38] Yang Liu, Haiwei Dong, Longyu Zhang, and Abdulmotaleb El Saddik. Technical evaluation of hololens for multimedia: A first look. *IEEE MultiMedia*, 25(4):8–18, 2018.

[39] Zhihong Luo, Qiping Zhang, Yunfei Ma, Manish Singh, and Fadel Adib. 3d backscatter localization for fine-grained robotics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 765–782, 2019.

[40] Yunfei Ma, Nicholas Selby, and Fadel Adib. Minding the billions: Ultra-wideband localization for deployed rfid tags. In *Proceedings of the 23rd annual international conference on mobile computing and networking (MobiCom)*, pages 248–260, 2017.

[41] Sarmad Nozad Mahmood, Asnor Juraiza Ishak, Tale Saeidi, Hussein Alsariera, Sameer Alani, Alyani Ismail, and Azura Che Soh. Recent advances in wearable antenna technologies: a review. *progress in Electromagnetics Research B*, 89:1–27, 2020.

[42] Microsoft. `https://docs.microsoft.com/en-us/windows/mixed-reality/develop/unity/camera-in-unity`, 2022. Camera setup in Unity.

[43] Microsoft Visual Studio. `visualstudio.microsoft.com`, 2022.

[44] Robert Miesen, Fabian Kirsch, and Martin Vossiek. Uhf rfid localization based on synthetic apertures. *IEEE Transactions on Automation Science and Engineering*, 10(3):807–815, 2013.

[45] A. Motroni, P. Nepa, P. Tripicchio, and M. Unetti. A multi-antenna sar-based method for uhf rfid tag localization via ugv. In *2018 IEEE International Conference on RFID Technology Application (RFID-TA)*, pages 1–6, 2018.

[46] Lionel M. Ni, Yunhao Liu, Yiu Cho Lau, and Abhishek P. Patil. Landmarc: Indoor location sensing using active rfid. *Wireless Networks*, 10:701–710, 2003.

[47] Nuand, BladeRF 2.0 Micro. `https://www.nuand.com/bladerf-2-0-micro/`, 2021.

[48] Vanja Plicanic, Buon Kiong Lau, Anders Derneryd, and Zhinong Ying. Actual diversity performance of a multiband diversity antenna with hand and head effects. *IEEE Transactions on Antennas and Propagation*, 57(5):1547–1556, 2009.

[49] Zulqarnain Rashid, Enric Peig, and Rafael Pous. Bringing online shopping experience to offline retail through augmented reality and rfid. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 45–51, 2015.

[50] Robotic Operating System. `www.ros.org`, 2020. Noetic.

[51] Anshel Sag. Why Microsoft Won The $22 Billion Army Hololens 2 AR Deal. Forbes, 2021. https://www.forbes.com/sites/moorinsights/2021/04/06/why-microsoft-won-the-22-billion-army-hololens-2-ar-deal/.

[52] Anshel Sag. Apple teases metaverse AR plans, stock jumps. Reuters, 2022. https://www.reuters.com/technology/apple-teases-metaverse-ar-plans-stock-jumps-2022-01-28/.

[53] Longfei Shangguan and Kyle Jamieson. The design and implementation of a mobile rfid tag sorting robot. In *Proceedings of the 14th annual international conference on mobile systems, applications, and services (MobiSys)*, pages 31–42, 2016.

[54] TeamViewer. `https://www.teamviewer.com/en-us/frontline/xpick/`, 2022.

[55] Russ Tedrake. *Robotic Manipulation*. 2021.

[56] Unity Technologies. `unity.com`, 2022.

[57] Jue Wang, Fadel Adib, Ross Knepper, Dina Katabi, and Daniela Rus. Rf-compass: Robot object manipulation using rfids. In *Proceedings of the 19th annual international conference on Mobile computing & networking (MobiCom)*, pages 3–14, 2013.

[58] Jue Wang and Dina Katabi. Dude, where's my card? rfid positioning that works with multipath and non-line of sight. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 51–62, 2013.

[59] Jue Wang, Deepak Vasisht, and Dina Katabi. Rf-idraw: virtual touch screen in the air using rf signals. In *ACM SIGCOMM*, 2015.

[60] Yan-Yan Wang, Yong-Ling Ban, and Yanhui Liu. Sub-6ghz 4g/5g conformal glasses antennas. *IEEE Access*, 7:182027–182036, 2019.

[61] Kunpeng Wei, Zhijun Zhang, and Zhenghe Feng. Design of a wideband horizontally polarized omnidirectional printed loop antenna. *IEEE Antennas and Wireless Propagation Letters*, 11:49–52, 2012.

[62] Lei Xie, Jianqiang Sun, Qingliang Cai, Chuyu Wang, Jie Wu, and Sanglu Lu. Tell me what i see: Recognize rfid tagged objects in augmented reality systems. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, page 916–927, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Lei Xie, Chuyu Wang, Yanling Bu, Jianqiang Sun, Qingliang Cai, Jie Wu, and Sanglu Lu. Taggedar: An rfid-based approach for recognition of multiple tagged objects in augmented reality systems. *IEEE Transactions on Mobile Computing*, 18(5):1188–1202, 2019.

[64] Akihiro Yamashita, Kei Sato, Syunta Sato, and Katsushi Matsubayashi. Pedestrian navigation system for visually impaired people using hololens and rfid. In *2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 130–135, 2017.

[65] Lei Yang, Yekui Chen, Xiang-Yang Li, Chaowei Xiao, Mo Li, and Yunhao Liu. Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 237–248. ACM, 2014.

[66] Lei Yang, Yekui Chen, Xiang-Yang Li, Chaowei Xiao, Mo Li, and Yunhao Liu. Tagoram: Real-time

tracking of mobile rfid tags to high precision using cots devices. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 237–248, 2014.

[67] KY Yazdanboost and Ryuji Kohno. Ultra wideband l-loop antenna. In *2005 IEEE International Conference on Ultra-Wideband*, pages 201–205. IEEE, 2005.

[68] Junho Yeo and Jong-Ig Lee. Miniaturized wideband loop antenna using a multiple half-circular-ring-based loop structure and horizontal slits for terrestrial dtv and uhd tv applications. *Sensors*, 21(9):2916, 2021.

[69] J. Zhang, S.K. Ong, and A.Y.C. Nee. Rfid-assisted assembly guidance system in an augmented reality environment. *International Journal of Production Research*, 49(13):3919–3938, 2011.

[70] Run Zhao, Dong Wang, Qian Zhang, Haonan Chen, and Huatao Xu. Pec: Synthetic aperture rfid localization with aperture position error compensation. In *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9, 2019.

[71] Jun Zhou, Hongjian Zhang, and Lingfei Mo. Two-dimension localization of passive rfid tags using aoa estimation. *2011 IEEE International Instrumentation and Measurement Technology Conference*, pages 1–5, 2011.

Appendix

Simulated and Measured Antenna Performance

Fig. 9 shows the simulated radiation pattern of the antenna mounted in this figure on the Hololens for visualization, demonstrating that the pattern is almost omni-directional (with a directivity of 4 dB), allowing the RF sensing module to localize items in the surrounding 3D environment. The fabricated conformal antenna was then placed on the headset and worn by the user to measure the gain across frequencies and elevation angles. The user was tilting their head up and down to mimic a scenario where they are looking for an item in the environment. The measured gains demonstrate the ability of the antenna to operate efficiently across a wide range of frequencies and elevation angles.

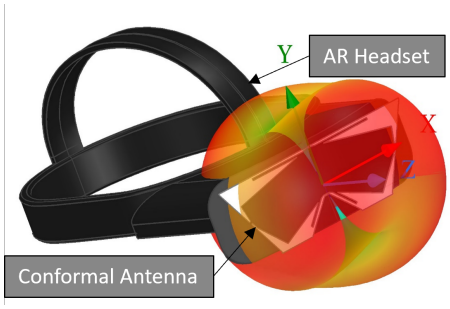


Figure 9: **Conformal antenna 3D radiation pattern.** Simulated radiation pattern of AR-conformal antenna visualized on the AR headset.

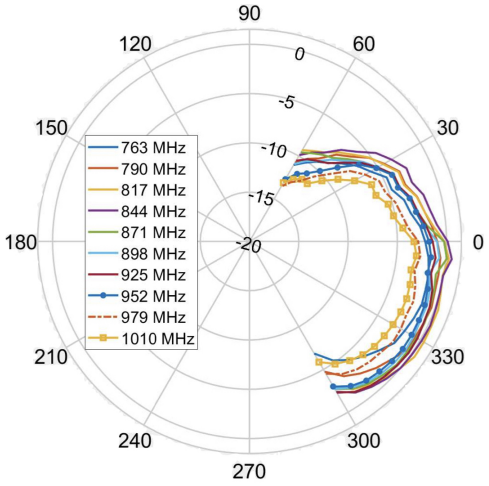


Figure 10: **Conformal antenna measured gains.** Measured gains across different elevations of AR-conformal antenna when mounted on the headset and worn by the user. The user was tilting their head up and down to cover the elevation plane.

Background on SAR

Performing RFID localization using SAR involves 3 steps:

1. The first step is to compute the RFID's channel at each measurement location. As an RFID reader queries different tags in the environment, it can compute the wireless channel h for each of these tags by leveraging the received signal $s(t)$ and the tag's known packet preamble $p(t)$ using the following equation [40]:

$$h = \sum_t s(t)p^*(t) \tag{10}$$

where $p^*(t)$ is the conjugate of $p(t)$.

2. Given the wireless channel from different antenna locations, the second step is to estimate the power arriving from each (x,y,z) location within the workspace. This can be done with the following equations:

$$P(x,y,z) = \left|\left| \frac{1}{N} \sum_{i=1}^{N} h_i e^{\frac{4\pi d_i(x,y,z)}{\lambda}} \right|\right| \tag{11}$$

$$d_i(x,y,z) = \sqrt{(x-x_i)^2 + (y-y_i)^2 + (z-z_i)^2} \tag{12}$$

where $P(x,y,z)$ is the estimated received power, N is the number of measurements taken, h_i is the ith channel estimate, and λ is the wavelength of the used signal.[11] (x_i, y_i, z_i) is the location of the ith measurement, and d_i is the distance from (x_i, y_i, z_i) to (x,y,z).

3. The third step is to localize the tag by assigning its location to the (x,y,z) location with the highest power. Formally, the location of the tag, p_{tag}, is:

$$p_{tag} = \texttt{argmax}_{(x,y,z)}(P(x,y,z)) \tag{13}$$

Finally, past work has shown that including frequency diversity (i.e., wideband measurements) in addition to SAR's spatial diversity can improve the localization accuracy. To do this, wideband SAR takes measurement across a wide range of distinct frequencies and coherently combines the measurements for each frequency and each location. Formally:

$$P(x,y,z) = \left|\left| \frac{1}{M}\frac{1}{N}\sum_{j=1}^{M}\sum_{i=1}^{N} h_{i,j} e^{\frac{4\pi d_i(x,y,z)}{\lambda_j}} \right|\right| \tag{14}$$

where M is the number of frequencies used, $h_{i,j}$ is the channel measurement of the ith location with the jth frequency, and λ_j is the wavelength of jth frequency.

Impact of Headset Self-Tracking Error

As mentioned before, X-AR uses the headset's built-in self-tracking to enable AR-based SAR and localize RFID tags in the environment. One important question is whether the accuracy of the Microsoft Hololens' self-tracking is sufficient to support SAR and accurate RFID localization, especially over a random human walking trajectory. Prior reports have evaluated the accuracy of Hololens self-tracking [38] showing an average error of 0.56 cm.

To investigate the impact of self-tracking error, we simulated X-AR's SAR-based RFID localization and added an average of 0.56cm self-tracking error into our simulation. We compared this to a simulation of SAR with an ideal self-tracking system (i.e., 0 cm self tracking error). Fig. 11 plots a CDF of the L2 norm of the simulated localization error for a headset with ideal tracking (orange) and for a headset with simulated self-tracking error (purple). We make the following remarks:

- When simulating the Hololens with self-tracking error, X-AR is able to achieve a median of 8.1 cm. This high accuracy demonstrates that the self-tracking accuracy of the Hololens is sufficient for SAR-based localization.
- The simulated localization accuracy is close to the empirical evaluation (9.8 cm in §7.1).

[11] Note that the exponent contains an extra multiple of 2, since the signal travels $2d_i$ from the antenna to the tag and back to the antenna.

- When simulating an ideal headset (with no tracking error), the median localization error is 2.7 cm. This implies that as the AR headset self-tracking technology evolves, the performance of X-AR in localizing RFID tagged target items will further improve (albeit, it's not clear whether 2.7cm would yield meaningful UI/UX improvements for our use-cases on top of the 8.1cm accuracy).

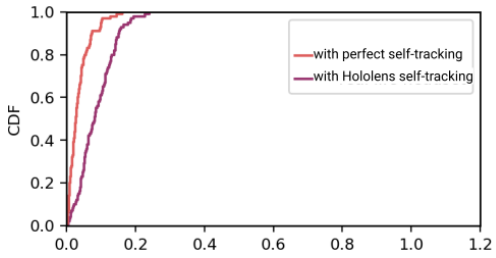


Figure 11: **Impact of Headset Self-Tracking Error.** CDF of simulated L2 norm of RFID localization error for a headset with an ideal self tracking module (orange) and for a headset with simulated self-tracking error (purple)

Acoustic Sensing and Communication Using Metasurface

Yongzhao Zhang[+], Yezhou Wang[+], Lanqing Yang[+], Mei Wang[†]
Yi-Chao Chen[+*], Lili Qiu[†*], Yihong Liu[♮], Guangtao Xue[+], Jiadi Yu[+]
[+]*Shanghai Jiao Tong University,* [†]*UT Austin,* [*]*Microsoft Research Asia,* [♮]*University of Glasgow*

Abstract

Acoustic sensing is increasingly popular owing to widely available devices that support them. Yet the sensing resolution and range are still limited due to limited bandwidth and sharp decay in the signal at inaudible frequencies. Inspired by recent development in acoustic metasurfaces, in this paper, we first perform an in-depth study of acoustic metasurface (AMS) and compare it with the phased array speaker. Our results show that AMS is attractive as it achieves a significant SNR increase while maintaining a compact size. A major limitation of existing AMS is its static configuration. Since our target may be at any possible location, it is important to support scanning in different directions. We develop a novel acoustic system that leverages a metasurface and a small number of speakers. We jointly optimize the configuration of metasurface and transmission signals from the speakers to achieve low-cost dynamic steering. Using a prototype implementation and extensive evaluation, we demonstrate its effectiveness in improving SNR, acoustic sensing accuracy, and acoustic communication reliability over a wide range of scenarios.

1 Introduction

Motivation: Acoustic sensing and communication are becoming increasingly popular due to widely available devices that support it, including smartphones, smart speakers, and many IoT devices. Many interesting sensing systems have been proposed using acoustic signals (e.g., [15, 25, 35–39, 42, 51, 57, 65]). For example, [35, 36, 42, 51, 57, 65] develop smartphone based approaches that transmit inaudible acoustic signals to track a target's distance, position, and movement. [38, 52] enables more accurate sensing by exploiting a microphone array on a smart speaker. [11] develops acoustic communication systems as an NFC alternative, [9] designs an underwater messaging system using acoustic signals since acoustic signals attenuate slower than RF signals. Refer to [5, 12] for more comprehensive surveys on the underwater acoustic communication systems. Despite significant advances in acoustic sensing, there is a fundamental limit on its sensing range and resolution as shown in the Cramer-Rao bound, which indicates the sensing resolution is limited by SNR and the number of transmitters and receivers. Similarly, acoustic communication also faces similar challenges according to the Shannon capacity.

In order to further improve the performance, one could increase the number of transceivers. However, increasing the number of transceivers increases the cost, size, and energy consumption. In addition, existing sound cards cannot support more than 8 channels. All of these factors significantly limit their applicability in a real-world deployment.

Another option is to adopt an acoustic lens (or acoustic metasurface, AMS). Like optical lenses, acoustic lenses can steer the direction of acoustic wave propagation and focus in a certain region. However, an acoustic metasurface is usually bulky due to the large wavelength of acoustic waves. Recently there emerged some metasurface quantization designs (e.g., [40, 41]). They comprise many sub-wavelength cells, where each cell can act like a mini-antenna and modify the phase and/or intensity of the incident wave so that collectively the AMS can manipulate the wave in an interesting way (e.g., steer the outgoing wave towards a certain direction).

Our approach: Inspired by the potential benefit of AMS, we first compare a passive AMS with beamforming using multiple speakers. We find beamforming using 3 and 6 speakers increases SNR by $4.7dB$ and $7.9dB$, respectively. In comparison, an acoustic metasurface of size 16×16 cells under 1 speaker increases SNR by $15.5dB$. The results suggest AMS is attractive since it can significantly increase the SNR using a compact design without consuming power. To achieve a similar SNR increase, we need 36 speakers spanning $30cm$, which is bulky and challenging to deploy.

While passive AMS is attractive, the existing AMS can support only static configuration (e.g., always beamform towards

Yongzhao Zhang, Yezhou Wang, Lanqing Yang, Yihong Liu did this work as interns at Microsoft Research Asia and Yi-Chao Chen did this work as a visiting researcher at Microsoft Research Asia.

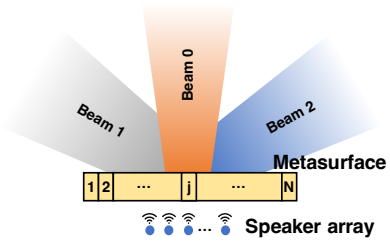


Figure 1: Dynamic beam steering with speaker array and acoustic metasurface (AMS).

a fixed angle). Since the target can be at any location, dynamic beam steering is necessary. One option is through the mechanical movement of the AMS, which not only increases the cost but also limits the speed of adaptation. In this paper, we propose combining beamforming using a small number of speakers with AMS, as shown in Figure 1, to achieve the best of both worlds: the use of AMS allows us to keep the number of speakers low while still achieving sharp beam, large SNR gain, and high resolution; the use of beamforming even using very few speakers can enable dynamic steering without movement. A small number of speakers with a passive AMS can achieve similar beamforming resolution as a large number of speakers. For example, our evaluation shows using an AMS with 16x16 cells and a 6-speaker phased array is comparable to a 9x16=144 phased array in terms of beam width.

To this end, we develop a novel algorithm to jointly optimize the AMS configuration and beamforming weights for the phase array. Specifically, the joint AMS and beamforming design can be formulated as an optimization problem whose objective is to maximize the signal strength along each of the desired angles (e.g., sampled from a range of angles) and minimize the performance variance across these angles and energy in the side lobes. We use the gradient projection method to solve the optimization problem. In addition, we augment our optimization framework to further optimize the speakers' placement and improve the performance.

Based on our designed algorithm, we implement an acoustic system that comprises a 3D-printed AMS, 6 speakers, and a microphone. We apply our algorithm to steer the outgoing beam in real time. We evaluate our design using (i) SNR of the received signal, (ii) sensing performance (i.e., distance estimation using Frequency Modulated continuous Waves (FMCW) and angle estimation using the MUltiple SIgnal Classification (MUSIC) algorithm), and (iii) the communication error.

Our contributions can be summarized as follow:

- Using extensive evaluation and analysis, we shed light on the benefits of phased array versus AMS.
- We jointly optimize AMS and phased array configurations to enable dynamic beam steering and high SNR.
- We further improve the performance by optimizing the speaker placement.
- We develop an acoustic system based on our joint design of AMS and beamforming and apply it to acoustic sensing and communication. Our evaluation shows our system yields a significant improvement in SNR, distance estimation, angle estimation, and communication reliability. In particular, leveraging AMS and phased array allows us to dynamically steer the beam to the desired direction and boost SNR by 18.4dB over a single speaker without AMS. The improved SNR in turn increases the acoustic sensing and communication ranges. Our approach increases the sensing range from 1.5m in a single speaker without AMS to 4m using 6-speaker with AMS; similarly, it increases the communication range from 0.8m to 3.9m.

Paper outline: We review existing work in Section 2, and introduce acoustic metasurface in Section 3. We describe our algorithm to jointly optimize AMS and speaker array system in Section 4. We present our simulation and testbed experiment results in Section 5. We discuss the limitation and future work in Section 6. We conclude in Section 7.

2 Related Work

Our work is closely related to wireless sensing, acoustic communication, acoustic metasurface, and phased array.

Wireless sensing: Wireless sensing has become increasingly popular due to many important applications. Many algorithms and systems have been developed recently using acoustic [15, 25, 30, 35, 42, 45, 51, 57, 64, 65, 67], WiFi (e.g., [21, 47, 50]), mmWave (e.g., [20, 58, 61]), and RFID signals (e.g., [17, 34, 53–55]).

Among them, acoustic sensing is appealing due to its high accuracy and widely available commodity devices that support it. They use time of flight (e.g., BeepBeep [45]), Doppler shift (e.g., AAMouse [64]), FMCW (e.g., [35,42]), phase (e.g., [57]), correlation (e.g., [43]), channel impulse response (e.g., Strata [65]), and Angle of Arrival (AoA) (e.g., [38, 49, 56]) for sensing. Some works also leverage machine learning for acoustic sensing (e.g., by applying neural networks to either post-processed signals or raw signals) and show ML based sensing is promising.

There are also significant works on sensing using RF signals. Some leverage similar algorithms as in acoustic sensing, while others explore new features and algorithms. For example, [33] use Channel State Information (CSI). Tagyro [59] tracks rotation using an array of passive RFID tags and two orthogonal RFID reader antennas. [46] further exploits polarization to track rotation and translation movement. [22] pushes the tracking accuracy to sub-centimeter level using a large phased array and large bandwidth.

Acoustic Communication: Sound has been a popular way of communicating information. Interestingly, we can also encode and transmit digital data over the acoustic channel. At a high level, it is essentially the same as RF communication but uses a different frequency. A number of interesting systems have been developed for acoustic digital communication (e.g., [11, 23, 66]) as an alternative to Near Field Communication (NFC) owing to the wide availability of speakers and microphones that support acoustic communication. Many of them leverage OFDM due to its robustness to multipath fading. Therefore, we also adopt the OFDM based acoustic communication in our work.

Acoustic metasurface: Ultimately, the accuracy of acoustic sensing depends on the SNR and numbers of speakers and microphones. Acoustic metasurface can boost SNR using a passive 2D structure, which can help improve sensing performance. A metasurface has many unit cells, and each cell can be potentially treated as a mini sound source. In this way, AMS effectively increases the number of speakers, thereby improving sensing resolution. By controlling the phase and/or amplitude of acoustic wave propagation through each unit cell, AMS can manipulate the wave fields. Many designs of acoustic metasurface have been proposed in the literature [6, 32]. Coiling-up space structure [27–29, 40, 41, 60] achieves phase manipulation by forcing acoustic waves to propagate along a coiled path. Helmholtz-resonator-like structure [13, 26] produces a tunable phase velocity and a high transmission efficiency with multiple Helmholtz resonators. Membrane-type structure [14, 19, 44, 62, 63] eliminates reflection with carefully designed membrane resonators. The above designs are not reconfigurable. There are active acoustic metasurfaces [10, 18, 24], as well. They use mechanical structure or emerging materials that can be deformed under the control of a magnetic field or electric current. However, these designs are expensive and bulky to implement.

Our work is inspired by [40, 41]. [40] develops a powerful methodology that assembles many sub-wavelength pre-manufactured 3D units into an acoustic metasurface. Each unit encodes a specific phase offset. By re-arranging these units, one can produce many different metasurfaces. Since acoustic sensing/communication usually use inaudible sounds with much smaller wavelength to avoid disturbance, the coiling-up metasurface is more compact than Helmholtz-resonator and membrane-type structure. [41] discusses several applications of these metasurfaces, including generating acoustic collimator, acoustic magnifying glasses, and acoustic telescopes. Our work goes beyond [40, 41] by enabling dynamic steering through combining multiple speakers with AMS and applying it to acoustic sensing and digital communication.

[16] proposes using 3D-printed metamaterial to cover the microphone and embed the direction-based signature. During the calibration stage, recordings from all possible angles are collected. During the online usage, the current recording is compared with all recordings collected in the calibration to find the best match, which is used for AoA estimation. [7] develops an acoustic sensing system that uses 3D printed smart surface to embed direction information into the signals for generating a depth map. Our work is related to the above work but goes beyond them by (i) eliminating labor-intensive calibration and (ii) directly increasing SNR and sensing resolution, which can benefit any sensing or communication approaches instead of tailing to one specific sensing scheme. Therefore our design is more general and support more applications. Moreover, our optimization framework for configuring AMS and a speaker array is flexible and can support a range of important what-if analyses. Our adoption of a regular shaped AMS also makes it easier to analyze and optimize its impact on the overall system performance.

Phased array: Multiple transmitters and/or multiple receivers can be used to strengthen the received signals. At the transmitter end, beamforming can be used to generate transmissions that arrive in phase at the receiver so that the multipath signals are added up constructively. At the receiver end, the receiver can compensate for the phase difference of the received signals across different antennas to ensure constructive combining. As mentioned earlier, in order to achieve a comparable gain of AMS, a large phased array is necessary, which increases the size, cost, computation, and power. This motivates our design of AMS based sensing and communication system.

3 Acoustic Metasurface

In this section, We provide background on acoustic metasurface and its properties.

3.1 Background of Acoustic Metasurface

The ability to shape acoustic fields has diverse applications, such as high-quality sound production, particle manipulation, non-invasive therapies, and increasing sensing and communication range and resolution. One way to shape the acoustic fields is to use phased arrays by controlling the phase and amplitude of the transmission signals emitted from each of the speakers. The cost, power consumption, and size of a phased array rapidly increase with the number of speakers.

A few recent research papers show that acoustic metasurfaces could be a promising solution. An acoustic metasurface is a 2D structure that consists of many sub-wavelength cells [40, 41]. By carefully designing each of its cells, we can manipulate acoustic waves in an interesting way. Each unit cell can be viewed as a mini sound source. To perform beamforming in a certain direction, we can ensure the paths going through different cells in the metasurface add up constructively in the desired direction. This can be achieved by

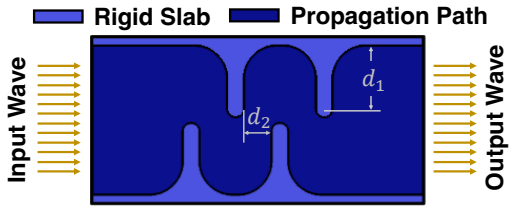


Figure 2: The structure of a unit cell is mainly determined by the parameters d_1 and d_2. Different lengths of the propagation paths induce different phase delay at the output.

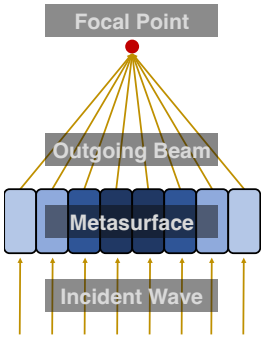
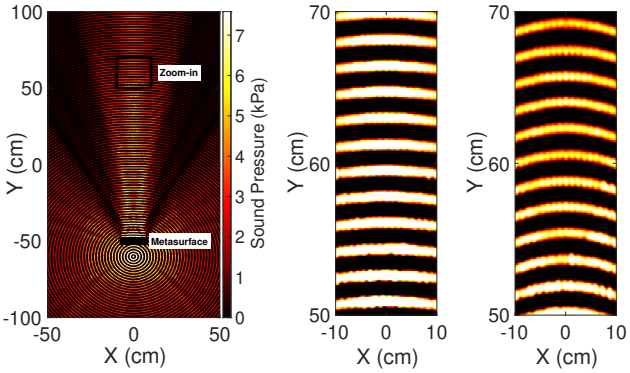


Figure 3: A metasurface consists of multiple unit cells and beamforms towards a focal point by properly configuring the unit cells.

letting each unit cell compensate for the phase difference. For example, without special design, in the desired direction, the path going through cell 1 differs from cell 2 by $\phi_{1,2}$. To ensure the signals from these two paths add up constructively, we can design the unit cell 1 and cell 2 to compensate phase difference $\phi_{1,2}$. One way to achieve this is to impose different geometric structure so that the path going through cell 2 is $\frac{\phi_{1,2}}{2\pi}\lambda$ longer than cell 1.

Figure 2 shows an example structure of unit cells used in [40]. Assume the sound waves pass through the unit cell from the left. The curved propagation path will increase the time it takes to penetrate the unit cell, which essentially introduces a phase shift to the outgoing wave. The structures of different unit cells are determined by two dominant parameters: d_1 and d_2 [40], which result in different propagation path lengths and hence different phase delay. One way to determine d_1 and d_2 is through enumeration in a simulator (e.g., COMSOL [2], which is a widely used finite-element-based multiphysics simulator).

We arrange the unit cells in a straight line to form a 1D metasurface, or in a rectangle to form a 2D metasurface. By introducing an appropriate phase shift at each cell, we can achieve beamforming. Figure 3 shows an example. To make it easy to assemble/re-assemble a metasurface, [40] quantizes the types of unit cells into 16 choices, which covers the phase shift from 0 to 2π. So for each unit cell, we can choose one



(a) Sound field (b) 100mm (c) 200mm

Figure 4: (a) Sound field simulated in COMSOL with a 16×1 metasurface when we transmit $20kHz$ sound at the focal point ($100mm$). Part of the energy is concentrated in a specific direction. (b) and (c) show the sound field in the zoom-in area when the speaker is placed at $100mm$ and $200mm$, respectively. When the speaker is at the focal point, the signal coming out of the metasurface is a parallel wave.

whose phase shift is the closest to our desired shift.

3.2 Properties of Acoustic Metasurface

High transmission efficiency: As shown in Figure 2, the unit cells have intricate maze-like internal structures, with four parallel bars positioned orthogonal to the direction of incoming sound waves. Interestingly, the transmission efficiency is high and reaches 98% on average across all unit cells [40]. This is due to the following two major reasons: i) The sub-wavelength cells produce diffraction and cause the energy of sound to bypass the parallel bars instead of being reflected back; and ii) The bars inside each unit cell are curved instead of sharp angles to reduce acoustic impedance and maintain high transmission efficiency. Overall, the acoustic metasurface has negligible power loss, so we do not consider the power loss when developing AMS.

Focusing behavior: Figure 3 shows that an incident plane wave is focused at a focal point after passing through the metasurface. Due to the reciprocity principle [8], when a point source is placed at the focal point, the signal coming out of the AMS should be a plane wave towards the direction orthogonal to the metasurface. Figure 4(a) and 4(b) show an example scenario, where the source is placed at $100mm$, which is the focal point. We observe the outgoing wave is nearly parallel in one direction. Second, when the source is not at a focal point, the wave is no longer parallel, as shown in Figure 4(c), and the energy of the signal will be dispersed to nearby directions, making signal strength attenuates faster. Since we want to concentrate the energy in one direction and make the sound wave propagate in a longer range, plane waves are preferred.

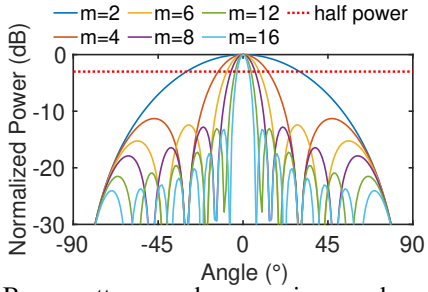


Figure 5: Beam patterns under a varying number of speakers m. The received sound is normalized by dividing it by the maximum power and then converted to decibels.

Adaptation: Once a metasurface is printed, the mapping from the incoming wave to the outgoing wave is fixed. Since the target can be in any direction, it is important to change the direction of the outgoing wave. Given the fixed metasurface, one way to change the direction of the outgoing wave is to move the AMS either through translation movement or through rotation. While movement is feasible, mechanical movement is slow, consumes significant power, causes wear and tear, and may even require operator intervention. Therefore, in this paper we seek a software-based approach to realize fast dynamic adaptation.

4 Phased Array with Metasurface

The passive acoustic metasurface is a fixed 2D structure. Once designed, it converts from the incident wave to the outgoing wave in a fixed manner. For practical use, it is desirable to dynamically adjust the direction of the wave coming out of the metasurface. We can achieve this using either mechanical movement or beamforming using a phased array. We take the latter approach due to its software control and eliminating the need of movement. An important question is how to configure the metasurface and phased array to realize our desired beamforming. Below we first introduce phased array and then describe how to use phased array with metasurface to achieve dynamic control at a low cost.

4.1 Phased Array

Phased arrays use beamforming to combine signals from multiple speakers constructively. Beamforming can be applied to either senders or receivers or both. There are a number of beamforming algorithms. They vary in the optimization objectives: some maximize the signal, while others minimize interference.

In analog beamforming, beamforming is performed on analog signals at the transmitter before sending to the air or at the receiver before the analog to digital conversion. In digital beamforming, beamforming is performed on digital signals at the transmitter before digital to analog conversion or at the receiver after analog to digital conversion.

The beamforming capability depends significantly on the number of speakers and their separation. As Figure 5 shows, the beam width in the desired direction is relatively large and the sidelobes are significant when the number of speakers is within 8. The half power beam width (HPBW) at 0° (i.e., perpendicular to the speaker array) can be approximated as follows [48]: $\theta_{0.5} \approx \frac{0.886\lambda}{md}$ where λ is the wave length, m is the number of speakers, and d represents the speaker separation, which is usually recommended to be $\frac{\lambda}{2}$. For example, the HPBW will be 59.6°, 25.5°, 16.9°, and 6.3° when the number of speakers is 2, 4, 6, and 16, respectively. The beam width for a general angle can be derived as follows: $\theta_{0.5s} = \frac{\theta_{0.5}}{\cos\theta_s}$ where θ_s is the steering angle and $\theta_{0.5s}$ is the HPBW of the steered beam. This indicates that the scanning range should not be too large and usually we let $\theta_s \leq 60°$. These results show that the acoustic beamforming resolution using a small phased array is limited.

4.2 Phased Array Coupled with Metasurface

Passive AMS is not reconfigurable on-the-fly once it is assembled. To provide dynamic adaptation while achieving high resolution and long range, we propose using a small number of speakers along with an acoustic metasurface. We optimize the speakers' beamforming so that the outgoing wave from the AMS is towards our desired angle.

More specifically, phased array can control the direction of the output signal, which serves as the incoming signal towards the AMS. By optimizing the transmission signals, we can potentially generate any shaped waves coming out of the AMS. The use of multiple speakers allows us to achieve fast dynamic control without movement. In order to fully realize this capability, we should carefully design the AMS to cover a wide range of angles and control the transmission signals from multiple speakers in order to dynamically generate the desired signal coming out of the AMS. Below we first formulate the problem and then present our solution.

4.2.1 Problem Formulation

As shown in Figure 1, there are M speakers. Let w_i denote the codeword for the i-th speaker, where w_i is a complex number whose magnitude and phase are the scaling factor and phase shift for the i-th transmission signal, respectively. There are N unit cells in AMS. The acoustic signal received by the j-th AMS cell from the i-th speaker $S_{i,j}$ can be computed as follow, where t_i is the i-th speaker's transmission signal and H_{ij} denotes the channel between the i-th speaker and j-th cell.

$$S_{i,j} = H_{ij} w_i t_i \tag{1}$$

Since the relative position between the AMS cell j and transmitter i is pre-determined, we can derive $H_{i,j} = F(d_{i,j}) = a(d_{i,j}) e^{-j2\pi f \frac{d_{i,j}}{c}}$, where c is speed of acoustic signal, $d_{i,j}$ is

the distance from the i-th transmitter to j-th cell, $a(d_{i,j})$ is the amount of signal attenuation at the distance $d_{i,j}$, and $F(\cdot)$ is a function that models how the channel attenuates with the distance $d_{i,j}$.

We can take the placement of the phased array into account, denoted as x, and re-write the above relationship in a matrix form as follows:

$$S_{in} = H(x)w \tag{2}$$

We omit the transmission signal t_i before beamforming hereafter because it is the same at each speaker.

Each cell in AMS modifies the incident signal (e.g., by adding a path delay and/or changing the amplitude). Such a modification can be captured using a matrix, denoted as G, and the design of G will be covered in Sec 4.2.3. Then the signal coming out of the AMS becomes

$$S_{out} = GH(x)w \tag{3}$$

Finally, let R_d denote the signal in a given steering direction d from the AMS. R_d can be derived as follow, where K_d denotes the steering vector corresponding to the direction d between the AMS and target.

$$R_d = K_d GH(x)w \tag{4}$$

Our goal is to design the AMS and codeword of the speaker array to maximize the signal strength along each angle of interest. For example, if we want to support a scanning angle from -60° to 60°, for each angle within the range, we want to maximize the signal strength. Note that the AMS has a fixed configuration across all angles, while the codeword can change for each beamforming angle as in typical beamforming scenarios. Therefore, the signal of interest R can be derived as follow:

$$R = KGH(x)W \tag{5}$$

where R is a matrix of size $d \times d$ (each row represents the received signal from a given direction d and each column represents the steering direction), K is a $d \times N$ matrix specifying the steering vector from the N unit-cell AMS, G is an $N \times N$ matrix and its diagonal elements specify how the N-cell AMS translates the incident signal into outgoing signal, $H(x)$ is an $N \times M$ matrix specifying the channel from M transmitters to N-cell AMS, and W is $M \times d$ codebook for M speakers corresponding to d directions.

The channel H and steering vector K are fixed and can be derived analytically. Given H and K, we want to find the optimal static AMS configuration G and codebook W to perform beamforming across a wide range of angles. Since we optimize the power of the beams in each direction, we use the power P of the received signal R hereafter, which is denoted by $P = |R|^2$.

The structure of our received signals P can be visualized in Figure 6, where we aim to have high signal strength along the diagonal elements, which indicates our signal is beamformed towards the desired steering angle.

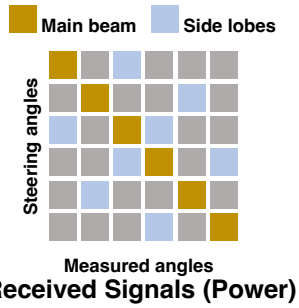


Figure 6: The structure of the received signal at various angles.

Our objective function comprises the following 3 terms:

Sum Power: Due to the use of a static metasurface design and the need to accommodate a wide range of angles, our goal is to maximize the sum of power across all d directions. This can be derived as follow:

$$L_{power} = \text{tr}(P) \tag{6}$$

where $tr(\cdot)$ is the trace of a matrix (i.e., the sum of the diagonal elements in the trace). This is shown in Figure 6.

Minimum Variance Criterion: Solely maximizing the total power may introduce some dead zones for certain directions. To avoid that problem, we add the variance of P's diagonal as the penalty term L_{var} to ensure all directions are covered:

$$L_{var} = \text{var}(\text{diag}(P)). \tag{7}$$

For generality, we introduce a weight matrix Q, which can put different weights on different angles. As a result, we have:

$$L_{var} = \text{var}(\text{diag}(PQ)) \tag{8}$$

where $Q = \text{diag}(q_1, q_2, \ldots, q_l)$ is a set of weights to control. If we have prior knowledge about the target's (approximate) location, we can increase the entries in Q that correspond to the locations close to the target.

Minimum Sidelobe: Suppressing the side lobes is critical for sensing and communication. In the signal processing literature, extensive works have been done to control sidelobe level (SLL). Sidelobe nullification and minimization are two common methods. Some methods require prior knowledge about the sidelobe's direction, while other methods minimize the maximum sidelobe. We experiment different ways of suppressing sidelobes and find minimizing the average SLL (i.e., minimize the sum of absolute values of all non-diagonal peaks in P) is most effective in our context.

We observe that the non-diagonal peaks shown in Figure 6 are considered as sidelobes and can degrade the overall performance. Therefore, we propose to minimize the sum of non-diagonal peaks as follows:

$$L_{sidelobe} = \sum \text{non-diagonal peaks} \tag{9}$$

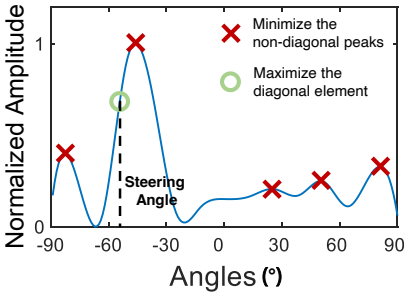


Figure 7: Beam pattern when steered to a specific angle during optimization. Minimizing the non-diagonal peaks helps reduce the side lobes and increase the directivity.

To derive $L_{sidelobe}$, we identify peaks in the P matrix (e.g., using findpeaks() function) and then sum up the peaks that are in non-diagonal entries of the matrix. It not only reduces the sidelobes, but also improves the quality of the main lobe. As shown in Figure 7, if the highest peak is a non-diagonal element, we also minimize it to revise the direction.

Putting together, we have the following optimization model:

$$\min_{W,\Theta,x} \; -L_{power} + \mu L_{var} + \gamma L_{sidelobe}$$

$$s.t. \quad \begin{cases} |G_{ii}| = 1, \ (i = 1,2,\ldots,N) \\ |W_{ij}| \leq 1 \ (i = 1,2,\ldots,M, \ j = 1,2,\ldots,d). \end{cases}$$

where μ and γ are parameters controlling the importance of the variance and sidelobe terms, respectively. We have two constraints on the magnitude of the metasurface parameters G and codebook W. Both G and W should be no more than 1.

The constraints on the magnitude of metasurface G are called constant modulus constraints (CMC). It is well-known that problems involving CMC are nonconvex and NP-hard [31]. $|G_{ii}| = 1$ refers to the points on the surface of an N dimensional hypercube, which indicates each metasurface cell does not change the magnitude of the incoming signal. These are non-convex constraints. $|W_{ij}| \leq 1$ are constraints on the magnitude of the phased array. The set contains the entire hypercube and includes the interior. Thus, it is a convex set. Therefore, for the phased array codebook, we restrict the amplitude to be within 1 instead of exactly equal to 1 to make the problem easier to solve.

4.2.2 Optimization

Our problem is a non-linear constrained optimization problem. Due to the presence of the constraints, we cannot directly apply the gradient descent scheme. Therefore, we use the gradient projection method, which ensures the solution after each gradient descent update still falls within the feasible set Ω. Specifically, if the $k+1$-th update (i.e., $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$) makes the solution fall outside the feasible region, where α_k is the learning rate and $d^{(k)}$ is the gradient, we project it to a point inside the feasible set Ω as follows:

$$x^{(k+1)} = \Pi[x^{(k)} + \alpha_k d^{(k)}] \tag{10}$$

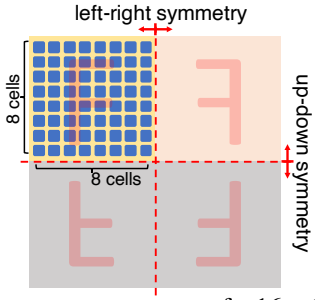


Figure 8: The symmetry property of a 16×16 metasurface.

where Π is projection operator, and $\Pi[x]$ is called the projection of x in Ω. To do that, we normalize the amplitude of G_{ii} after each update and normalize W_{ij} if it is larger than 1. As a result, we use Adam optimizer in Pytorch for optimization. Adam is an extended version of stochastic gradient descent that adapts the learning rate for each parameter. We modify the output from the Adam during each iteration using Equation 10 to ensure constraints are satisfied.

4.2.3 Additional Design Details

In this section, we describe how to get the input required for optimization.

Symmetry Property of AMS As mentioned earlier, the diagonal of variable G should represent the phase delay for metasurface cells. Our metasurface is a 2D structure. We observe that the configuration of the AMS should be left-right symmetric and up-down symmetric, as shown in Figure 8, since the scanning performance should be the same in left and right in the azimuth direction and the beam pattern should also be the same in top and bottom in the elevation direction. By utilizing the left-right and up-down symmetry property, we can reduce the search dimension for G by 75%.

Codebook Since the range of the steering angle is from $-60°$ to $60°$, the codebook is also symmetric between the positive angles and negative angles. Therefore, we can optimize half of the codebook (i.e., corresponding to the steering angle in $(-60°, 0)$) and copy them to generate the codebook for $(0, 60°)$.

Channel From Phased Array to Metasurface The channel $H(x)$ can be determined based on the speakers and metasurface cells' positions. Let $x = \{x_1, x_2, \ldots, x_M\}$ denote the speakers' locations, and $g = \{g_1, g_2, \ldots, g_N\}$ denote the metasurface cells' locations. We can derive the channel as follows:

$$H(x) = \begin{bmatrix} F(\|x_1 - g_1\|) & \ldots & F(\|x_M - g_1\|) \\ F(\|x_1 - g_2\|) & \ldots & F(\|x_M - g_2\|) \\ \vdots & \ddots & \vdots \\ F(\|x_1 - g_N\|) & \ldots & F(\|x_M - g_N\|) \end{bmatrix} \tag{11}$$

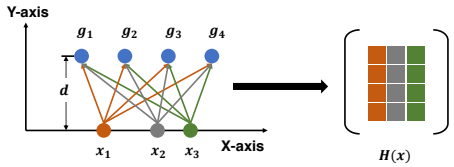


Figure 9: Derive channel matrix H with varying speaker distributions x. Consider 3 speakers and 4 metasurface cells as an example. We can derive the distance between each speaker and metasurface cell to get the channel H between the phased array and metasurface.

where $\|\cdot\|$ denotes the distance between two points (i.e., a speaker and a metasurface cell) and $F()$ denotes the function that maps the distance to the wireless channel, including the amplitude and phase. Figure 9 shows an example.

We can either (i) take a given phased array setup (e.g., uniformly distributed linear array) as the input or (ii) optimize the phased array setup. In the latter case, we treat x_i as the optimization variables along with the other variables. Note that we do not impose any constraints on x because we already consider the symmetry property of metasurface G and codebook W. Equation 11 assumes a single line-of-sight path between the phased array and metasurface, which is realistic since the metasurface is close to the phased array and there is no blockage.

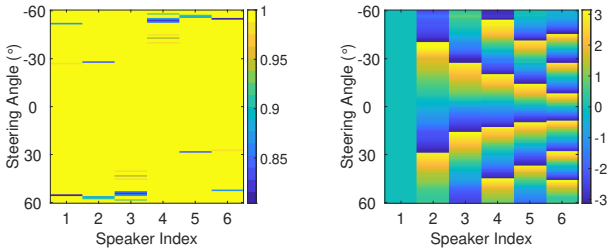
4.2.4 System Design

In this section, we provide further details of our system design, including the AMS design, codebook design, and array placement.

We sample angles from $-60°$ to $60°$ with $1°$ apart. Therefore, for a 6-speaker system, the codebook W is a 121×6 matrix, which contains 121 independent codewords for 121 directions and 6 speakers. Figure 10 shows the amplitude and phase of the optimized codebook, where the first speaker is set as the reference and is aligned to be zero phase. Since our goal is to maximize the sum power of diagonal elements, the amplitude of each element in the codebook is 1 to achieve the maximum transmission power, while the phase is manipulated to generate our desired sound field at the metasurface.

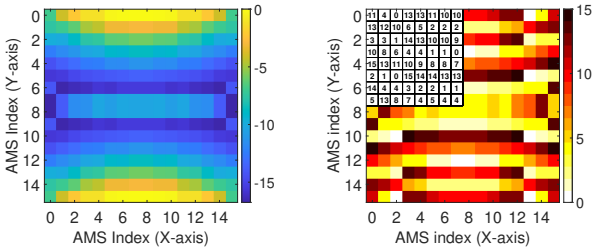
Next, we reconstruct the phase distribution of the metasurface by utilizing the diagonal elements of G and the symmetry property. The results are given in Figure 11(a). This is different from that of [40] due to the presence of a phased array. As mentioned in Sec. 3, the phase shift of each AMS cell is quantized to 16 levels for flexible design and assembly/disassembly. Then the final AMS can be assembled by choosing the unit cells with the closest phase shift, as shown in Figure 11(b), where the color reflects the unit cell index and a higher index indicates a larger phase shift.

We can either (i) place speakers in the phased array uniformly or arbitrarily and feed the placement to our optimization algorithm or (ii) let our algorithm optimize the placement



(a) Magnitude of codebook. (b) Phase of codebook.

Figure 10: The optimized codebook design, where the first speaker is set as the reference. The magnitudes of the codebook are all close to 1, but some are slightly less than 1 since their constraints are ≤ 1.



(a) Phase distribution. (b) Cell index.

Figure 11: Phase distribution and cell indices of the optimized 16×16 metasurface design, where the cell index is the index to one of the 16 quantized phase shifts. The numbers in the upper left corner denote the cell indices for the top left metasurface and we omit the other parts for brevity due to the left-right symmetry and top-bottom symmetry.

along with other configuration parameters. In evaluation, we compare uniform placement and optimized placement.

5 Performance Evaluation

In this section, we first present our evaluation methodology and then describe performance results.

5.1 Evaluation Methodology

We use the experiment setup shown in Figure 12 for our evaluation. The system can be divided into three parts: speakers, microphones, and an acoustic metasurface (also referred to as an acoustic lens or AMS). We use uniform placement as the default configuration. In this case, we have 6 identical miniature speakers (16Ω, $0.25W$) as the transmitter. Each speaker is connected with an operational amplifier THS4001 [4] to amplify the voltage and a power amplifier LM386 [3] to amplify the current. The distance between the centers of adjacent speakers is $8.6mm$, which is a half wavelength of $20kHz$ sound. We used 4 microphones to form a microphone array as a receiver. The distances between the 4 microphones were $3.06cm$, $2.04cm$, and $3.06cm$ to reduce ambiguity and obtain better performance [38]. All speakers and microphones are

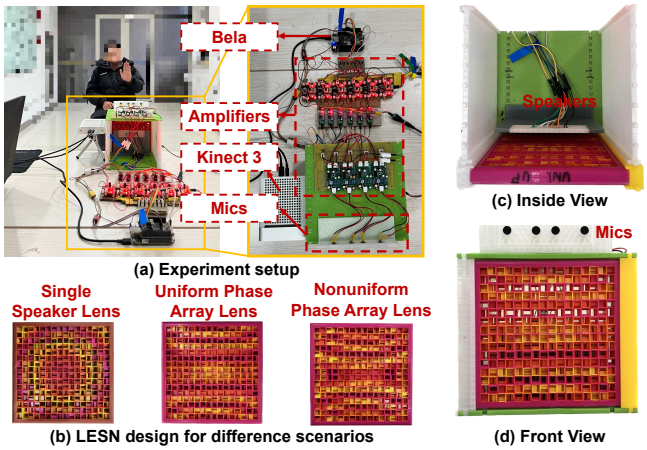


Figure 12: System setup.

connected to the same Bela board [1] for signal synchronization. We also optimize speaker placement using the approach described in Section 4.2.

We construct an acoustic lens according to our optimization in Section 4.2.2. Our lens consists of 256 (16×16) unit cells, spanning over $15cm \times 15cm$. Since the unit cells are quantized into 16 choices, we 3D print 16 different types of unit cells and assemble them to an acoustic lens according to the evaluation scenario. For example, we assemble an acoustic lens for a 1-speaker setup, a different acoustic lens for 6 speakers with uniform separation, and another one for 6 speakers with non-uniform separation, as we jointly design the metasurface with the speaker array. To ensure most signals coming out of the speakers go through the acoustic lens, we place our lens $2cm$ away from the speaker array. For a single speaker, we follow the setup in [41] where the lens is placed $10cm$ away from the speaker array.

We evaluate our approach in terms of (i) SNR, (ii) sensing accuracy, and (iii) communication performance. For acoustic sensing, we use Kinect V3 to get the ground truth distance and angle of arrival (AoA). We let the speakers transmit the following FMCW signal: $tx(t) = cos(2\pi f_{min} t + \frac{\pi B t^2}{T})$, where $f_{min} = 16kHz$, $B = 4kHz$, and $T = 0.1s$. We quantify the sensing accuracy using distance error and angle of arrival error.

1D MUSIC is a widely used AoA estimation algorithm. It computes the auto-correlation matrix R of the received signals x as $R = x^H x$, where x is a $1 \times N$ vector and x^H is the conjugate transpose of x, and then performs eigenvalue decomposition on R. Let R_N represent the noise space matrix, which is the space spanned by the $N - M$ smallest eigenvectors, where M is the number of signals. The peak in the pseudo spectrum $p(\theta) = \frac{1}{a(\theta)^H R_N R_N^H a(\theta)}$ corresponds to the AoA.

For acoustic communication, we encode the data using OFDM. Each OFDM frame contains 180 BPSK symbols, which are striped onto 12 subcarriers spanning over $18kHz$-$20kHz$. We use CDMA as FEC code to improve resilience and the code rate is 50%. We quantify the communication performance using bit error rate (BER) and frame error rate (FER). While there are other coding schemes for acoustic communication, the benefit of our approach (i.e., acoustic lens with a speaker array) is likely similar across different acoustic coding schemes.

Unless otherwise specified, all results are from **testbed experiments**: we use a 6-speaker array with an equal separation of $9.4mm$ between the two adjacent speakers and 16×16 acoustic lens; in device-free acoustic sensing experiments, the microphone array is $3cm$ above the acoustic lens to track the distance and AoA of a person's hand so that the signal from the speaker to the target goes through the metasurface and the signal reflected from the target and received by the microphone array does not go through the metasurface; in acoustic communication experiments, the receiver is at $1.5m$ away from the speaker array. We also evaluate the impact of various parameters by varying their values.

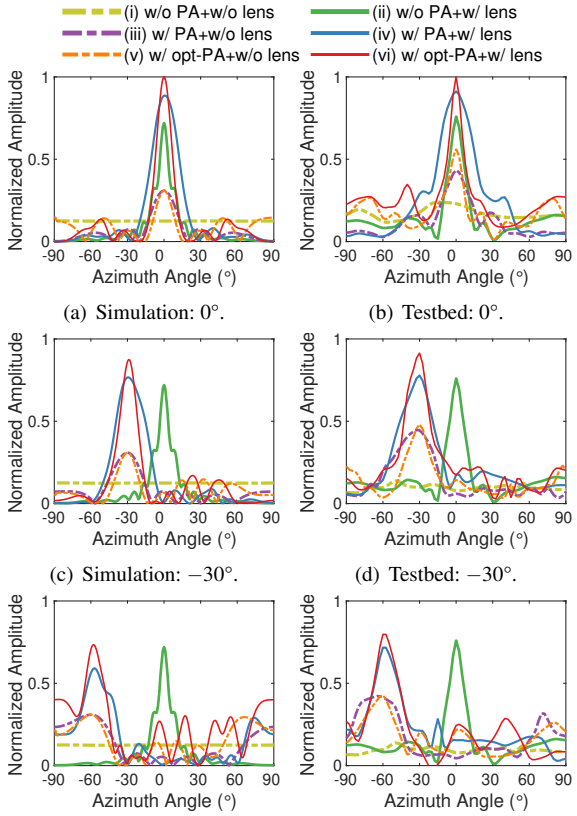
5.2 SNR Comparison

We first compare various schemes in terms of SNR.

5.2.1 Beam Pattern

We place a receiver at $1.5m$ away, 0° from the speaker(s) and measure the sound field intensity. Figure 13(a)(c)(e) compare the beam patterns of six schemes in COMSOL simulation [2], and Figure 13(b)(d)(f) compare them in testbed. The six schemes include: (i) a single speaker without lens (**w/o PA + w/o lens**), (ii) a single speaker with a lens (**w/o PA + w/ lens**), (iii) a phased array without lens (**w/ PA + w/o lens**), (iv) a phased array with a lens (**w/ PA + w/ lens**), (v) an optimized phased array without lens (**w/ opt-PA + w/o lens**), and (vi) an optimized phased array with a lens (**w/ opt-PA + w/ lens**). Our goal is to focus the transmission signal in any desired direction. As we can see, (vi) yields the highest peak in the desired angle, which is 1.2, 2.9, 10.5, 10.5, and $18.4dB$ higher than (iv), (ii), (v), (iii), and (i), respectively. Optimized array placement yields $1.2dB$ gain over uniform placement. Leveraging acoustic lens yields $15.5dB$ gain when applied to a single lens, but its angle cannot be adapted and is always fixed at 0°. Combining a phased array having a uniform separation with an acoustic lens allows us to focus the beam in the desired direction while achieving $17.2dB$ gain over a single speaker without lens and $9.3dB$ gain over a phased array without lens.

We further evaluate the signal strength and vary the steering angle, as shown in Figure 14. The improvement of (vi) over (iv) shows the benefit of the optimized array placement, and the improvement of (vi) over the other schemes shows the benefit of combining lens and phased array in the optimized placement. These results show that (vi) yields a high SNR gain across a wide angle from $-60°$ to 60°.



(a) Simulation: 0°. (b) Testbed: 0°.

(c) Simulation: −30°. (d) Testbed: −30°.

(e) Simulation: −60°. (f) Testbed: −60°.

Figure 13: The amplitude of the acoustic signal at various angles of steering. The amplitude is normalized by dividing it by the maximum amplitude of the signal received at 0°. (a)(c)(e) show the results from the simulation using COMSOL and (b)(d)(f) show the results from the testbed.

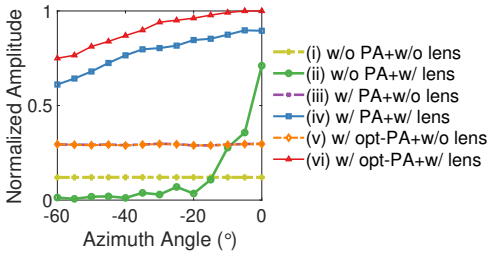
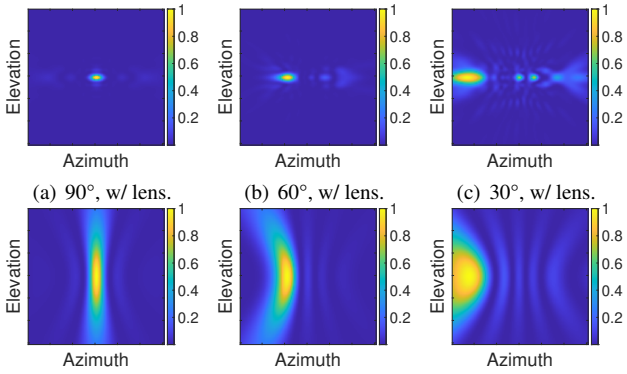


Figure 14: Power gain while steering at various angles. The amplitude is normalized by dividing it by the maximum amplitude of scheme (vi). The testbed uses a 16×16 acoustic lens and 6 speakers to form a phased array.



(a) 90°, w/ lens. (b) 60°, w/ lens. (c) 30°, w/ lens.

(d) 90°, w/o lens. (e) 60°, w/o lens. (f) 30°, w/o lens.

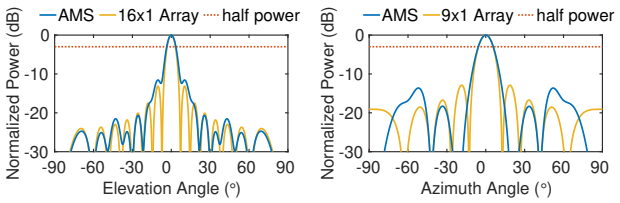
Figure 15: In the COMSOL simulation, comparing the sound signal strength using a phased array (6 speakers) with and without the acoustic lens in the azimuth direction.

5.2.2 Beam Width

The 2D structure of our lens design allows us to focus beams in both the azimuth and elevation directions. To show the impact of the beam width in both directions, we measure the sound field in a far field plane, which is parallel to the surface of the lens. Figure 15 plots the sound field of three different beams steered to 30°, 60°, and 90°, respectively. As we can see, the linear phased array can only focus beams in the azimuth direction. In comparison, the acoustic lens can focus beams in both the azimuth and elevation directions. According to COMSOL simulation shown in Figure 16, the acoustic lens with a 6-speaker phased array generates a comparable beam pattern to a 16×1 array in the elevation direction and a comparable beam pattern to a 9×1 array in the azimuth direction. Therefore, the acoustic lens with a 6-speaker phased array is comparable to a $9 \times 16 = 144$ phased array in terms of beam width. This is a significant reduction in cost, size, energy, and computation. Moreover, we also find that equipping a 6-speaker phased array with acoustic lenses of sizes 32×32, 48×48, and 64×64 yield similar beam patterns to 16×28, 20×40, and 30×48 phased arrays, respectively.

5.2.3 Frequency Response

Our acoustic metasurfaces are designed for $20kHz$ sound, but we use 16-$20kHz$ and 18-$20kHz$ for acoustic sensing and communication, respectively. To understand how the lens works at a different frequency, we test the frequency response of the lens. We first calibrate the speaker(s) and microphone(s) and use compensation to generate close to a flat frequency response in the received signal. We then transmit a sine wave with a frequency varying from $10kHz$ to $22kHz$ and record the received sound at $1m$ and 0° from the speaker. We test all lens configurations, including a single-speaker lens, uniform phased-array lens, and non-uniform phased-array lens. Fig. 17 shows the frequency response. As expected, the peak of the lens frequency response is $20kHz$. It drops rapidly after $20kHz$. Fortunately, the frequency response remains stable in $14kHz$-$20kHz$, which means that we can use this band for sensing and communication.



(a) Elevation Dimension. (b) Azimuth Dimension.

Figure 16: Using a 16×16 acoustic lens with a 6-speaker phased array is comparable to a 9×16 phased array in terms of beam width in the elevation and azimuth direction according to COMSOL simulation.

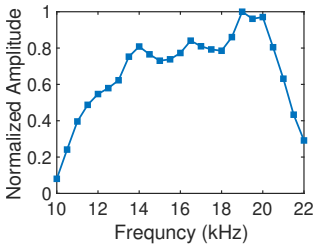


Figure 17: The frequency response of the acoustic lens.

5.2.4 Impact of Lens Size and Phased Array Size

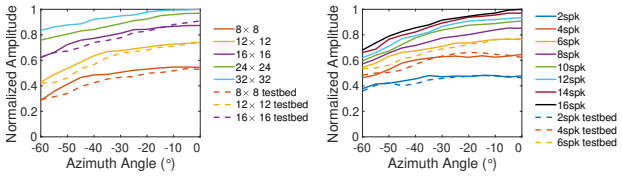
Figure 18(a) plots the received power of (vi) in the desired direction using different lens sizes. We only plot $-60°$ to $0°$ as the result from $0°$ to $60°$ is symmetric. We make several observations. First, as we would expect, increasing the lens sizes improves the normalized power (i.e., the received power is normalized by dividing by the power of 32×32 lens at $0°$ and $1.5m$ away). In particular, the normalized power increases from 0.29 using 8×8 lens to 0.42 using 12×12 lens, to 0.62 using 16×16 lens, to 0.76 using 24×24 lens, and to 0.83 using 32×32 lens at $-60°$. Second, comparing the simulation (solid curves) with the testbed results (dotted curves), we observe high consistency, which validates the fidelity of our simulation.

Figure 18(b) further plots the normalized power (i.e., the received power is normalized by dividing by the power of $m = 16$ phased array at $0°$ and $1.5m$ away) of (iv) as we vary the number of speakers. As expected, increasing the number of speakers from 2 to 4, 6, 8, and 16 increases the normalized power to 1.3x, 1.6x, 1.8x and 2.1x, respectively.

5.3 Distance Estimation Performance

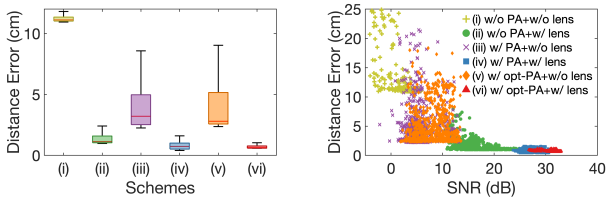
Next we evaluate the impact of our approach on distance estimation accuracy. As described in Section 5.1, we evaluate the benefit of acoustic lens and phased array on the well-known distance estimation techniques: FMCW.

Figure 19(a) plots the errorbar of the distance estimation error, where the center, lower bound, and upper bound of the errorbar correspond to median, 25%, 75% of the distance error, respectively. (vi) performs the best and out-performs



(a) Lens size. (b) Phased array size.

Figure 18: The impact of lens size and phased array size on the received power of (iv). Dotted curves represent results from the testbed experiments while solid curves represent results from COMSOL simulation.



(a) Distance estimation error. (b) SNR v.s. distance error.

Figure 19: The distance estimation performance using acoustic lens.

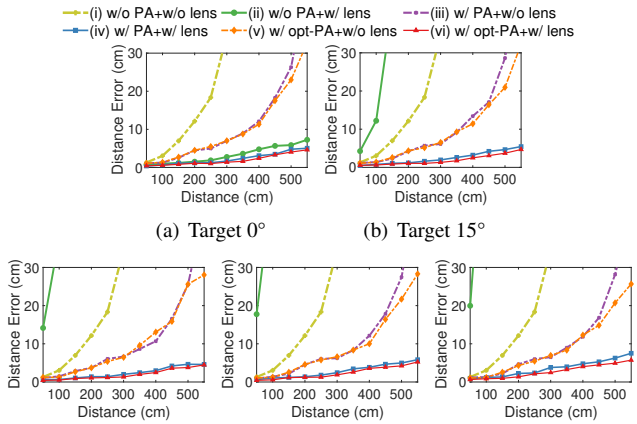
(i), (ii), (iii), (iv), and (v) by 93%, 25%, 76%, 12%, and 72%, respectively. To better understand where the improvement comes from, we plot the distance estimation error of each FMCW chirp and its corresponding SNR in Figure 19(b). We can see that, as expected, the FMCW chirp with a higher SNR leads to a lower error. The optimized phased array with acoustic lens has the highest SNR, so it yields the lowest distance estimation error.

5.3.1 Impact of Measurement Distance

We evaluate the impact of distance on the distance estimation error as shown in Figure 20. As we can see in Figure 20(a), (vi) performs the best and out-performs (i), (ii), (iii), (iv), and (v) by 94%, 45%, 79%, 18%, and 81%, respectively. (vi) increases the operation distance from $0.5m$ in (i) to $3.5m$. Note that $3.5m$ operation distance is very good considering the total power of our 6 speakers is only $0.1W$. In comparison, [38] achieves $4.5m$ operation distance using a $2.5W$ speaker.

5.3.2 Impact of Measurement Angle

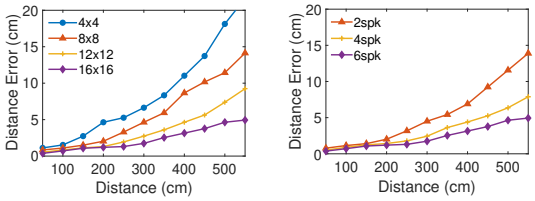
Next we further show the impact of measurement angle on the distance estimation performance. From Figure 20(b)-(e), we can see that (vi) still performs the best in all directions and out-performs (i), (ii), (iii), (iv), and (v) by 93%, 99%, 77%, 21%, and 76%, respectively. However, when we increase the measurement angle from $0°$ to $15°$, $30°$, $45°$, and $60°$, the distance error of (vi) also increases from $1.28cm$ to $1.30cm$, $1.32cm$, $1.86cm$, and $2.47cm$, respectively.



(a) Target 0° (b) Target 15°

(c) Target 30° (d) Target 45° (e) Target 60°

Figure 20: The impact of distance on the distance estimation error while steering in various directions.



(a) Lens size. (b) Phased array size.

Figure 21: The impact of lens size and phased array size on the distance estimation error of (iv).

5.3.3 Impact of Lens Size and Phased Array Size

We evaluate the impact of lens size on distance estimation performance. We experiment with 4×4, 8×8, 12×12 and 16×16 lenses in our testbed. As shown in Figure 21(a), increasing the lens size improves the distance estimation errors. A 16×16 lens reduces the error over a 4×4, 8×8, and 12×12 lens by 73%, 62%, and 36%, respectively.

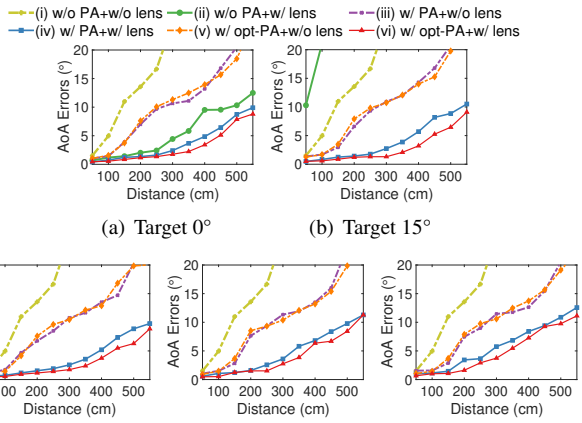
We also evaluate the impact of phased array size. As shown in Figure 21(b), increasing the array size improves the distance estimation due to the enhanced SNR. For example, a 6-speaker phased array with AMS reduces the distance estimation error over 1-, 2-, and 4-speaker phased array by 95%, 91%, and 80%, respectively.

5.4 AoA Estimation Performance

In this section, we compare the AoA estimation using 1D MUSIC as introduced in Section 5.1.

5.4.1 Impact of Measurement Distance

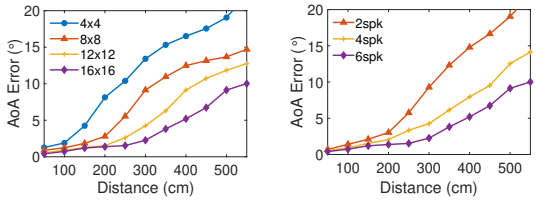
Figure 22(a) plots the AoA error versus the measurement distance. In all schemes, the distance estimation errors increase with an increasing distance. (vi) performs the best and reduces the AoA error of (i), (ii), (iii), (iv), and (v) by 92%, 44%, 76%, 16%, and 77%, respectively. The result shows that



(a) Target 0° (b) Target 15°

(c) Target 30° (d) Target 45° (e) Target 60°

Figure 22: The impact of distance on AoA estimation error while steering in various directions.



(a) Lens size. (b) Phased array size.

Figure 23: The impact of lens size and phased array size on the AoA estimation error of (iv).

our approach effectively improves AoA sensing accuracy by increasing the SNR.

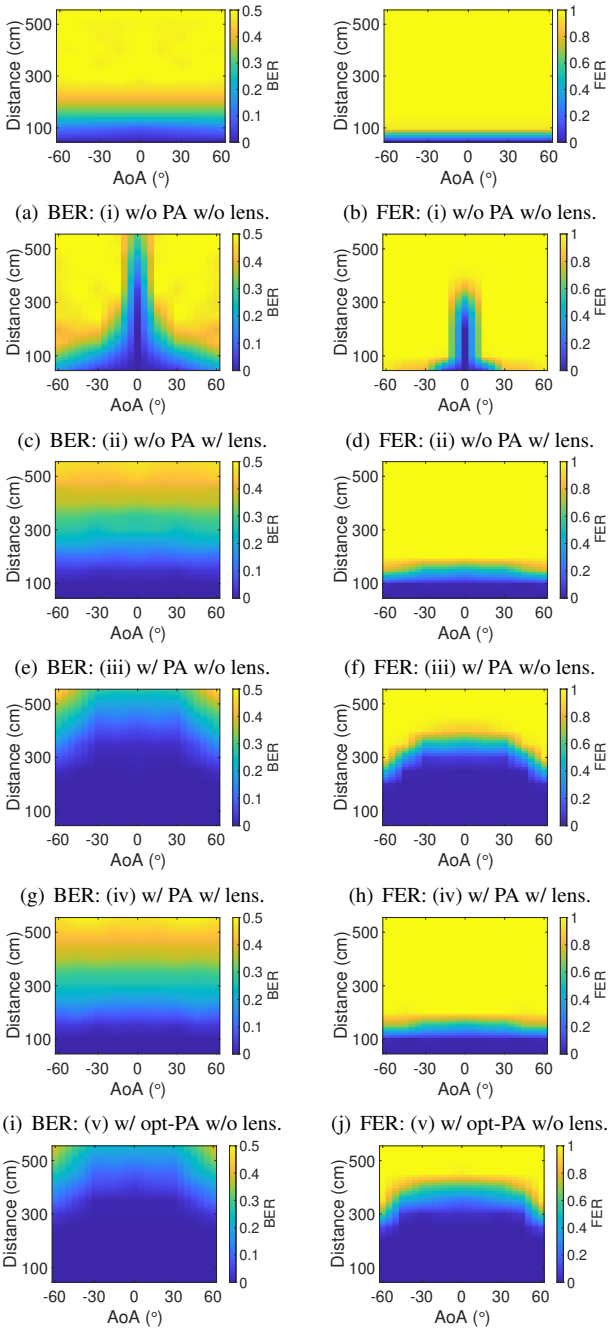
5.4.2 Impact of Measurement Angle

We further show the impact of measurement angle on AoA estimation. Figure 22(b)-(e) plots AoA estimation error for other directions. When the distance is small, the AoA estimation error remains low across all measurement angles of interest; when the distance is large, the AoA error increases more rapidly with the increasing angle. This is expected because when SNR is sufficiently high, the measurement angle has less impact; but when SNR is low, the measurement angle matters. (vi) out-performs (i), (ii), (iii), (iv), and (v) by 90%, 96%, 81%, 26%, and 82%, respectively.

5.4.3 Impact of Lens Size and Phased Array Size

As shown in Figure 23(a), increasing the lens size effectively reduces the AoA estimation error. For example, increasing the lens size from 4×4 to 8×8 improves AoA estimation by 32%, increasing from 8×8 to 12×12 improves by 53%, and increasing from 12×12 to 16×16 further improves by 46%.

Figure 23(b) further plots the AoA estimation as we vary the array size. Increasing the array size improves SNR and reduces the AoA estimation error. Using a 6-speaker phased array reduces the AoA error by 96%, 74%, and 52% over



(a) BER: (i) w/o PA w/o lens. (b) FER: (i) w/o PA w/o lens.
(c) BER: (ii) w/o PA w/ lens. (d) FER: (ii) w/o PA w/ lens.
(e) BER: (iii) w/ PA w/o lens. (f) FER: (iii) w/ PA w/o lens.
(g) BER: (iv) w/ PA w/ lens. (h) FER: (iv) w/ PA w/ lens.
(i) BER: (v) w/ opt-PA w/o lens. (j) FER: (v) w/ opt-PA w/o lens.
(k) BER: (vi) w/ opt-PA w/ lens. (l) FER: (vi) w/ opt-PA w/ lens.

Figure 24: The impact of distance and AoA on the communication performance.

using 1-, 2-, and 4-speaker phased array, respectively.

5.5 Acoustic Communication Performance

Finally, we evaluate the impact of our approach on acoustic communication performance.

5.5.1 Impact of Distance

We put the speakers at a fixed location and gradually increase the distance between the microphone and the speakers. We tested the above six scenarios for communication. Figure 24 shows the bit error rate (BER) and frame error rate (FER) across different AoAs at different distances. We can see that both BER and FER increase with the distance. Due to the use of fixed modulation and FEC, the FER rapidly increases after the distance goes beyond a certain point. If we define the communication range as the range corresponding to 50% FER, we observe that (vi) has 3.9m communication range as shown in Figure 24(l). In comparison, (i), (ii), (iii), (iv), and (v) have communication ranges of 0.8m, 3.0m, 1.5m, 3.5m, and 1.5m, respectively.

5.5.2 Impact of AoA

Figure 24 also shows the impact of AoA on BER and FER in our testbed. (vi) achieves low error rates within 60° and 2.8m. This is good coverage considering the total power of our 6 speakers is only 20mW. In comparison, for the same 60° coverage, the other schemes' range is much smaller.

6 Discussion

Acoustic metasurfaces can effectively boost the signal quality, and improve sensing and communication performance. Compared with a large phased array, our approach of using a small phased array and metasurface is more compact, cost effective, and energy efficient. To fully realize the potential of AMS, several challenges remain to be addressed in the future: (i) further reducing the AMS size so that it can be applied to more applications (e.g., mobile devices), (ii) supporting a wider band, and (iii) further simplifying fabrication process. Figure 17 shows that our AMS can support 16-20KHz within 20% amplitude loss. This is sufficient for our purpose but may need further improvement if a wider band is required.

7 Conclusion

In this paper, we develop a novel acoustic system that uses AMS and multiple speakers together to achieve dynamic steering and high SNR. The increase in SNR can be translated into higher accuracy in distance and AoA estimation and larger communication range in acoustic communication. Encouraged by the promising results, we plan to explore more applications that can benefit from our design.

Acknowledgments

We are grateful for Yasaman Ghasempour's insightful feedback and anonymous reviewers' helpful comments.

References

[1] Bela: create beautiful interaction with sensors and sound. https://bela.io/, 2022.

[2] COMSOL: simulate real-world designs, devices, and processes with multiphysics software from comsol. https://www.ti.com/product/LM386, 2023.2.

[3] LM386: 700-mw, mono, 5- to 18-v, analog input class-ab audio amplifier. https://www.ti.com/product/LM386, 2023.2.

[4] THS4001: 270-mhz voltage-feedback amplifier. https://www.ti.com/product/THS4001, 2023.2.

[5] I. F. Akyildiz, D. Pompili, and T. Melodia. State-of-the-art in protocol research for underwater acoustic sensor networks. In *Proc. of WUWNet*, 2006.

[6] Badreddine Assouar, Bin Liang, Ying Wu, Yong Li, Jian-Chun Cheng, and Yun Jing. Acoustic metasurfaces. *Nature Reviews Materials*, 3(12):460–472, 2018.

[7] Yang Bai, Nakul Garg, and Nirupam Roy. Spidr: Ultra-low-power acoustic spatial sensing for micro-robot navigation. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 99–113, 2022.

[8] Rainer Bauböck. Migration and citizenship. *Journal of Ethnic and Migration Studies*, 18(1):27–48, 1991.

[9] Tuochao Chen, Justin Chan, and Shyam Gollakota. Underwater messaging using mobile devices. In *Proc. of ACM SIGCOMM*, 2022.

[10] Xing Chen, Peng Liu, Zewei Hou, and Yongmao Pei. Magnetic-control multifunctional acoustic metasurface for reflected wave manipulation at deep subwavelength scale. *Scientific reports*, 7(1):1–9, 2017.

[11] Krishna Chintalapudi, Venkat Padmanabhan, and Ramarathnam Venkatesan. Dhwani: Secure peer-to-peer acoustic nfc. In *Proc. of ACM SIGCOMM*, 2013.

[12] Mandar Chitre, Shiraz Shahabudeen, and Milica Stojanovic. Underwater acoustic communications and networking: Recent advances and future challenges. *Marine Technology Society Journal*.

[13] Nicholas Fang, Dongjuan Xi, Jianyi Xu, Muralidhar Ambati, Werayut Srituravanich, Cheng Sun, and Xiang Zhang. Ultrasonic metamaterials with negative modulus. *Nature materials*, 5(6):452–456, 2006.

[14] Romain Fleury and Andrea Alù. Extraordinary sound transmission through density-near-zero ultranarrow channels. *Physical review letters*, 111(5):055501, 2013.

[15] Zhihui Gao, Ang Li, Dong Li, Jialin Liu, Jie Xiong, Yu Wang, Bing Li, and Yiran Chen. Mom: Microphone based 3d orientation measurement. In *Proc. of ACM/IEEE IPSN*, 2022.

[16] Nakul Garg, Yang Bai, and Nirupam Roy. Owlet: enabling spatial information in ubiquitous acoustic devices. In *Proc. of ACM MobiSys*, 2021.

[17] Unsoo Ha, Junshan Leng, Alaa Khaddaj, and Fadel Adib. Food and liquid sensing in practical environments using rfids. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1083–1100, 2020.

[18] Fu-Li Hsiao, Ting-Kuo Li, Pin-Chieh Chen, Su-Chao Wang, Ke-Wei Lin, Wei-Ling Lin, Ying-Pin Tsai, Wen-Kai Lin, and Bor-Shyh Lin. Phase resonance and sensing application of an acoustic metamaterial based on a composite both-sides-open disk resonator arrays. *Sensors and Actuators A: Physical*, 339:113524, 2022.

[19] Yun Jing, Jun Xu, and Nicholas X Fang. Numerical study of a near-zero-index acoustic metamaterial. *Physics Letters A*, 376(45):2834–2837, 2012.

[20] Hao Kong, Xiangyu Xu, Jiadi Yu, Qilin Chen, Chenguang Ma, Yingying Chen, Yi-Chao Chen, and Linghe Kong. m$\hat{3}$track: mmwave-based multi-user 3d posture tracking. In *Proc. of ACM MobiSys*, 2022.

[21] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using WiFi. In *ACM SIGCOMM Computer Communication Review*, volume 45(4), pages 269–282. ACM, 2015.

[22] Manikanta Kotaru and Sachin Katti. Position tracking for virtual reality using commodity wifi. In *Proceedings of the 10th on Wireless of the Students, by the Students, and for the Students Workshop*, S3 '18, pages 15–17, New York, NY, USA, 2018. ACM.

[23] Hyewon Lee, Tae Hyun Kim, Jun Won Choi, and Sunghyun Choi. Chirp signal-based aerial acoustic communication for smart devices. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2407–2415. IEEE, 2015.

[24] Kyung Hoon Lee, Kunhao Yu, An Xin, Zhangzhengrong Feng, Qiming Wang, et al. Sharkskin-inspired magnetoactive reconfigurable acoustic metamaterials. *Research*, 2020, 2020.

[25] Dong Li, Jialin Liu, Sunghoon Ivan Lee, and Jie Xiong. Lasense: Pushing the limits of fine-grained activity sensing using acoustic signals. In *Proc. of IMWUT/UbiComp*, 2022.

[26] Jensen Li and Che Ting Chan. Double-negative acoustic metamaterial. *Physical Review E*, 70(5):055602, 2004.

[27] Yong Li, Xue Jiang, Rui-qi Li, Bin Liang, Xin-ye Zou, Lei-lei Yin, and Jian-chun Cheng. Experimental realization of full control of reflected waves with subwavelength acoustic metasurfaces. *Physical Review Applied*, 2(6):064002, 2014.

[28] Yong Li, Bin Liang, Zhong-ming Gu, Xin-ye Zou, and Jian-chun Cheng. Reflected wavefront manipulation based on ultrathin planar acoustic metasurfaces. *Scientific reports*, 3(1):1–6, 2013.

[29] Zixian Liang and Jensen Li. Extreme acoustic metamaterial by coiling up space. *Physical review letters*, 108(11):114301, 2012.

[30] Qiongzheng Lin, Zhenlin An, and Lei Yang. Rebooting ultrasonic positioning systems for ultrasound-incapable smart devices. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[31] Yuan Liu, Bo Jiu, and Hongwei Liu. Admm-based transmit beampattern synthesis for antenna arrays under a constant modulus constraint. *Signal Processing*, 171:107529, 2020.

[32] Guancong Ma and Ping Sheng. Acoustic metamaterials: From local resonances to broad horizons. *Science advances*, 2(2):e1501595, 2016.

[33] YONGSEN MA, GANG ZHOU, and SHUANGQUAN WANG. Wifi sensing with channel state information: A survey. *ACM Comput. Survey*, June 2019.

[34] Yunfei Ma, Nicholas Selby, and Fadel Adib. Minding the billions: Ultra-wideband localization for deployed rfids. In *Proc. of ACM MobiCom*, 2017.

[35] Wenguang Mao, Jian He, and Lili Qiu. CAT: high-precision acoustic motion tracking. In *Proc. of ACM MobiCom*, 2016.

[36] Wenguang Mao, Wei Sun, Mei Wang, and Lili Qiu. Deeprange: Ranging via deep learning. In *Proc. of UbiComp*, 2021.

[37] Wenguang Mao, Mei Wang, and Lili Qiu. Aim: Acoustic imaging on a mobile. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 468–481. ACM, 2018.

[38] Wenguang Mao, Mei Wang, Wei Sun, Lili Qiu, Swadhin Pradhan, and Yi-Chao Chen. Rnn-based room scale hand motion tracking. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[39] Wenguang Mao, Zaiwei Zhang, Lili Qiu, Jian He, Yuchen Cui, and Sangki Yun. Indoor follow me drone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 345–358. ACM, 2017.

[40] Gianluca Memoli, Mihai Caleap, Michihiro Asakawa, Deepak R. Sahoo, Bruce W. Drinkwater, and Sriram Subramanian. Metamaterial bricks and quantization of meta-surfaces. *Nature Communication*, 2017.

[41] Gianluca Memoli, Letizia Chisari, Jonathan P. Eccles, Mihai Caleap, Bruce W. Drinkwater, and Sriram Subramanian. Vari-sound: A varifocal lens for sound. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–14, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Rajalakshmi Nandakumar, Shyam Gollakota, and Nathaniel Watson. Contactless sleep apnea detection on smartphones. In *Proc. of ACM MobiSys*, 2015.

[43] Rajalakshmi Nandakumar, Vikram Iyer, Desney Tan, and Shyamnath Gollakota. FingerIO: Using active sonar for fine-grained finger tracking. In *Proc. of ACM CHI*, pages 1515–1525, 2016.

[44] Jong Jin Park, KJB Lee, Oliver B Wright, Myoung Ki Jung, and Sam H Lee. Giant acoustic concentration by extraordinary transmission in zero-mass metamaterials. *Physical review letters*, 110(24):244302, 2013.

[45] Chunyi Peng, Guobin Shen, Yongguang Zhang, Yanlin Li, and Kun Tan. BeepBeep: a high accuracy acoustic ranging system using COTS mobile devices. In *Proc. of ACM SenSys*, 2007.

[46] Swadhin Pradhan, Shuozhe Li, and Lili Qiu. Rotation sensing using passive rfid tags. In *Proc. of MobiHoc*, 2021.

[47] Qifan Pu, Sidhant Gupta, Shyam Gollakota, and Shwetak Patel. Whole-home gesture recognition using wireless signals. In *Proc. of ACM MobiCom*, 2013.

[48] Saeed Ur Rahman, Qunsheng CAO, Muhammad Mansoor Ahmed, and Hisham Khalil. Analysis of linear antenna array for minimum side lobe level, half power beamwidth, and nulls control using pso. *Journal of Microwaves, Optoelectronics and Electromagnetic Applications*, 16:577–591, 2017.

[49] Sheng Shen, Daguan Chen, Yu-Lin Wei, Zhijian Yang, and Romit Roy Choudhury. Voice localization using nearby wall reflections. In *Proc. of ACM MobiCom*, 2020.

[50] Deepak Vasisht, Swarun Kumar, and Dina Katabi. Decimeter-level localization with a single wifi access point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 165–178, 2016.

[51] Anran Wang and Shyamnath Gollakota. Millisonic: Pushing the limits of acoustic motion tracking. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2019.

[52] Anran Wang, Jacob E Sunshine, and Shyamnath Gollakota. Contactless infant monitoring using white noise. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[53] Ju Wang, Jie Xiong, Xiaojiang Chen, Hongbo Jiang, Rajesh Krishna Balan, and Dingyi Fang. Tagscan: Simultaneous target imaging and material identification with commodity rfid devices. In *Proc. of ACM MobiCom*, pages 288–300. ACM, 2017.

[54] Jue Wang, Fadel Adib, Ross Knepper, Dina Katabi, and Daniela Rus. RF-compass: robot object manipulation using RFIDs. In *Proc. of the 19th annual international conference on Mobile computing and networking*, pages 3–14, 2013.

[55] Jue Wang and Dina Katabi. Dude, where's my card? RFID positioning that works with multipath and non-line of sight. In *Proc. of the ACM SIGCOMM*, pages 51–62, 2013.

[56] Mei Wang, Wei Sun, and Lili Qiu. Mavl: Multiresolution analysis of voice localization. In *Proc. of NSDI*, 2021.

[57] Wei Wang, Alex X Liu, and Ke Sun. Device-free gesture tracking using acoustic signals. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 82–94. ACM, 2016.

[58] Teng Wei and Xinyu Zhang. mTrack: high precision passive tracking using millimeter wave radios. In *Proc. of ACM MobiCom*, 2015.

[59] Teng Wei and Xinyu Zhang. Gyro in the air: Tracking 3d orientation of batteryless internet-of-things. In *Proceedings of MobiCom*, MobiCom '16, pages 55–68, New York, NY, USA, 2016. ACM.

[60] Yangbo Xie, Wenqi Wang, Huanyang Chen, Adam Konneker, Bogdan-Ioan Popa, and Steven A Cummer. Wavefront modulation and subwavelength diffractive acoustics with an acoustic metasurface. *Nature communications*, 5(1):5553, 2014.

[61] Hongfei Xue, Yan Ju, Chenglin Miao, Yijiang Wang, Shiyang Wang, Aidong Zhang, and Lu Su. mmmesh: towards 3d real-time dynamic human mesh construction using millimeter-wave. In *Proc. of MobiSys*, 2021.

[62] Min Yang, Guancong Ma, Ying Wu, Zhiyu Yang, and Ping Sheng. Homogenization scheme for acoustic metamaterials. *Physical Review B*, 89(6):064309, 2014.

[63] Min Yang, Guancong Ma, Zhiyu Yang, and Ping Sheng. Coupled membranes with doubly negative mass density and bulk modulus. *Physical review letters*, 110(13):134301, 2013.

[64] Sangki Yun, Yi chao Chen, and Lili Qiu. Turning a mobile device into a mouse in the air. In *Proc. of ACM MobiSys*, May 2015.

[65] Sangki Yun, Yi-Chao Chen, Huihuang Zheng, Lili Qiu, and Wenguang Mao. Strata: Fine-grained acoustic-based device-free tracking. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 15–28. ACM, 2017.

[66] Bingsheng Zhang, Qin Zhan, Si Chen, Muyuan Li, Kui Ren, Cong Wang, and Di Ma. Enabling keyless secure acoustic communication for smartphones. *IEEE internet of things journal*, 1(1):33–45, 2014.

[67] Zengbin Zhang, David Chu, Xiaomeng Chen, and Thomas Moscibroda. Swordfight: Enabling a new class of phone-to-phone action games on commodity phones. In *Proc. of ACM MobiSys*, 2012.

Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays

Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica
University of California, Berkeley

Abstract

Cloud applications are increasingly distributing data across multiple regions and cloud providers. Unfortunately, wide-area bulk data transfers are often slow, bottlenecking applications. We demonstrate that it is possible to significantly improve inter-region cloud bulk transfer throughput by adapting network overlays to the cloud setting—that is, by routing data through indirect paths at the application layer. However, directly applying network overlays in this setting can result in unacceptable increases in cloud egress prices. We present Skyplane, a system for bulk data transfer between cloud object stores that uses cloud-aware network overlays to optimally navigate the trade-off between price and performance. Skyplane's planner uses mixed-integer linear programming to determine the optimal overlay path and resource allocation for data transfer, subject to user-provided constraints on price or performance. Skyplane outperforms public cloud transfer services by up to 4.6× for transfers within one cloud and by up to 5.0× across clouds.

1 Introduction

Increasingly, cloud applications transfer data across datacenter boundaries, both across multiple regions within a cloud provider (multi-region) and across multiple cloud providers (multi-cloud). This is in part due to privacy regulations, the availability of specialized hardware, and the desire to prevent vendor lock-in. In a recent survey [26], more than 86% of 727 respondents had adopted a multi-cloud strategy across diverse workloads. Thus, support for fast, cross-cloud bulk transfers is increasingly important.

Applications transfer data between datacenters for batch processing (e.g. ETL [9], Geo-Distributed Analytics [54]), and production serving (e.g. search indices [34]). Extensive prior work optimizes the throughput of bulk data transfers between datacenters within application-defined minimum performance constraints [34, 36, 38, 64]. All major clouds offer services for bulk transfers such as AWS DataSync [5], Azure AzCopy [22], and GCP Storage Transfer Service [31].

From the perspective of a cloud customer, transfer throughput and cost (price) are the two important metrics of transfers in the cloud. Thus we ask *how can we optimize network cost and throughput for cloud bulk transfers?* We study this question in the context of designing and implementing Skyplane, an open-source cloud object transfer system.

A seemingly natural approach is to optimize the routing protocols in cloud providers internal networks to support higher-throughput data transfers. Unfortunately, this is not feasible for two reasons. First, rearchitecting the IP layer routing protocol to optimize for high-throughput bulk transfer could be negatively impact other applications that are sensitive to network latency. Second, cloud providers lack a strong incentive to optimize data transfer to other clouds. Indeed, AWS DataSync [5], AzCopy [22], GCP Storage Transfer [31], AWS Snowball [62], and Azure Data Box Disk [12], all support data transfer *into*, but not out of, their respective clouds. Improvements to cross-cloud peering must be achieved with the cooperation of both the source and destination providers.

Skyplane's key observation is that we can instead identify *overlay paths*—paths that send data via intermediate regions—that are faster than the direct path. The throughput of the direct path from Azure's `Central Canada` region to GCP's `asia-northeast1` region is 6.2 Gbps. Instead, Skyplane can route the transfer via an intermediate stop at Azure's `US West 2` with a throughput of 12.4 Gbps for a 2.0× speedup (Fig. 1). Crucially, this can be implemented on top of the cloud providers' services without their explicit buy-in.

We are not the first to propose the use of overlay networks on the public Internet [8]. However, these techniques ignore two key considerations of public clouds: **price** and **elasticity**.

First, the highest-bandwidth overlay path may have an unacceptably high **price**. Cloud providers charge for data egress separately for each hop along the overlay path. To reduce the cost of the overlay, it is essential to transfer data along cheap paths to trade off price and performance. For example, in Fig. 1, one can achieve 13.9 Gbps by instead using Azure's `East Japan` region as the relay, but the cost would be 1.9× that of transferring data directly. In contrast, using Azure's

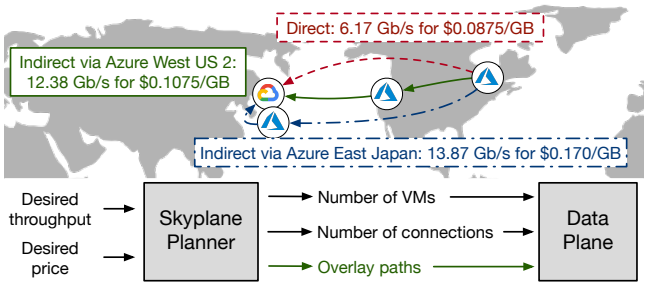


Figure 1: **Cloud-aware overlays:** Skyplane optimally transfers across cloud regions and providers subject to the user's cost and throughput requirements. Skyplane finds the visualized overlay path from Azure's `Central Canada` region to GCP's `asia-northeast1`, which is 2.0× faster but just 1.2× higher in price than the direct path.

`West US 2` region has only a 1.2× cost overhead with similar performance. Thus, Skyplane operates in a richer *problem space* than traditional application-level routing—one where cloud instance and cloud egress fees are significant.

Second, whereas the bandwidth between two nodes in a traditional network overlay [8] is considered "fixed," in Skyplane's setting it depends on **elasticity**—the ability to allocate more resources at each cloud region. For example, one can increase the capacity of any overlay path by simply allocating more VM instances in each cloud region. There are a limited number of physical machines at each cloud region, which cloud providers pass on to users in the form of instance limits. An overlay enables improved throughput beyond this limit. Thus, Skyplane operates in a richer *solution space* than traditional application-level routing—one where we must choose the number of VMs to use as relays due to cloud elasticity.

Skyplane addresses both **price** and **elasticity**, empowering users to navigate the trade-off between price and performance while leveraging the elasticity of cloud resources. Users can ask Skyplane to maximize bandwidth subject to a cost ceiling, or minimize cost subject to a bandwidth floor.

At the heart of Skyplane is a planner that computes a data transfer plan, subject to the user's constraints, that specifies the overlay path to use and amount of cloud resources to allocate along that path. Price and elasticity make it challenging to compute the plan. Our insight is that, with some care, planning can be formulated as *linear* constraints. Thus, Skyplane's planner can discover the optimal plan by solving a mixed-integer linear program (MILP), or closely approximate the optimal plan by solving a relaxed linear program (LP). Both can be accomplished using a fast, off-the-shelf solver.

Our Skyplane prototype[1] outperforms AWS DataSync by up to 4.6× and GCP Storage Transfer by up to 5.0×. Skyplane also outperforms academic baselines such as RON by 34% while reducing cost by 62%.

[1] https://github.com/skyplane-project/skyplane

2 Background

Network overlays In the early 2000s, network overlays emerged as a technique for *application-level routing* without the *participation of underlying network providers*. These network overlays can be designed to improve performance or reliability. Notable network overlays include Chord [60], Resilient Overlay Networks (RON) [8], Bullet [41], Baidu BDS [65] and Akamai's backbone [52, 58].

Although ISPs may have broad visibility into their networks, the metrics that ISPs use to select routes may not align with user preferences. Wide-area networks today do not allow specification of alternative routing preferences while network overlays provide applications a mechanism to control routing decisions. For example, Akamai uses a network overlay to reduce the latency of CDN misses while RON routes around network outages via an unaffected intermediate host.

RON is implemented by periodically measuring network performance via probes embedded in a fixed set of routers. When path outages occur, RON selects an intermediate relay router to circumvent the outage. This intermediate router is selected to have low packet loss or latency to/from the client and server. Optionally, RON can use a model of TCP Reno's throughput [53] to select intermediate routers. RON will generally select only a single intermediate node.

Wide-area networking in the cloud From the perspective of cloud customers, the cloud is *elastic*—additional resources can be allocated on demand. For example, an overloaded cloud application can leverage the cloud's elasticity by allocating additional VM instances. However, the physical reality of the cloud is that there are only finite resources at each region. Therefore, cloud providers impose *service limits* on their customers for resources such as VMs.

Each VM's network bandwidth is throttled according to its instance type. For example, an AWS `m5.8xlarge` instance can use at most 10 Gbps of network bandwidth, and an Azure `Standard_D32_v5` instance can use at most 16 Gbps of network bandwidth. Furthermore, only some of the available bandwidth can be used for egress traffic to another cloud provider. The policies differ by cloud provider. AWS limits VM egress bandwidth to the larger of 5 Gbps or 50% of total bandwidth [4], GCP limits VM egress bandwidth to any public IP address to 7 Gbps [30], and Microsoft Azure has no egress limit beyond the total bandwidth limit for a VM. Of course, the actual achievable TCP network bandwidth is subject to congestion control which may be less than the limit.

Cloud egress pricing Cloud providers charge egress prices for network traffic leaving a cloud region. Importantly, egress prices are assessed based on the *volume* of data transferred, not the rate at which it is transferred. Transferring a file at 10 Mbps or at 10 Gbps will result in the same egress charge. Egress charges introduce asymmetry in billing—there is no corresponding ingress charge for transfers into a cloud.

For intra-cloud transfers (i.e., transfers between two regions or zones in the same cloud), transfers between geographically distant endpoints are priced more than transfers between nearby endpoints. In contrast, inter-cloud transfers (i.e., transfers between two cloud providers) are billed at the same rate regardless of the transfer's geographic distance. For example, the egress price from a single Azure region is billed at the same rate for *any* destination outside of Azure, including any region in AWS or GCP [6, 29, 51].

Egress prices typically dominate the cost of a bulk transfers. For example, if a VM sends data at a rate of 1 Gbps for an hour on AWS with an Internet egress price of $0.09/GB, the total egress charge will total $40.50, which far exceeds the VM price of $1.50 (for `m5.8xlarge`) [6].

Cloud object storage AWS, Azure, and GCP provide object storage APIs that allow customers to save data attached to a string key. Data is stored immutably and therefore any updates require writing a new version. Unlike POSIX file systems, object stores do not provide atomic metadata operations (e.g., rename). Consistency models vary across providers. Cloud object stores store copies of a blob on multiple machines to improve availability and durability. Large objects support concurrent writes via sharding. Read throughput of a single shard may be limited by the provider (e.g. 60 MB/s for Azure [13]).

3 Overview of Skyplane

Skyplane allows applications to efficiently transfer large objects from an object store in one region to an object store in another cloud region or provider. To use Skyplane, the user installs the Skyplane client locally and configures it with access to cloud provider-supplied credentials. Then, the user submits a job, together with a constraint on price or bandwidth. The job specifies which objects to transfer, the source cloud provider and region, and the destination cloud provider and region. The constraint can have one of two forms: it can ask Skyplane to optimize either *bandwidth subject to a price ceiling*, or *price subject to a bandwidth floor*.

Skyplane itself comprises a *planner* (Fig. 1, bottom) and a *data plane* (Fig. 2). Given the user's job and constraint, the planner produces an optimal data transfer plan to complete the job subject to the constraint. The planner relies on a profile of the network throughput between different cloud regions. The data plane is responsible for executing the data transfer plan: allocating cloud resources (e.g., VMs), transferring data between them, and interacting with object stores.

3.1 Overlay formulation in Skyplane's planner

Suppose the user needs to transfer an object from a source cloud region, A, to a destination cloud region, B. A naïve object transfer system might spawn VMs in regions A and B, and transfer data via a TCP connection between the two VMs.

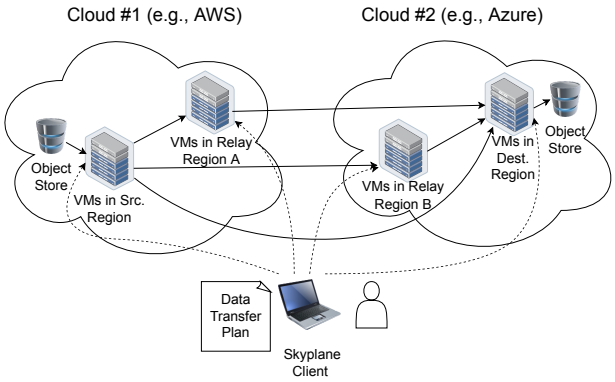


Figure 2: Skyplane splits an example data transfer over three paths: the direct path, and two indirect paths. Dashed lines indicate control orchestration (e.g., for spawning VMs) and solid lines depict the flow of object data.

Skyplane improves performance compared to this baseline by applying principles from overlay networks [8]. For example, Skyplane may identify a third cloud region, C, and transfer data from A to B via C. This is accomplished at the application layer; Skyplane will spawn a VM in region C, set up TCP connections from A to C and from C to B. We refer to intermediate regions like C as *relay regions*.

The baseline approach ($A \rightarrow B$) routes data along the "direct path," since it uses the default path provided by the Internet. However, Skyplane ($A \rightarrow C \rightarrow B$) routes data along the an "indirect path," that may not be on the Internet-provided default path. An indirect path may use multiple relays although a single relay is usually sufficient.

A key difference between Skyplane and classical overlay networks is that Skyplane takes price into account when choosing the overlay path to use for a job. Concretely, Skyplane's planner uses a *price grid* and a *throughput grid* to determine which indirect path to use. The price grid specifies the price of transferring data between each pair of cloud regions, in each direction. We computed the price grid based on information on the cloud providers' websites and from querying the cloud APIs. The throughput grid is collected by measuring the network, as we explain in the next subsection.

Note that throughput grid measurements are made using TCP connections, subject to TCP congestion control. Thus, the throughput grid measures the bandwidth available to *a single user* for transferring data, accounting for cross-traffic from other users' flows. We assume a high degree of statistical multiplexing in wide-area network traffic—in other words, that the bandwidth consumed by a single user's bulk transfer is negligible compared to the total available inter-region bandwidth. This allows a Skyplane user to compute a data transfer plan without regard to other users' bulk transfers using Skyplane or other bulk transfer tools—all cross traffic from other users is assumed to be accounted for in the throughput grid.

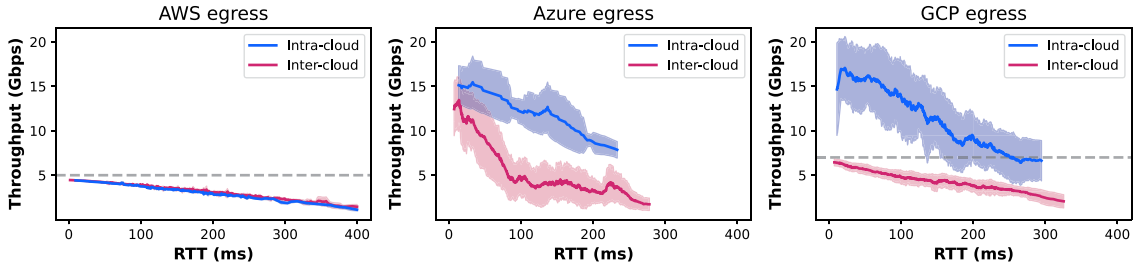


Figure 3: **Intra-cloud vs. inter-cloud links**: Inter-cloud links are consistently slower than intra-cloud links for network routes from Azure and GCP. Service limits are shown with a dashed line; GCP throttles inter-cloud egress to 7 Gbps while AWS throttles *all egress traffic* to 5 Gbps.

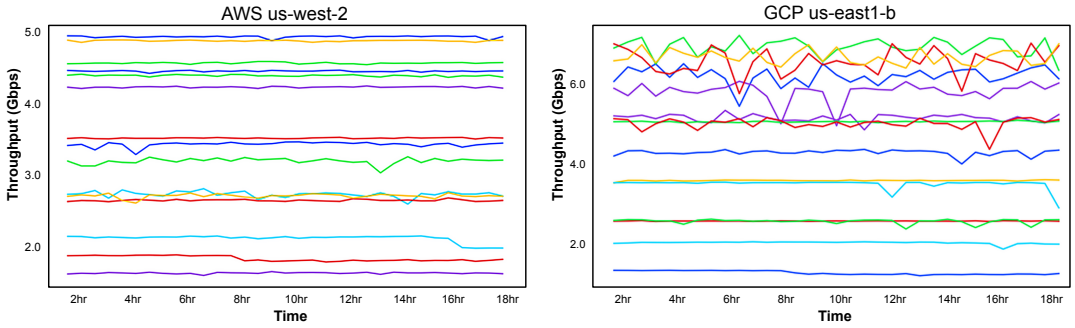


Figure 4: **Stability of egress flows over 18 hour period**: Continuous probes of cloud networks over one day reveal that routes from AWS have stable throughput over time. Paths between GCP regions are noisy but have a consistent mean.

As we show in the next subsection, the bandwidth of inter-region TCP connections is relatively stable in the short term, validating our assumption of high statistical multiplexing.

3.2 Profiling cloud networks

The planner relies on a profile of the network throughput between pairs of cloud regions. We collected a throughput grid by measuring the TCP goodput between each region pair using `iperf3`. In total, computing this profile cost approximately $4000 in egress charges.

Fig. 3 displays the relationship between network latency and throughput for profiling routes originating from GCP and Azure for our measured throughput grid. For GCP, we leverage internal IPs which improve intra-cloud bandwidth. For both GCP and Azure, intra-cloud routes had lower tail RTTs than inter-cloud routes. We observe that in both GCP and Azure, inter-cloud links are slower than intra-cloud links. As Azure has no service limit for egress bandwidth, we see the fastest intra-cloud links achieve up to the NIC capacity of 16 Gbps. However, both GCP and AWS encounter egress throttling at 7 Gbps and 5 Gbps respectively.

A natural question is how frequently the throughput grid must be re-measured. Fig. 4 visualizes achieved throughput from AWS `us-west-2` and GCP `us-east1-b` taken every 30 minutes over an 18 hour timespan. Throughput is very stable over time for both inter-cloud and intra-cloud routes from AWS `us-west-2`. Routes from GCP `us-east1-b` to AWS destinations is similarly very stable but intra-cloud routes to GCP destinations are less stable. Regardless, the overall rank order of regions by throughput remains mostly consistent over medium-term timescales. Thus, it should be sufficient to profile networks relatively infrequently (i.e. every few days). In practice, this information could be collected by third-party service, or measured via active probing along live transfers.

3.3 Skyplane's data plane

Skyplane's data plane executes data transfers using the plan computed by Skyplane's planner. Ephemeral VMs for a single transfer, called "gateways," are provisioned in the source region, destination region, and overlay regions for a transfer plan. Each source gateway reads a small shard of data from the object store and transfers data via intermediate gateways to the destination where the shard is written.

Skyplane reads data from an object store in the source cloud region and writes data to an object store in the destination cloud region. We focus on the object stores provided as a service by AWS S3, Azure Blob Storage, and Google Storage. Unlike a traditional overlay network, there is no central Skyplane service that allocates resources to each user from a pool of "Skyplane resources." Instead, Skyplane can be understood

as a local service run by each user that is invoked when an application needs to transfer data. Skyplane directly allocates cloud resources on the user's behalf when processing a job, and manages those resources to transfer the user's data across cloud regions. This allows Skyplane to allocate and manage each user's resources according to their cost and performance objectives, independently from the cloud providers' existing data transfer services, while relying on clouds to offer a large pool of resources and manage isolation between users.

4 Principles of Skyplane's planner

Skyplane's planner[2] is responsible for developing a plan for transferring data across the wide area to complete an object transfer job submitted by a user or their application (Fig. 5). This plan describes the overlay path and the amount of cloud resources to allocate along that path to facilitate the transfer.

Skyplane's planner supports two modes:

Cost minimizing: The planner will minimize cost subject to an application-specified throughput constraint.

Throughput maximizing: The planner will maximize throughput subject to an application-specified cost constraint.

As we will describe in §5, Skyplane finds the optimal plan by formulating it as an Mixed-Integer Linear Program (MILP) and using a fast but exponential-time solver. This section describes the degrees of freedom available to the optimizer to navigate the price-performance trade-off for the user's specified constraint. Our goal is to describe what aspects of the plan are at the planner's disposal, justify why it is reasonable to vary those aspects of the plan, and describe certain techniques available to the planner to manage the price-performance trade-off. Note that the planner is not directly programmed to use these techniques; they are merely patterns that it discovers in the course of finding the optimal MILP solution.

4.1 Achieving low instance and egress costs

That bandwidth costs dominate the cost of data transfer (§2) is both a challenge and an opportunity for Skyplane. It is an opportunity because it allows Skyplane to be competitive with the price of using data transfer tools provided directly by the cloud providers (e.g. AWS DataSync, AzCopy, GCP Cloud Transfer Service), as those tools incur bandwidth costs but not instance costs. It is a challenge for Skyplane because it implies that, used naïvely, indirect paths are much more expensive than direct paths. This is because egress bandwidth is charged for each hop along the path. For example, for a path $A \to C \to B$, the bandwidth cost must be paid for both $A \to C$ and $C \to B$, which could be *double* the cost of transferring over the direct path. As a result, it is crucial for Skyplane's optimizer carefully manage egress transfer costs.

[2]Explore Skyplane's planner at `https://optimizer.skyplane.org`

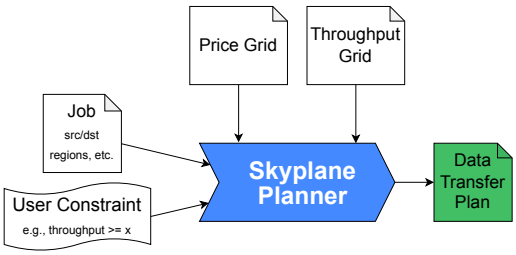


Figure 5: Skyplane's planner considers throughput and cost constraints from the user along with per-cloud price information and an inter-region throughput profile grid to determine the optimal data transfer plan.

4.1.1 Choosing the relay region

One way for Skyplane to manage the additional cost associated with indirect paths is to carefully choose the relay region C to minimize this cost. For example, suppose that a user needs to transfer an object from AWS `us-west-2` (region A) to Azure `UK South` (region B). The direct path $A \to B$ would require the user to pay $0.09 per GB, the cost of bandwidth leaving AWS' network. If the relay region C is chosen in `us-central-1` or `us-east-1`, then the overall bandwidth price will only increase slightly; while the $C \to B$ transfer still incurs $0.09 per GB, as data is leaving AWS' network, the $A \to C$ bandwidth only costs $0.02 per GB, as it is an intra-continental transfer within the cloud provider's network. Skyplane's planner can use the throughput and price grids to identify relay regions that improve the performance of the transfer while minimizing additional bandwidth costs.

4.1.2 Combining multiple paths

Another way to manage the cost of indirect paths is to split the data transfer over multiple paths, in order to make fine-grained trade-offs between price and performance. For example, suppose that Skyplane identifies a high-bandwidth indirect path, but that the path is more expensive than the user's price ceiling. Skyplane can still benefit partially from that indirect path by sending part of the data over that path, at higher cost, and the remaining data over the direct path $A \to B$, at lower cost. Thus, Skyplane may average the price and performance of multiple paths, when doing so allows Skyplane to more optimally satisfy the user's constraints.

4.2 Parallel TCP for high bandwidth

Skyplane uses parallel TCP connections—that is, bundles of TCP connections—to achieve high goodput over a chosen path. This is a well-known technique for achieving good performance, particularly for wide-area transfers [1, 59]. Our Skyplane implementation uses up to 64 outgoing connections for each VM instance, as we empirically measured that using

additional connections typically resulted in diminishing benefits in aggregate goodput. When collecting measurements for the throughput grid, we make sure to use 64 parallel connections to measure the achievable TCP goodput for each ordered pair of regions.

It is known that using multiple TCP streams in parallel may cause an application to obtain more than its "fair share" of bandwidth [25, §A.1], particularly in contexts where networks are running at nearly 100% utilization [36]. Our view is that, despite this, it is acceptable to use multiple TCP connections in parallel in the context of Skyplane. There are three reasons for this. First, it is common for applications to use parallel TCP, including for workloads like bulk data transfer [1, 44]. It is important for Skyplane to appropriately compete with such applications for limited bandwidth. Second, the user pays the cloud provider for bandwidth, both in the form of the bandwidth price (total amount transferred) and the instance price (rate at which data can be transferred), and it is natural for users to be able to make use of the bandwidth that they pay for. Third, cloud providers control the datacenter network, and can shape traffic in the presence of congestion to ensure that each customer gets a fair share of bandwidth.

4.3 Multiple VMs for high bandwidth

For a given overlay path, Skyplane must allocate sufficient resources along the path to achieve high bandwidth. However, the achievable outgoing bandwidth from a VM instance is limited, as described in §2.

Therefore, Skyplane may allocate multiple VM instances at certain regions along the path, to increase *aggregate* data transfer rate of the VMs at each region. Although simply using larger VMs may seem like a viable alternative, it is less effective than using multiple instances due to per-instance bandwidth limits. Skyplane uses a fixed VM size, and its planner chooses how many instances to allocate in each region, under the assumption that TCP goodput scales linearly with the number of allocated VM sizes.

It may seem that Skyplane can achieve an arbitrarily high bandwidth by spawning many instances in each region. Unfortunately, this simple strategy does not work because cloud resources are not perfectly elastic. The finite capacity for VMs in a datacenter is passed down to cloud customers in the form of service limits, which limit the number of VM instances, and therefore the amount of network bandwidth, that users can allocate in each region. While users can request limit increases, these are ultimately subject to resource availability. To model this, Skyplane's planner takes into account a limit on the number of instances that a user can allocate per region.

5 Finding optimal transfer plans

Skyplane's planner searches for cost-efficient high-throughput transfer plans that jointly specify the overlay path, TCP connections between regions and VMs to provision per region.

Variables	
$\mathrm{F} \in \mathbb{R}_+^{\lvert V\rvert\times\lvert V\rvert}$	*Throughput grid*
$\mathrm{N} \in \mathbb{Z}_+^{\lvert V\rvert}$	*VMs per region*
$\mathrm{M} \in \mathbb{Z}_+^{\lvert V\rvert\times\lvert V\rvert}$	*TCP conn. per region*
Constraint: goal throughput	
$\textsc{Tput Goal} \in \mathbb{R}_+^{\lvert V\rvert\times\lvert V\rvert}$	*User's desired throughput*
Constants: provider limit	
$\textsc{Limit}^{link} \in \mathbb{R}_+^{\lvert V\rvert\times\lvert V\rvert}$	*Throughput grid limit*
$\textsc{Limit}^{conn} \in \mathbb{Z}_+^{\lvert V\rvert\times\lvert V\rvert}$	*TCP connection limit*
$\textsc{Limit}^{ingress} \in \mathbb{Z}_+^{\lvert V\rvert}$	*VM limit*
$\textsc{Limit}^{egress} \in \mathbb{Z}_+^{\lvert V\rvert}$	*Egress bandwidth limit*
Constants: provider cost	
$\textsc{Cost}^{egress} \in \mathbb{R}_+^{\lvert V\rvert}$	*Egress cost ($/Gbit)*
$\textsc{Cost}^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*VM cost ($/s)*

Table 1: Symbol table for Skyplane's ILP formulation.

At the core of Skyplane's planner is an optimizer that finds the optimal plan using off-the-shelf Linear Programming (LP) solvers. We formalize the constraints of our problem as Mixed Integer LP (MILP) which can quickly be solved in under 5 seconds with an open-source solver. The problem can be further relaxed into a continuous LP which is solvable in worst-case polynomial time via interior point methods [39].

Independently optimizing for each variable then combining partial solutions would not guarantee a globally optimal solution. It is therefore important that Skyplane's planner models all variables in an integrated search space to obtain provably optimal data transfer plans.

5.1 Cost minimizing overlay paths

Flow networks can naturally represent overlay networking topologies like those used by Akamai [58]. We start with a min-cost flow problem. The following primal LP finds the optimal flow matrix $\mathrm{F} \in \mathbb{R}_+^{\lvert V\rvert\times\lvert V\rvert}$ for a network topology graph $G = (V, E)$ where nodes represent regions and edges are links:

$$\begin{aligned} \underset{\mathrm{F}}{\text{arg min}} \quad & \langle C, \mathrm{F}\rangle \\ \text{subject to} \quad & \textstyle\sum_{(c,v)\in E} \mathrm{F}_{c,v} \geq \textsc{Tput Goal} \\ & \textstyle\sum_{(u,v)\in E} \mathrm{F}_{u,v} = \sum_{v,w} \mathrm{F}_{v,w} \quad \forall v \in V - \{s,t\} \\ & 0 \leq \mathrm{F} \leq \textsc{Limit}^{link} \end{aligned} \tag{1}$$

where s and t are the source and destination regions, $\textsc{Limit}^{link} \in \mathbb{R}_+^{\lvert V\rvert\times\lvert V\rvert}$ is the maximum capacity for each link and $C \in \mathbb{R}_+^{\lvert V\rvert\times\lvert V\rvert}$ is the cost per unit of bandwidth between regions. We use the same notation for matrix and vector inner products: $\langle C, \mathrm{F}\rangle = \sum_{u,v} C_{u,v}\mathrm{F}_{u,v}$.

5.1.1 Objective: Minimize cost from egress and VMs

Min-cost flows do not accurately reflect the cost of transfers in the cloud. The total cost of a transfer in Skyplane includes *egress cost* and *VM cost*. Note that this objective is not linear; we present a linear reformulation in Sec. 5.1.1. We present the full objective is in the in Equation 4a.

Modeling egress cost Unlike physical networks, virtual networks in the cloud will charge the same amount if 1GB of data is sent at 1 Mbps or 10 Gbps. Transfers are priced according to *egress volume* ($ per GB, COST^{egress}) rather than *bandwidth* ($ per Gbps). We can update the cost function to instead model the transfer cost by first computing how much the overlay path costs to run per unit time and then scale that by the runtime for a transfer. We denote the total volume of the transfer as VOLUME. Total egress cost is then:

$$\underbrace{\langle \mathrm{F}, \text{COST}^{egress} \rangle}_{\text{Egress cost per s}} * \underbrace{\text{VOLUME} \div \textstyle\sum_{v \in V} \mathrm{F}_{s,v}}_{\text{Transfer time}} \tag{2}$$

Modeling VM cost Multiple VMs can increase aggregate bandwidth as discussed in Sec. 4.3. To optimally trade-off parallel VMs with the overlay, we introduce a new decision variable $\mathrm{N} \in \mathbb{Z}_+^{|V|}$ that models the number of instances use to transfer data per region. VM count per region may vary due to asymmetric egress and ingress limits. To accurately consider transfer costs from VMs, we add the the following instance cost expression to Equation 2 where COST^{VM} is a vector containing the cost per second per VM in each region:

$$\underbrace{\langle \mathrm{N}, \text{COST}^{VM} \rangle}_{\text{VM cost per s}} * \underbrace{\text{VOLUME} \div \textstyle\sum_{v \in V} \mathrm{F}_{s,v}}_{\text{Transfer time}} \tag{3}$$

Linear reformulation of the objective As written, the objective in Equation 4a is not linear due to a product of variables between F and N. By reformulating the problem to instead consider finding a plan that provides *exactly* TPUT GOAL (instead at least), the runtime for the transfer can be reduced to a constant VOLUME ÷ TPUT GOAL.

5.1.2 Constraints: Cloud provider service limits

Resources are not infinite at cloud regions; providers limit the number of VMs that a user may provision and in some cases, providers may throttle the performance of ingress and egress.

Per VM ingress and egress limits AWS and GCP each throttle egress from their clouds via SDN policies. For AWS, instances with 32 cores or less are limited to 5 Gbps. For GCP, individual flows are limited to 3 Gbps and total egress is service limited to 7 Gbps. Ingress is bottlenecked by VM NIC bandwidth. We constrain the maximum ingress bandwidth per VM to $\text{LIMIT}^{ingress}$ via Constraint 4f and the maximum egress bandwidth per VM to LIMIT^{egress} via Constraint 4g.

Constraining TCP connections Using parallel TCP connections is a well known approach to improve WAN performance as discussed in Section 4.2. Yet, bandwidth does not scale linearly with connections (Figure 9a). We introduce a decision variable $\mathrm{M} \in \mathbb{Z}_+^{|V| \times |V|}$ representing the number of connections between a *pair of regions* (not per VM pair). Constraint 4b ensures M is constrained by N and LIMIT^{conn} (typically 64 per VM). We then limit the total incoming and outgoing connections with Constraints 4i and 4h.

Per-region VM limits We introduce the variable $\mathrm{N} \in \mathbb{Z}_+^{|V|}$ to denote the number of VMs per region. N must be under the global instance cap in Constraint 4j. The optimizer linearly scales the maximum number of egress TCP connections per region by the number of VMs provisioned in each region.

5.1.3 Continuous relaxation of MILP

To improve solve times, N and M are relaxed into real valued variables $\mathrm{N} \in \mathbb{R}_+^{|V|}$ and $\mathrm{M} \in \mathbb{R}_+^{|V| \times |V|}$. Rounding variables down performs comparably to randomized rounding with solutions $\leq 1\%$ from optimal. The relaxed problem has worst case polynomial time complexity [39].

5.1.4 Full formulation of the cost optimal solver

All variables and constants are listed in Table 1. The full formulation of Skyplane's optimizer is:

$$\underset{\substack{\mathrm{F},\,\mathrm{N}\\ \mathrm{M}}}{\arg\min} \quad \frac{\text{VOLUME}}{\text{TPUT GOAL}} \left(\langle \mathrm{F}, \text{COST}^{egress} \rangle + \langle \mathrm{N}, \text{COST}^{VM} \rangle \right) \tag{4a}$$

subject to

$$\mathrm{F} \leq (\text{LIMIT}^{link} \odot \mathrm{M}) \div \text{LIMIT}^{conn} \tag{4b}$$

$$\textstyle\sum_{v \in V} \mathrm{F}_{s,v} \geq \text{TPUT GOAL} \tag{4c}$$

$$\textstyle\sum_{u \in V} \mathrm{F}_{u,t} \geq \text{TPUT GOAL} \tag{4d}$$

$$\textstyle\sum_{u \in V} \mathrm{F}_{u,v} = \sum_{u \in V} \mathrm{F}_{v,u} \quad \forall v \in V - \{\mathrm{s}, \mathrm{t}\} \tag{4e}$$

$$\textstyle\sum_{u \in V} \mathrm{F}_{u,v} \leq \text{LIMIT}_v^{ingress} * \mathrm{N}_v \quad \forall v \in V \tag{4f}$$

$$\textstyle\sum_{v \in V} \mathrm{F}_{u,v} \leq \text{LIMIT}_u^{egress} * \mathrm{N}_u \quad \forall u \in V \tag{4g}$$

$$\textstyle\sum_{v \in V} \mathrm{M}_{u,v} \leq \text{LIMIT}^{conn} * \mathrm{N}_v \quad \forall u \in V \tag{4h}$$

$$\textstyle\sum_{u \in V} \mathrm{M}_{u,v} \leq \text{LIMIT}^{conn} * \mathrm{N}_u \quad \forall v \in V \tag{4i}$$

$$N_v \leq \text{LIMIT}^{VM} \quad \forall v \in V \tag{4j}$$

5.2 Throughput maximizing overlay paths

Directly solving for a throughput maximizing path under a cost ceiling is non-trivial as we cannot use the linear reformulation of the cost objective. We can approximate a solution by solving for the minimum cost transfer plan at a range of many throughput goals. The result of this procedure is a Pareto

frontier curve (as shown in Fig. 9c). A throughput maximizing solution can be extracted from this curve. The quality of approximate solution will depend on how many samples are used. A single AWS `c5.9xlarge` instance can evaluate 100 samples in under 20 seconds.

6 Implementation of Skyplane

We implemented Skyplane in Python 3. Skyplane's planner uses the proprietary Gurobi library to solve MILP instances (used in our evaluation), but the Coin-OR library can be used instead to avoid this dependency. Our implementation currently supports the three major cloud providers: Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

We use `m5.8xlarge` instances on AWS, as smaller VM sizes were subject to burstable networking performance, which we wished to avoid [4, 7]. For consistency, we used `Standard_D32_v5` instances on Microsoft Azure and `n2-standard-32` instances on Google Cloud.

A user initiates a transfer from their application with the *Skyplane client*. The client provisions VMs in each region according to the transfer plan and runs the *Skyplane gateway* program on each VM. The gateway is responsible for actually reading from source object stores, relaying data through overlay regions and writing to destination object stores.

While transfer time is dominated by network throughput, the time to spawn gateway VMs contributes to the transfer latency. To minimize unnecessary bloat in VM images, we use compact OSes such as Bottlerocket [3] and package dependencies via Docker.

Skyplane assumes that objects are broken up into small *chunks* of approximately equal size. Applications can often do this without significant burden; for example, machine learning applications store data as `TFRecords`, which are easy to split into small chunks. This allows Skyplane to read and write data quickly from and to cloud object stores, by issuing many read/write operations in parallel to different chunks.

To mitigate the impact of straggler connections, Skyplane dynamically partitions data across TCP connections as they become ready to accept more data. This is in contrast to tools like GridFTP [1], which assign data blocks to connections in a round-robin fashion. The downside is that, for plans that use multiple overlay paths, the amount of data sent on each path may deviate from the targets computed at planning time, which could cause the actual cost of transferring data to deviate from the cost predicted by Skyplane's planner.

To avoid overflowing buffers at relay regions, Skyplane uses hop-by-hop flow control to stop reading data from incoming TCP connections when a VM's queue of chunks reaches capacity. Bufferbloat-type problems [28] are not a concern for Skyplane, with regard to queued chunks, as we pipeline transfers to optimize for throughput instead of latency.

7 Evaluation

To evaluate Skyplane, we investigate transfer time and price. We will sometimes use transfer throughput as a proxy for transfer time. In our price calculations, we include both instance cost and egress cost.

7.1 Experimental setup

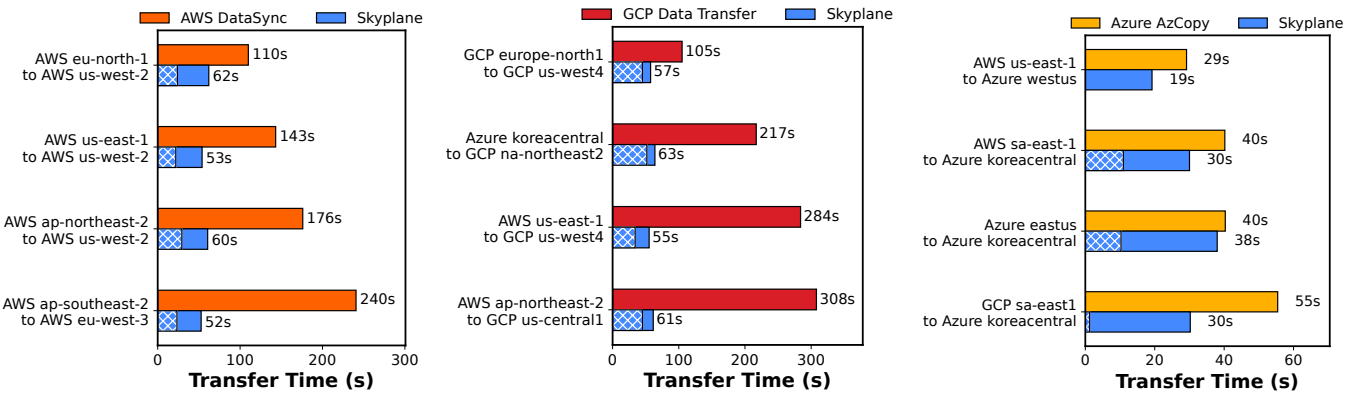
We evaluate Skyplane with 20 AWS regions, 24 Azure regions and 27 GCP regions. For all experiments, we use public IP addresses attached to the VMs for transferring data. In some cases, one can achieve better performance for intra-cloud overlay hops by using private IP addresses assigned to each VM. For GCP this yields higher performance; for AWS and Azure it may yield higher performance, but requires peering virtual networks which incurs additional fees.

Furthermore, Azure and GCP allow one to select *network tiers* to control whether data is transferred via the cloud provider's network or via the public Internet. The Skyplane prototype utilizes external IPs over standard network tiers. That said, Skyplane is not incompatible with optimizations like VPC peering or hot-potato routing tiers to reduce cost and improve performance which we leave to future work. We use the CUBIC congestion control protocol in experiments.

7.2 How much faster is Skyplane than existing data transfer solutions?

Existing cloud providers offer data transfer tools such as AWS DataSync, GCP Storage Transfer, and Azure AzCopy for low-cost transfers of bulk data into their respective clouds. These tools do not disclose what mechanisms they use to transfer data—for example, the number of VMs and TCP connections (if any) used for a transfer, or the QoS (if any) associated with the network traffic. When evaluating Skyplane, we restrict Skyplane to use at most 8 VMs in each region. This is conservative; for example, on equalizing $/GB for some routes, Skyplane could provision *up to 262 VMs* per region within DataSync's service fee. Moreover, while these services only support data transfer *into* their respective clouds, Skyplane supports data transfer between every region pair.

We consider transferring the training and validation set for ImageNet [23]. We specifically use the `TFRecords` as generated by Google as part of the Cloud TPU benchmark example [23]. We evaluate flows between regions within a single cloud (intra-provider) and between clouds (inter-provider). We expected that data transfer within each cloud provider (e.g., between AWS's `us-east-1` and AWS's `us-west-1`) to perform well as they have full visibility into their networks and can utilize private interfaces with higher performance than over public API. For example, Azure Blob Storage throttles per-object reads for third-party VMs [50]. Our experiments



(a) AWS DataSync comparison

(b) GCP Storage Transfer comparison

(c) Azure AzCopy comparison

Figure 6: **Comparison to cloud transfer systems**: The thatch pattern in each bar represents the storage I/O overhead.

did observe this behavior. However, Skyplane benefits from parallelizing the transfers.

We compare against AWS DataSync, GCP Storage Transfer and Azure AzCopy in Fig. 6. We evaluated Skyplane with a cost budget cap that is lower than the service fee for cloud transfer services in all our experiments. For each source-destination pair, we additionally measured the time to transfer procedurally-generated data using Skyplane; this allows us to break out the overhead of reading and writing to cloud storage as a "thatched" region in each bar. Skyplane significantly outperforms AWS DataSync and GCP Cloud Transfer in all configurations. In certain cases, Azure AzCopy performs about as well as Skyplane. We chose the `koreacentral` region because we expected the greatest improvements from the overlay in that region; however, storage overheads (the "thatched" regions of the bars), not networking overheads, dominated the runtime. It is possible that AzCopy avoids the Azure Blob Storage I/O overhead that dominates Skyplane's transfer time by leveraging Azure's `Copy Blob From URL` API call to download data directly into the servers running Azure Blob Storage [11].

7.3 How much faster are the overlay paths?

The planner optimally explores the trade-off between improved throughput and cost for cloud data transfers. We explore solving for the optimal transfer path between all pairs of clouds regions between all cloud providers. We evaluated 22 AWS regions, 23 unrestricted Azure regions and 27 GCP regions which leads to 5,184 possible replication routes. It would be too expensive to transfer a large amount of data along each path in order to measure the empirical achieved throughput; therefore we use the planner to generate a plan and compare the resulting plan with the direct path, both in terms of expected throughput and cost. We compute predicted costs for transferring a 50 GB dataset between each possible source and destination. We report the speedup relative to Skyplane with a direct connection between each set of instances. Notice that the baseline is itself an ablation of Skyplane and it generally outperforms existing cloud transfer services to begin with (see §7.2).

The results are shown in Fig. 7. For each pair of source and destination clouds, we show distribution of predicted throughputs across region pairs, both with Skyplane's planner restricted to the direct path and allowing Skyplane's planner to use overlay paths. The results show that Skyplane's overlay routing meaningfully improves achievable throughput between cloud regions. Note that transfers out of AWS cannot exceed 5 Gbps and transfers leaving GCP cannot exceed 7 Gbps due to these cloud providers' caps on egress bandwidth.

7.4 Where are transfer bottlenecks?

To understand how the overlay improves throughput, we characterize the fraction of transfers that are bottlenecked at each location. In Fig. 8, we visualize the percentage of transfers from §7.3 that were bottlecked at a VM in the source region, the network link leaving the source region, a VMs in optional overlay regions, a network links leaving an overlay region, and a VM in the destination region. We consider a particular location to be a bottleneck if utilization is over 99%. Multiple locations may simultaneously be a bottleneck for one transfer.

For Skyplane with overlay routing disabled, the network link from the source to the destination region is the most common bottleneck for transfers. In a small set of cases, the source VM is a bottleneck for the transfer. Generally, the direct path is not fast enough to saturate the maximum egress bandwidth limit for a VM. The overlay shifts source link bottlenecks by reducing the number of transfers bottlenecked by the source link by 32%. The bottleneck shifts to the source VM or in some cases a network link leaving an overlay region.

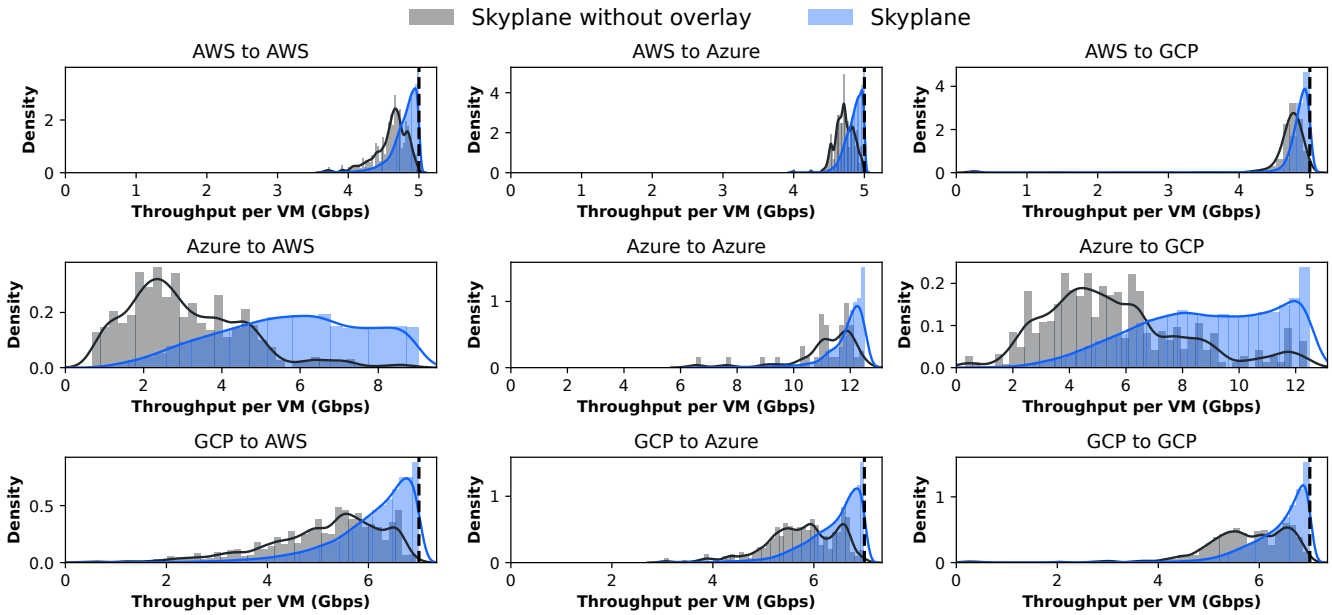


Figure 7: **Ablation of predicted overlays**: Overlay routes improve throughput per VM instance. We visualize the distribution of predicted throughput by the planner with all optimizations enabled (Skyplane) and with all optimizations except for overlay routing (Skyplane without overlay). The AWS and GCP egress limits are displayed with a dashed line.

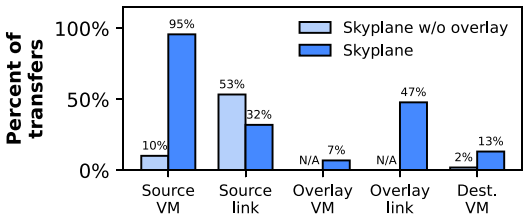


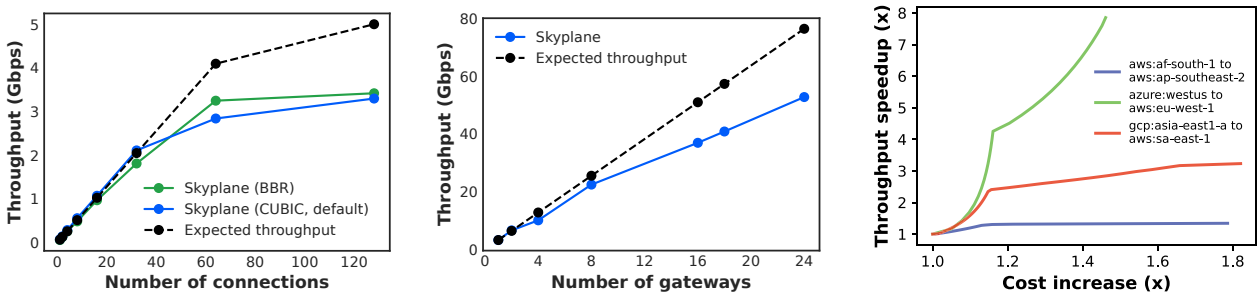
Figure 8: **Transfers bottlenecked at each location**: For transfers in Fig. 7, we visualize what percentage of transfers were bottlenecked at various locations. Enabling the overlay shifts bottlenecks from the network to the VM.

7.5 Skyplane microbenchmarks

Impact of parallel TCP connections Fig. 9a shows the impact of varying the number of parallel TCP connections used to transfer data between VMs. For this experiment, the source VM was located in AWS `ap-northeast-1` and the destination VM was located in AWS `eu-central-1`. Skyplane transfers 32 GB of synthetic, procedurally-generated data in these experiments to avoid incurring object store I/O overheads and thereby isolate network performance. The black dashed line shows the expected throughput, assuming that bandwidth scales linearly with the number of parallel TCP connections up to AWS' 5 Gbps egress cap. The blue line shows Skyplane's achieved throughput, and the green line uses Skyplane's achieved throughput using the BBR congestion control algorithm (used only this experiment). For this experiment, the source VM was located in AWS `ap-northeast-1` and the destination VM was located in AWS `eu-central-1`. Skyplane's achieved throughput plateaus below the 5 Gbps egress cap, and 64 connections is enough to come close.

Impact of parallel VMs Fig. 9b shows the impact of using multiple VMs in each region to achieve higher aggregate throughput. The black dashed line shows the expected throughput, assuming that bandwidth scales linearly with the number of VMs. Although Skyplane's performance is significantly less than the expected throughput for a large number of gateways, the graph shows that using parallel VMs is an effective way for Skyplane to scale its aggregate bandwidth. Additionally, using parallel VMs is a particularly valuable tool in the context of inter-cloud transfers, as Skyplane can use multiple VMs in one cloud provider to circumvent the egress limit. For example, for an overlay hop from an AWS region to an Azure region, one may allocate many instances in AWS but few in Azure, to account for AWS' egress cap.

Trade-off between cost and throughput Fig. 9c shows the impact on overlay path throughput as the price budget is varied. We adjusted the cost budget afforded to the planner (x-axis), and plot the throughput predicted by the planner for the output plan (y-axis). We show three routes where the overlay benefits are considerable (Azure `westus` to AWS `eu-west-1`), good (GCP `asia-east1-a` to AWS `sa-east-1`) and minimal (AWS `af-south-1` to AWS



(a) TCP connections versus throughput (b) Number of gateway VMs versus throughput (c) Predicted planner throughput versus cost

Figure 9: **Skyplane ablations**: We evaluate the impact of parallel TCP connections, parallel gateway VMs and overlay cost.

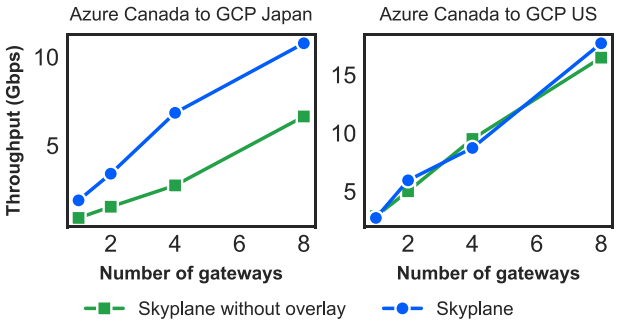


Figure 10: **Scaling VMs versus overlay**: In situations where the direct path is slow, the overlay is faster than simply scaling the number of VMs used alone.

`ap-southeast-2`). As the cost budget increases, Skyplane uses increasingly complex overlay topologies, adding new overlay paths as the instance limit (1 VM, in this case) is saturated in each region. Each elbow in the plot (e.g. 1.2× for the Azure to AWS route) represents a point where Skyplane adds a new overlay route via a faster but more costly region. At some point, the planner cannot increase throughput further as the overlay network is saturated.

Is it better to use VMs to form overlay paths or parallelize the direct path? Given a limited number of VMs (§4.3), a natural question is whether it is better to use those VMs to form overlay paths or to parallelize the direct path. In Fig. 10, we evaluate Skyplane with and without the overlay enabled for various numbers of VMs in the context of an inter-continental transfer and an intra-continental transfer. For the inter-continental transfer, using the VMs with overlays enabled provides a 2.08× geomean speedup compared to using those VMs to parallelize the direct path. However, for the intra-continental transfer, there is little benefit to using VMs in overlay paths (1.03× geomean speedup).

Table 2: **Comparison with academic baselines**: Skyplane outperforms RON's path selection heuristic implemented in Skyplane [8].

Method	Time	Throughput	Cost
GCT GridFTP [1, 10] (1 VM)	133s	1.03 Gbps	$1.40
Skyplane (1 VM, direct)	**73s**	**1.71 Gbps**	$1.40
Skyplane w/ RON routes (4 VMs) [8]	21s	6.02 Gbps	$2.27
Skyplane (cost optimized, 4 VMs)	32s	3.88 Gbps	$1.56
Skyplane (throughput optimized, 4 VMs)	**16s**	**8.07 Gbps**	$1.59

7.6 Comparison against academic baselines

In Table 2, we compare Skyplane with RON [8] and the community-maintained fork [10] of GridFTP [1] for a 16 GB data transfer from Azure `East US` to AWS `ap-northeast-1`. To isolate network throughput from I/O overheads, we benchmark the transfers without object stores (VM to VM only).

We use the open-source GCT fork of GridFTP [10]. Although GCT GridFTP theoretically supports striped transfers across multiple machines, we were unable to find a supported non-commercial implementation. To make a fair comparison, we run both GCT GridFTP and Skyplane with a single VM per region. Skyplane is 1.6× faster than GCT GridFTP.

We implement RON's path selection heuristic in Skyplane to compare overlays between RON and Skyplane. Our results show that Skyplane has better cost and throughput than RON. Skyplane with routes from RON's path selection heuristic achieves 3.5× higher throughput than Skyplane with a single VM but at 62% cost overhead. Skyplane's planner instead finds overlay paths with up to 4.7× higher throughput than the direct path within a 14% cost overhead.

8 Related Work

Skyplane builds on the overlay network literature [8, 16, 58]. As discussed in §1, Skyplane adapts classical overlays to the cloud setting, accounting for the price of network bandwidth

and leveraging the elasticity of cloud resources. CRONets [16] briefly discusses cost, but focuses on comparing cloud-based options to private leased lines. Unlike Skyplane, it does not discuss how to manage the cost of cloud resources. Lai et al. [46] find relay regions improve throughput in AWS when utilizing a single TCP connection but find the 2 Gbps instance NIC limit from their chosen instance class limits the benefit of overlay paths. CloudCast [56] examines the use of triangle overlays in the cloud to reduce network latency while Skyplane examines throughput.

Several existing efforts [27, 49, 55] aim to optimize bulk data transfers by reducing the amount of data transferred. Such techniques are complementary to Skyplane; one can first apply these techniques to reduce the amount of data to transfer, and then apply Skyplane's techniques to transfer that reduced data efficiently. Unlike Skyplane, these works do not use cost when selecting the network path to use for a transfer.

Another line of research aims to improve bulk data transfers by improving resource management. GridFTP [1] is a tool for wide-area transfers that techniques such as using multiple machines and TCP connections. GridFTP sends all data over the direct path and does not utilize overlays. Khanna et al. [40] explore application of network overlays to GridFTP but do not consider elasticity and egress price in the cloud. Other solutions, like PSockets [59], also use parallel TCP connections for high bandwidth. Pied Piper [14] also explored how cloud resource elasticity could be used to improve cloud data transfers, but utilize a different mechanism than Skyplane.

There have been decades of improvements and optimizations at the transport layer to make TCP perform better in large-BDP settings within TCP itself [2, 15, 17, 33], while others concern operating system support for TCP [20, 24, 48]. Improvements to TCP are complementary to Skyplane. CodedBulk [61] uses network coding to complete bulk-transfer multicast jobs quickly [61]. Another set of research [18,63,64] investigates how to schedule urgent and non-urgent bulk transfers to meet a transfer's deadline. None of these techniques consider the cost of transferring data in the cloud.

Traffic engineering (TE) systems, like Google's B4 [35,36] and BwE [43] and Microsoft's SWAN [34], Cascara [57], and BlastShield [42], are used internally by cloud providers to navigate the cost-performance trade-off in their wide-area networks. The precise nature of the trade-off differs from Skyplane in two ways. First, TE systems consider costs in terms of the *bandwidth* provisioned (e.g., the cost of installing long-distance cables [36], or the 95th percentile bandwidth for peering links [57]). In contrast, Skyplane considers cost from the perspective of a cloud customer, where the cost depends on the volume and not bandwidth of data transferred. Second, TE systems like Cascara [57] assume a static topology and aim to reallocate bandwidth to save cost, with a global view of a single provider's network. Skyplane optimizes a single user's transfer, with the ability to use overlay regions in multiple cloud providers' networks.

Skyplane has similarities to Content Delivery Networks (CDNs) [58], most notably in that both make use of overlay networks. However, Skyplane's focus is different from CDNs. CDNs focus on caching objects near users, in order to provide low network latency. In contrast, Skyplane focuses on transferring large amounts of data quickly, with a focus on achieving high bandwidth rather than low network latency such as in workloads like ML training and database replication. CDNs are more suitable for workloads where popular objects need to be replicated to many regions so that geo-distributed users can access them with low network latency.

One application of bulk transfers is VM migration [19, 32, 37, 45] that balance VM downtime and bandwidth consumed when transferring VMs. Supercloud [37] uses a network of vSwitches in an overlay that maintains TCP connections upon migration, not to provide high bandwidth at low cost.

Some existing research efforts and commercial products focus on bulk transfer jobs that are not time-critical. For example, Laoutaris et al. [47] propose techniques to reduce the cost of transferring data for delay tolerant applications.

Cloud providers provide services for bulk transfer, such as AWS Snowball [62], Azure Data Box [12], and GCP Transfer Appliance [21], that have users ship their data via physical drives via the postal service. For sufficiently large transfers, these services may allow data to be transferred into the cloud datacenter more quickly than using the Internet.

9 Conclusion

This paper explores how to efficiently transfer data between cloud regions using cloud-aware overlay networks. Our key observation is that principles from overlay networks can be applied to the cloud setting to identify high-quality network paths that lead to fast transfer times. However, adapting principles from overlay networks to the cloud setting requires consideration of cloud resource pricing, most notably the egress fees associated with network bandwidth. Skyplane manages the trade-off between performance and cost when performing bulk data transfer. It works by accepting a user- or application-provided constraint on performance and solving a mixed integer linear program (MILP) to obtain the optimal data transfer plan. Skyplane can reduce the time to transfer data by up to 5.0× at minimal additional cost.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rachee Singh, for their helpful feedback. We also thank Asim Biswal, Jason Ding, Daniel Kang, Vincent Liu, Xuting Liu, and Anton Zabreyko. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF GRFP Award DGE-1752814, and gifts from Amazon, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, and VMWare.

References

[1] William Allcock, John Bresnahn, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus striped GridFTP framework and server. In *Supercomputing*, 2005.

[2] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. RFC 2488, 1999.

[3] Amazon Web Services. Amazon Bottlerocket OS. https://aws.amazon.com/bottlerocket, 2022.

[4] Amazon Web Services. Amazon EC2 instance network bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html, 2022.

[5] Amazon Web Services. Aws DataSync: online data transfer and migration. https://aws.amazon.com/datasync, 2022.

[6] Amazon Web Services. EC2 on-demand instance pricing. https://aws.amazon.com/ec2/pricing/on-demand/, 2022.

[7] Amazon Web Services. General purpose instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html#general-purpose-network-performance, 2022.

[8] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP*. ACM, 2001.

[9] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR*, 2021.

[10] GCT authors. Grid community toolkit. https://github.com/gridcf/gct, 2022.

[11] Microsoft Azure. Copy blob from URL. https://docs.microsoft.com/en-us/rest/api/storageservices/copy-blob-from-url, 2022.

[12] Microsoft Azure. Microsoft Azure Data Box. https://azure.microsoft.com/en-us/products/databox/, 2022.

[13] Microsoft Azure. Scalability and performance targets for blob storage. https://learn.microsoft.com/en-us/azure/storage/blobs/scalability-targets, 2022.

[14] Aran Bergman, Israel Cidon, Isaac Keslassy, Noga Rotman, Michael Schapira, Alex Markuze, and Eyal Zohar. Pied Piper: Rethinking Internet data delivery. In *CoRR*, 2018.

[15] D. Borman, B. Braden, and V. Jacobson. TCP extensions for high performance. RFC 7323, 2014.

[16] Chris X. Cai, Franck Le, Xin Sun, Geoffrey G. Xie, Hani Jamjoom, and Roy H. Campbell. CRONets: Cloud-routed overlay networks. In *ICDCS*. IEEE, 2016.

[17] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: congestion-based congestion control. In *CACM*, 2017.

[18] Bin Bin Chen and Pascale Vicat-Blane Primet. Scheduling deadline-constrained bulk data transfers to minimize network congestion. In *CCGrid*. IEEE, 2007.

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[20] David D. Clark. The structuring of systems using upcalls. In *SOSP*, 1985.

[21] Google Cloud Platform. Google Cloud transfer appliance. https://cloud.google.com/transfer-appliance/docs/4.0/overview, 2022.

[22] AzCopy contributors. Azure storage AzCopy. https://github.com/Azure/azure-storage-azcopy, 2022.

[23] TensorFlow contributors. Training ResNet on Cloud TPU. https://cloud.google.com/tpu/docs/tutorials/resnet, 02 2022.

[24] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *SOSP*, 1993.

[25] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *Trans. Networking*, 1999.

[26] Forrester/Virtustream. A clear multicloud strategy delivers business value.

[27] Sebastian Frischbier, Alessandro Margara, Tobias Freudenreich, Patrick Eugster, David Eyers, and Peter Pietzuch. McCAT: Multi-cloud cost-aware transport. In *EuroSys Poster Track*, 2014.

[28] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the Internet. *CACM*, 2012.

[29] Google Cloud. All networking pricing | Virtual Private Cloud | Google Cloud. https://cloud.google.com/vpc/network-pricing, 2022.

[30] Google Cloud. Network bandwidth | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/network-bandwidth, 2022.

[31] Google Cloud Platform. Storage transfer service. https://cloud.google.com/storage-transfer-service, 2022.

[32] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: Agile VM handoff for edge computing. In *SEC*, 2017.

[33] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. In *SIGOPS*, 2008.

[34] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nandury, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[35] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *SIGCOMM*, 2018.

[36] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Us Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed Software Defined WAN. In *SIGCOMM*, 2013.

[37] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Supercloud: Opportunities and challenges. In *SIGOPS*, 2015.

[38] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *SIGCOMM*. ACM, 2014.

[39] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *STOC*, 1984.

[40] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, Rajkumar Kettimuthu, P. Sadayappan, Ian Foster, and Joel Saltz. Using overlays for efficient data transfer over shared wide-area networks. In *Supercomputing*, 2008.

[41] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. *SOSP*, 2003.

[42] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BlastShield. In *NSDI*. USENIX, 2022.

[43] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM*, 2015.

[44] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*, chapter 3, page 308. International 6th edition, 2013.

[45] H. Andrés Lagar-Cavilla, Joseph A. Whitney, Roy Bryant, Philip Patchin, Michael Brudno, Eyal de Lara, Stephen M. Rumble, M. Satyanarayanan, and Adin Scannell. Snowflock: Virtual machine cloning as a first-class cloud primitive. *ACM Trans. Comput. Syst.*, 29(1), feb 2011.

[46] Fan Lai, Mosharaf Chowdhury, and Harsha Madhyastha. To relay or not to relay for Inter-Cloud transfers? In *HotCloud*, 2018.

[47] Nikolaos Laoutaris, Georgios Smaragdakis, Pablo Rodriguez, and Ravi Sundaram. Delay tolerant bulk data transfers on the Internet. In *SIGMETRICS*, 2009.

[48] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *SOSP*, 1993.

[49] Miguel Matos, António Sousa, José Pereira, and Rui Oliveira. CLON: Overlay network for clouds. In *WDDM*, 2009.

[50] Microsoft Azure. Scalability and performance targets for blob storage. https://docs.microsoft.com/en-us/azure/storage/blobs/scalability-targets, 2021.

[51] Microsoft Azure. Pricing - bandwidth | Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/bandwidth/, 2022.

[52] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: A platform for high-performance Internet applications. *SIGOPS*, 2010.

[53] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling tcp throughput: A simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, page 303–314, New York, NY, USA, 1998. Association for Computing Machinery.

[54] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[55] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *NSDI*, 2014.

[56] Noga H. Rotman, Yaniv Ben-Itzhak, Aran Bergman, Israel Cidon, Igor Golikov, Alex Markuze, and Eyal Zohar. CloudCast: Characterizing public clouds connectivity. *CoRR*, 2022.

[57] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with CASCARA. In *NSDI*, 2021.

[58] Ramesh K. Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. Overlay networks: An Akamai perspective. *Advanced Content Delivery, Streaming, and Cloud Services*, 2014.

[59] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing*. ACM/IEEE, 2000.

[60] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM*, 2001.

[61] Shih-Hao Tseng, Saksham Agarwal, Rachit Agarwal, Hitesh Ballani, and Ao Tang. CodedBulk: Inter-datacenter bulk transfers using networkcoding. In *NSDI*, 2021.

[62] Amazon Web Services. AWS Snowball. https://aws.amazon.com/snowball, 2022.

[63] Yu Wu, Zhizhong Zhang, Chuan Wu, Chuanxiong Guo, Zongpeng Li, and Francis C. M. Lau. Orchestrating bulk data transfers across geo-distributed datacenters. *Trans. Cloud Computing*.

[64] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing deadlines for inter-datacenter transfers. In *EuroSys*, 2015.

[65] Yuchao Zhang, Junchen Jiang, Ke Xu, Xiaohui Nie, Martin J. Reed, Haiyang Wang, Guang Yao, Miao Zhang, and Kai Chen. BDS: A centralized near-optimal overlay network for inter-datacenter data replication. In *EuroSys*, 2018.

Electrode: Accelerating Distributed Protocols with eBPF

Yang Zhou* *Harvard University*
Zezhou Wang* *Peking University*
Sowmya Dharanipragada *Cornell University*
Minlan Yu *Harvard University*

Abstract

Implementing distributed protocols under a standard Linux kernel networking stack enjoys the benefits of load-aware CPU scaling, high compatibility, and robust security and isolation. However, it suffers from low performance because of excessive user-kernel crossings and kernel networking stack traversing. We present Electrode with a set of eBPF-based performance optimizations designed for distributed protocols. These optimizations get executed in the kernel before the networking stack but achieve similar functionalities as were implemented in user space (e.g., message broadcasting, collecting quorum of acknowledgments), thus avoiding the overheads incurred by user-kernel crossings and kernel networking stack traversing. We show that when applied to a classic Multi-Paxos state machine replication protocol, Electrode improves its throughput by up to 128.4% and latency by up to 41.7%.

1 Introduction

Distributed protocols such as Paxos [37] for state machine replication are important building blocks for highly-available distributed applications. For example, Google's Chubby [6] uses a variant of classic Paxos [37] and Multi-Paxos [36] to implement a highly-available lock service, powering their business-critical GFS [16] and Bigdata [7] applications. Google's globally-distributed database Spanner [8] and Microsoft's data center management tool Autopilot [22] also run Paxos protocols to maintain their high availability.

Existing high-performance implementation of distributed protocols tends to be radical and not readily-deployable. DPDK-based kernel-bypass approaches [27, 79] allow direct access to the underlying NIC hardware, but require application developers to build their own networking stack and maintain compatibility with the evolving kernel networking stack [75]. DPDK also dedicates CPU cores to busily poll the network interface for I/O competition, sacrificing CPU resources and wasting energy during low I/O loads. This is especially a problem for embedded devices [51, 60, 70] where CPU resources are rare. Other approaches co-design specialized distributed systems with niche network hardware including RDMA [11, 28, 76], FPGA [23], SmartNICs [66], and programmable switches [25]. These advanced hardware devices are not widely available in today's cloud environments, and systems built on top of them tend to be difficult to design, implement, and deploy [27].

Instead, we would prefer the widely-deployed and well-maintained standard kernel networking stack that also provides load-aware CPU scaling and strong security and isolation among different applications [5, 59]. However, implementing distributed protocols under the standard kernel networking stack often gives poor performance. The root causes are the high packet processing overhead in the kernel networking stack and heavy communications in distributed protocols. Our measurement shows that over half of CPU time is spent on the kernel networking stack in a typical Paxos deployment (§2); such overhead is mainly caused by user-kernel crossings (and associated context switches) and traversing the kernel networking stack. Moreover, when using a classic leader-based Multi-Paxos protocol [43, 54] to implement state machine replication, e.g., with five replicas, processing a single request would require the leader node to send/receive *fourteen* messages in total (see Figure 1a), suffering from the kernel stack overhead fourteen times[1].

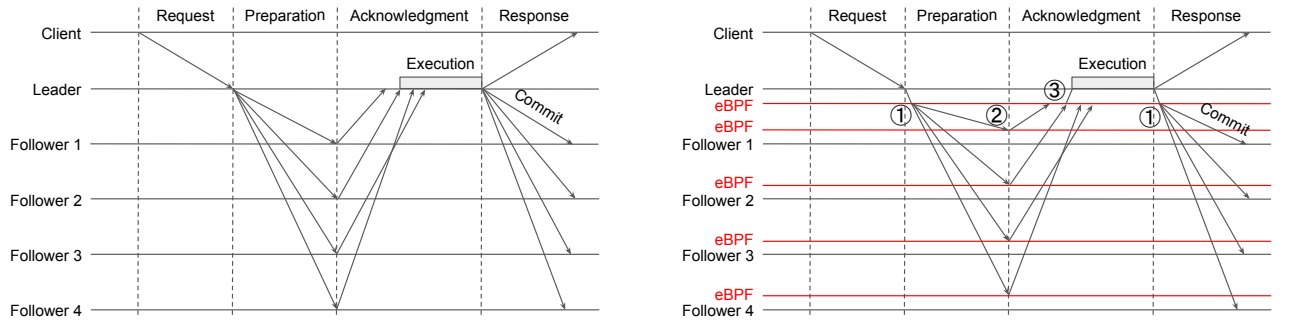
In this paper, we focus on accelerating Paxos protocols inside data centers by offloading protocol operations to the kernel via eBPF (i.e., extended Berkeley Packet Filter) [46, 49]. eBPF allows safely executing customized yet constrained functions inside the kernel at various locations. Similar to kernel bypass, the offloaded operations get executed immediately after the NIC driver receives the packet, without user-kernel crossing and kernel networking stack traversing. Unlike kernel bypass, eBPF is an OS-native mechanism such that eBPF-offloaded operations do not sacrifice security and isolation properties while amenable to load-aware CPU scaling without busy-polling.

The key challenge is, given the constrained programming model of eBPF, *which parts of Paxos protocols to offload that can greatly reduce kernel stack overhead while being implementable and efficient in eBPF*. Note that the eBPF subsystem requires every offloaded function to be statically verified to guarantee kernel security, which only allows limited instructions, bounded loops, static memory allocation, etc.

Our insight is that common operations of Paxos protocols, e.g., message broadcasting and waiting on quorums, incur large kernel stack overhead, but are naturally offloadable by existing eBPF programming capacity. For example, Paxos protocols require a leader node to broadcast preparation messages to follower nodes; if implemented using multiple `sendto()` syscalls conventionally, it would incur multiple user-kernel

*Equal contribution

[1] Linux io_uring [1] can reduce user-kernel crossings, but cannot reduce kernel stack traversing (see §8 for details).



(a) The Multi-Paxos/Viewstamped Replication protocol.

(b) Electrode-accelerated Multi-Paxos/Viewstamped Replication.

Figure 1: Normal case execution of the leader-based Multi-Paxos/Viewstamped Replication protocol vs. Electrode-accelerated one with 5 replicas. Electrode offloads ①: message broadcasting (§4.1), ②: fast acknowledging (§4.2), and ③: wait-on-quorums (§4.3) to eBPF to reduce the kernel networking stack overhead.

crossings and kernel networking stack traversing. Instead, eBPF has a `bpf_clone_redirect()` [45] function that enables us to clone an in-kernel packet buffer multiple times and send them to different destinations; this eBPF-based message broadcasting only needs one user-kernel crossing and one kernel networking stack traversing. Besides broadcasting, we also utilize eBPF to reduce unnecessary wake-ups of user-space applications when waiting on quorums, and optimize how follower nodes handle preparation messages by early acknowledging before entering the kernel networking stack. The final result of these three eBPF-based optimizations is Electrode[2] (Figure 1b). When applying Electrode to a classic leader-based Multi-Paxos protocol, it achieves up to 128.4% higher throughput and 41.7% lower latency. This translates into up to 112.9% higher throughput and 19.3% lower latency for a Paxos-based transactional replicated key-value store.

Electrode has some limitations: it currently targets protocols implemented in UDP and relies on application-level retransmission to handle packet loss. This works well for Paxos protocols whose requests are usually small enough to fit into a single packet, and data center environments where packet loss is rare [28, 61].

2 Background

2.1 Consensus Protocols

Distributed protocols that coordinate and synchronize among a collection of nodes have become an indispensable part of the modern data center application stack. Storage systems in data centers replicate data for fault tolerance and availability. For instance, Berkeley-DB [55] uses a consensus protocol to replicate its logs over a set of distributed replicas. Transactional storage systems like H-Store [71] and Spanner commit their updates to multiple replicas in order to be more failure resilient. At the heart of most replication-based systems is a consensus protocol [36, 37, 43, 54] that ensures that operations execute in a consistent manner across all replicas.

[2]Electrode is a Pokémon that has a high speed score.

Here, we consider a set of nodes either functioning as clients or replicas. Clients are the users of a particular application-level service hosted by a collection of replicas. It should also be noted here that clients could often just be other servers within the same data center. Clients submit requests to one or more replicas, which triggers a round of agreement to occur. Paxos is a common protocol that is used to obtain an agreement in the presence of node and network failures.

Since applications often need to reach agreements on many client requests, servers use agreement protocols like Paxos to implement a state machine-based abstraction that requires all the replicas to process the exact same set of client requests in the same order. This log-based state machine abstraction is often optimized by the use of a leader. In a leader-based protocol, all the instances of agreement on client requests are mediated through the leader and the leader also dictates the order of the log.

In Figure 1a, we have an example of VR (Viewstamped Replication), a leader-based Multi-Paxos protocol that uses Paxos for running agreements on individual requests. The leader here is responsible for ordering all client requests by assigning sequence numbers to them, and the followers (non-leader nodes) are responsible for responding to the leader and applying all the updates in the order in which they're sequenced by the leader.

The leader is also responsible for initiating agreement by sending out a preparation message to all the other replicas. The leader then waits for a quorum of acknowledgments from all the other replicas before broadcasting a commit message to all the replicas. A successful iteration of this two-round protocol ensures that all non-failed replicas have the client's request. And the sequence number assigned by the leader determines the order in which all the replicas process this client's request. This pattern of broadcasting and waiting on quorums is common in many distributed protocols [38, 39, 80].

To gain more insights into the performance of the Multi-Paxos/VR protocol under the standard Linux kernel networking stack, we measure the CPU time breakdown of the leader node, shown in Table 1. There is 44.7% +

Function Name	Description	% CPU
`__libc_sendto()`	User function to send packets.	44.7
⊢ `sock_sendmsg()`	Kernel function to send packets.	32.2
⊢ `__alloc_skb()`	Allocate `sk_buff` for packets.	4.5
⊢ `dev_queue_xmit()`	Transmit `sk_buff`.	6.8
⊢ bookkeeping	For sock, IP, and UDP layers.	**20.9**
⊢ user-kernel crossing	Interrupt, mode switching, etc.	**12.5**
`__libc_recvfrom()`	User function to recv packets.	11.8
⊢ `sock_recvmsg()`	Kernel function to recv packets.	5.7
⊢ user-kernel crossing	Interrupt, mode switching, etc.	**6.1**

Table 1: CPU time breakdown for the leader node when running the Multi-Paxos/Viewstamped Replication protocol with 5 replicas. See §7 for measurement setup.

$11.8\% = 56.5\%$ of time spent on the `__libc_sendto()` and `__libc_recvfrom()` functions, while $20.9\% + 12.5\% + 6.1\% = 39.5\%$ of time spent on user-kernel crossing and kernel networking stack bookkeeping. These numbers concrete our previous motivations that implementing distributed protocols under kernel networking stack incurs significant overhead on user-kernel crossings and kernel stack traversing (while eBPF can potentially save them).

2.2 eBPF and Hooks

BPF (i.e., Berkeley Packet Filter) [49] enables user-space applications to customize packet filtering in the kernel. A BPF program, written in some predicates on packet fields, is triggered by the kernel event that a packet arrives at a NIC driver. Once triggered, the BPF program will run inside a kernel virtual machine with limited registers and scratch memory, and a reduced instruction set [49]. For example, the well-known *tcpdump* [20] command-line packet analyzer is based on BPF.

eBPF extends the BPF by increasing the number of registers and adding stack memory. The increased number of registers and stack memory enable the eBPF program to execute more complex operations—the developers can use a C-like language to express customized operations. This C-like code is compiled into an eBPF bytecode by the Clang/LLVM toolchain and runs inside the kernel virtual machine via just-in-time compilation.

eBPF also introduces various powerful in-kernel data structures called *eBPF maps*, which, paired with various helper functions, are used to store and maintain states across multiple triggering of eBPF programs. Example eBPF maps include array, per-CPU arrays, queues, stacks, and hashMaps [46]. These maps are also used to communicate among different eBPF programs and between eBPF programs and user-space processes. Each eBPF map can be identified by a `map_path` through the file system, e.g., `/sys/fs/bpf/<map_name>`, and user-space processes can access a map based on its path.

The kernel events that can trigger eBPF programs are called *eBPF hooks*. There are many hooks existing in Linux kernels

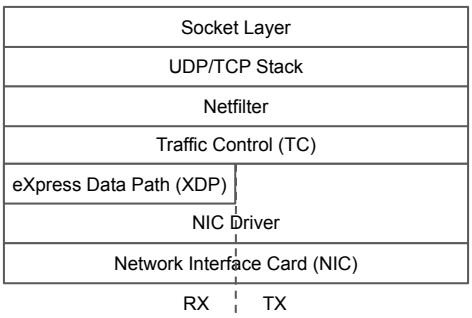


Figure 2: Linux kernel networking stacks and eBPF XDP/TC hooks.

and various device drivers, such as hooks in NIC drivers right after it receives a packet. User-space applications can attach eBPF programs to these eBPF hooks to customize the handling of corresponding kernel events.

Constrained programming model: An eBPF program needs to go through strict verification by an in-kernel eBPF verifier before attaching to an eBPF hook and running inside the kernel. The verification process does a static sanity check to make sure the eBPF program does not have out-of-bounds memory access (i.e., safety) and will always terminate (i.e., liveness). The verifier basically enumerates all possible cases of every conditional branch and loop to make sure every execution path meets the safety and liveness requirements. Because the verification tends to be time-consuming, each eBPF program can only contain up to 1 million instructions. For a larger eBPF program, the developer needs to split it into multiple smaller eBPF programs and uses *tail calls* to let one eBPF program call another one in a continuation manner.

Because of the strict verification process, dynamical memory allocation is not supported in eBPF programs; instead, eBPF programs can only rely on eBPF maps with capacity *specified statically* to maintain in-kernel states.

Due to these limitations, eBPF is commonly used in kernel tracing, profiling, and monitoring [3, 63] and L2-L4 low-level packet processing such as load balancing [14].

XDP (eXpress Data Path) [21, 64] technique implements an in-kernel eBPF hook that enables attached eBPF programs to process RX packets directly out of the NIC driver (Figure 2). Such processing gets triggered before any `sk_buff` [31] allocation or entering software socket queues, thus bypassing any higher-level networking stacks (e.g., UDP, TCP, Socket). XDP-based packet processing normally achieves comparable throughput and latency as DPDK-based kernel-bypass packet processing [21].

TC (Traffic Control) [47] is another important layer/hook which locates right after the XDP (Figure 2). In the TC layer, the `sk_buff` data structure has already been allocated by the kernel networking stack, thus the performance of TC-based packet processing will be slightly worse than XDP. However, the TC hook allows attached eBPF programs to process both RX and TX packets and manipulate the packet `sk_buff`. For

example, one can clone the `sk_buff` for a TX packet and thus implements packet broadcasting in the TC layer.

3 Electrode Overview

Electrode is a framework for offloading Paxos protocols under kernel networking stack to in-kernel eBPF programs to reduce user-kernel crossings and kernel networking stack traversing. Electrode has two goals in designing its eBPF offloads: 1) largely reducing kernel stack overhead to improve performance, and 2) carefully partitioning user- and kernel-space functionalities to keep offloads implementable and efficient inside the eBPF subsystem.

To achieve the first goal, Electrode carefully extracts generic and performance-critical fast-path operations from Paxos protocols to offload to the eBPF. As shown in Figure 1b, Electrode offloads message broadcasting (§4.1), fast acknowledging (§4.2), and wait-on-quorums (§4.3). These operations, if purely implemented in the user space, would involve many user-kernel crossings and kernel stack traversing, causing significant kernel stack overhead as shown in §2. Once implemented in the eBPF, message broadcasting allows the leader node to efficiently send preparation and commit messages to multiple follower nodes, by cloning and sending packets in the kernel; fast acknowledging enables follower nodes to buffer preparation messages in the kernel, and quickly respond to the leader node without involving user-space processes; wait-on-quorums lets the leader node eBPF program wait for a quorum number of acknowledgments from follower nodes, and only notify user-space processes once. Moreover, to simplify how user-space applications use these eBPF-based accelerations, Electrode further designs a set of user-space APIs (Table 2). Each API corresponds to one operation that Electrode offloads to the eBPF, and is used to invoke the offloaded function or retrieve eBPF processing results.

To achieve the second goal, Electrode keeps complicated slow-path operations of Paxos protocols in the user space. Specifically, Electrode leaves the procedures of failure recovery and handling message loss/reordering (i.e., gap agreement) to user-space applications, using similar mechanisms as VR [43] and NOPaxos [40]. These procedures involve accessing dynamic ranges of memory, which is hard to implement in eBPF under the static verification (see §8 for details).

Overall, Electrode has the following workflow: first, user-space applications attach eBPF programs to various hook locations corresponding to a network interface; then, user-space applications use Electrode APIs to invoke eBPF-offloaded functions or retrieve eBPF processing results; finally, the eBPF programs intercept and process target packets in the kernel without going through the networking stack or user-space applications (i.e., Paxos protocols in our case). Electrode targets accelerating the handling of messages that can fit into one ethernet packet (i.e., up to 9KB for jumbo frames). This is well-suited for locks, barriers, and configuration parameters [25, 78] that Paxos protocols commonly maintain. Non-target packets still go through the stack and reach user-space applications, without impacting applications' other operations or protocol semantics.

Finally, we note that Electrode does not aim to offload every operation of Paxos protocols to the eBPF, because of eBPF's constrained programming model vs. the diverse set of operations that Paxos protocols and related services could have. For example, currently, Electrode does not offload client-facing request/response handling. There are two reasons: 1) Paxos clients normally serialize/deserialize their requests using widely-used libraries such as protocol buffers [19]; however, parsing or constructing protocol buffers is difficult in eBPF, because it involves complex pointer arithmetics and conditional branches which cannot easily pass the eBPF verifier. 2) client-facing requests/responses are normally embedded into application-level services like the Chubby lock service [6], but it is hard and inefficient to implement them in eBPF because of the strict eBPF verifier and the lack of dynamic memory allocation. We discuss more on Electrode's offloading decisions in §8.

4 Electrode Designs

4.1 Message Broadcasting in TC

In Paxos protocols, one-to-all message broadcasting is widely used. For example, 1) the leader node sends preparation messages to all follower nodes, and 2) (after receiving enough acknowledgments from followers) the leader node sends commit messages to all follower nodes.

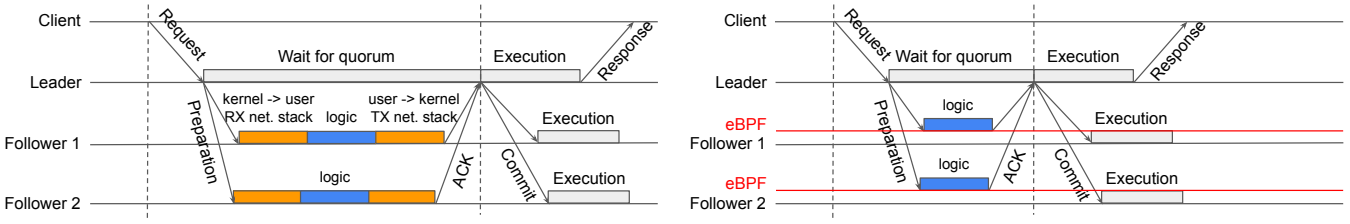
To implement the above message broadcasting, the most common way is sending the same message multiple times in the user space to different destinations. However, the overhead (i.e., user-kernel crossing and kernel networking stack traversing) of this implementation on the leader node increases linearly as the number of followers increases, while the overhead on each follower node remains constant. Thus, the leader node essentially becomes the system bottleneck, e.g., Table 1 has shown that 44.7% of CPU time is spent on sending messages on the leader node.

An alternative implementation is to use IP multicast [42, 68, 77]. However, IP multicast normally requires support from the underlying network switches (e.g., storing a large number of multicast group-table entries for the whole network topology) [68, 77] or considerable modifications of the Linux networking stack [42].

Electrode approach: Electrode provides a flexible host-based broadcasting solution by utilizing eBPF on the TC hook. Here, we require the eBPF program that implements broadcasting operations to attach to the TC hook, because only the TC hook can intercept and process outgoing packets (§2.2). After attaching the eBPF program, user-space applications can call the `elec_broadcast()` function shown in Table 2 with specified `sock_fd`, message, and a list of destination IPs to broadcast the message to these destinations through the socket.

Function Name	Arguments	Output	Description
elec_broadcast	sock_fd, message, {dst_ips}	status	Broadcasts <message> to all destinations through <sock_fd>
elec_poll_message	map_path	messages	Polls buffered messages from an eBPF-maintained in-kernel ring buffer identified by <map_path>
elec_check_quorum	received_message	bool	Checks if <received_message> (acknowledgment) indicates quorum reaching

Table 2: Electrode user-space APIs.



(a) Without fast acknowledging.

(b) With fast acknowledging in eBPF.

Figure 3: Fast acknowledging in eBPF reduces Paxos request latency. This example follows Figure 1, but omits followers 3 and 4 for brevity.

Under the hood, the eBPF program makes clones of the message packet using the `bpf_clone_redirect()` [45] helper function, modifies the destination addresses of cloned packets accordingly, and sends these packets out. The benefit of cloning packets and broadcasting in the kernel compared with sending the same message multiple times in the user space is that we only need to cross the user-kernel boundary and traverse the UDP and socket layer once.

Handling message loss: Electrode relies on application-level timeout and retransmission to handle message loss, similar to modern RPC-based applications [13, 69]. Specifically, if the leader node does not receive a response after a certain time of sending a request, it will resend the request; once a request experiences several timeouts, the leader node will mark the destination node as dead and start Paxos failure recovery. An alternative approach to handling message loss is doing retransmission in the kernel, which could save user-kernel context switching overheads, but such savings become marginal as packet loss happens rarely in data centers [28, 61]; it would also involve complex message buffer management in kernel/eBPF, hurting performance.

4.2 Fast Acknowledging in XDP

As shown in Figure 3a, a significant portion of Paxos request latency comes from the round-trip delay between the leader node and follower nodes. Note that the ACK messages in this figure mean Paxos protocol acknowledgments, not TCP acknowledgments. For Paxos protocols under the kernel networking stack, this round-trip delay includes not only physical propagation and transmission delay, but also the delay caused by the kernel networking stack (i.e., user-kernel crossing and networking stack traversing). As the fabric latency of nowadays data center network reaches a few tens of microseconds [48] or sub-ten microseconds [18, 27], the latency of the kernel networking stack, which is also around sub-ten microseconds [59], becomes non-negligible.

Electrode approach to reducing the Paxos request latency is to optimize the preparation handling in follower nodes by directly buffering the preparation messages into an in-kernel log and early acknowledging to the leader node. At the same time, user-space applications asynchronously poll and consume the buffered messages from the log, using the `elec_poll_message()` function shown in Table 2. Under the hood, the function calls a corresponding eBPF syscall to poll messages in batches, amortizing kernel crossing overhead. This asynchrony does not break the correctness of Paxos protocols because as long as a preparation message gets buffered into the log, it will be eventually processed by the user-space Paxos protocols, and the message processing order has been specified by the sequence number assigned by the leader node. Figure 3b shows that this approach removes *two* user-kernel crossings and networking stack traversing from the critical path of the Paxos request.

Note that not every preparation message can be handled using fast acknowledging; in some non-critical path cases (e.g., message loss/reordering, and node failure) where the eBPF program cannot handle because of its constrained programming model, our eBPF program can detect them and directly forward preparation messages to user-space Paxos protocols (detailed in §6).

In-kernel log implementation: The in-kernel log temporally stores incoming early-acknowledged preparation messages, which are polled and consumed by user-space applications concurrently. To implement this in-kernel log, we use a special eBPF map named `BPF_MAP_TYPE_RINGBUF` [30] (introduced from Linux kernel 5.8). This map implements an efficient multi-producer single-consumer (MPSC) ring buffer using shared memory and a lightweight spinlock, where we can have multiple writers in eBPF and one reader in user space. Based on our measurement, the time of pushing a preparation message into the ring buffer is roughly equal to memcpying this message, in cases without any lock contention. Note

that the in-kernel ring buffer also has a fixed size, because eBPF does not support dynamic memory allocation; in case it becomes full, the eBPF program can detect them and directly forward preparation messages to user-space applications.

4.3 Wait-on-Quorums in TC + XDP

Another common operation in Paxos protocols is the leader node waiting for a quorum number of acknowledgments (ACKs) from follower nodes (i.e., wait-on-quorums). Assume there are $2f+1$ replicas including one leader node and $2f$ follower nodes. In most Paxos protocols, once the leader collects f ACKs from different follower nodes, the Paxos request is considered *committed*.

Conventionally, wait-on-quorums is implemented by the user-space applications that receive all ACKs and count towards the quorum number. However, each acknowledgment handling incurs the overhead of the user-kernel crossing and traversing the kernel networking layer. The total overhead of handling all ACKs is linear to the number of follower replicas (i.e., $2f$). Moreover, among these $2f$ ACKs, only the first f ones are required to commit a Paxos request.

Electrode approach: Electrode moves the leader-side wait-on-quorums operations to the eBPF, requiring only one user-kernel crossing and one networking stack traversing. Electrode maintains an array of bitsets (and other metadata) in eBPF, each of which indicates whether a Paxos request has reached the quorum. Electrode only forwards ACK messages that indicate reaching the quorum to the user-space applications, while dropping others. Electrode maps each Paxos request to a specific bitset by using the unique increasing sequence number assigned by the leader node (§2). Note that we use the bitset instead of a counter to check if the quorum gets reached; this is because a timed-out preparation request could cause duplicate ACK messages from follower nodes, and we want to avoid double counting.

Electrode maintains the bitset setting and clearing (i.e., zeroing out) operations through two eBPF programs hooked at TC and XDP layers, respectively. The TC-hooked eBPF program intercepts each outgoing preparation message and clears the indexed bitset, while the XDP-hooked eBPF program intercepts each incoming ACK message from follower nodes and sets the bit corresponding to the follower node's index in replicas.

As shown in Listing 1, the `tc_ebpf` function/program intercepts each outgoing preparation message and clears a specific bitset indexed by the sequence number in each message. Line 6 checks if it is the first time to intercept a preparation message corresponding to this Paxos request, by comparing the `seq` stored along this bitset and the `seq` extracted from the message; if so, it updates the stored `seq` in the array and clears the bitset that may have been used by previous Paxos requests (line 17-18).

The `xdp_ebpf` program intercepts each incoming ACK message, updates the indexed bitset, drops most of the ACK packets, and only forwards packets to user-space applications that indicate reaching quorum or array overflow (explained in the next paragraph). Lines 17-18 check if this bitset corresponds to the `seq` in the ACK message, and set the proper bitset bit if so. Line 19 further checks if this ACK message reaches the quorum: if so, lines 20-21 will mark the packet as quorum-reaching and forward it to user-space applications; otherwise, line 22 just drops the packet. Once the user-space applications receive a quorum-reaching packet—checked by calling the `elec_check_quorum()` function shown in Table 2, it can directly consider this Paxos request as committed.

```
1  # Processing outgoing preparation message
2  # pkt: the packet of the message
3  # seq: unique increasing sequence number (from pkt)
4  def tc_ebpf(pkt, seq):
5      idx = seq % array_length
6      if array[idx].seq != seq
7          array[idx].seq = seq
8          array[idx].bitset.clear()
9      forward(pkt) # to follower node
10
11 # Processing incoming ACK message
12 # pkt   : the packet of the message
13 # seq   : unique increasing sequence number (from pkt)
14 # node_i: follower node index (from pkt)
15 def xdp_ebpf(pkt, seq, node_i):
16     idx = seq % array_length
17     if array[idx].seq == seq
18         array[idx].bitset.set(node_i)
19         if array[idx].bitset.count() == f
20             pkt.mark_quorum_reach(true)
21             forward(pkt) # to user-space application
22         else: drop(pkt)
23     else: # bitset overwritten by tc_ebpf
24         pkt.mark_quorum_reach(false)
25         forward(pkt)
```

Listing 1: Maintaining the fixed-length bitset array to achieve wait-on-quorums in eBPF. Each bitset operation is also protected by a spinlock; we omit it here for simplicity.

Handling array overflow: In some cases, a bitset might be overwritten by the `tc_ebpf` because of the fixed size of the bitset array. `xdp_ebpf` detects such array overflow in lines 17&23; once detected, lines 24-25 will mark the packet as *not*-quorum-reaching and forward it to user-space applications. Once the user-space applications receive a not-quorum-reaching packet, it resends the preparation messages to all follower nodes and waits for ACKs again. In practice, the leader node could limit the number of in-flight preparations while provisioning a large bitset array, such that the array overflow does not normally happen.

RSS: Electrode supports RSS (Receive-Side Scaling) which distributes incoming packets to different NIC queues and CPU cores. Specifically, Electrode has two receive-side optimizations: fast acknowledging and wait-on-quorums. For fast acknowledging, the eBPF programs in the follower node could maintain separate in-kernel ring buffers on different cores to avoid synchronization overhead during log append-

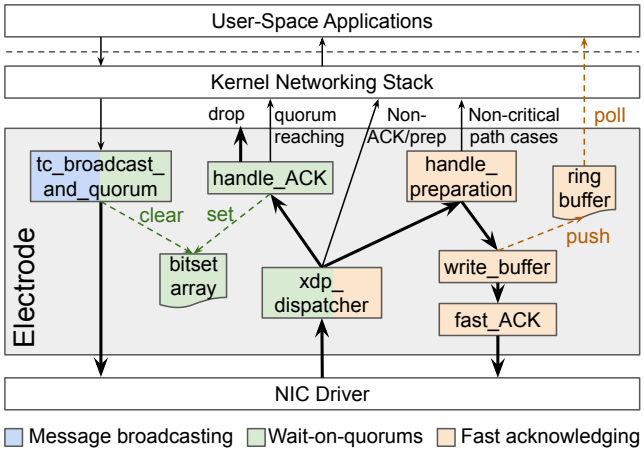


Figure 4: eBPF program structure of Electrode. The thickness of solid lines indicates traffic volume (the thicker, the higher).

ing, and use spinlocks to synchronize accesses to small shared in-kernel states (e.g., `ebpf_seq` in §6); the user-space applications asynchronously pull messages from all ring buffers, and process messages following the order specified by their embedded sequence numbers. For wait-on-quorums, the eBPF programs in the leader node could use atomic instructions to count how many ACKs it has received and check if the quorum is reached.

5 Electrode Implementation

Electrode is prototyped with six eBPF programs written in a restricted C language, and we utilize the Clang/LLVM toolchain for compiling source code to eBPF bytecode. These eBPF programs consist of 500 lines of C code in total. Application developers can also customize their own eBPF programs based on needs, e.g., only processing packets with a specific source port like [25]. Our prototype does not implement the RSS handling yet.

Figure 4 shows the structure of the six eBPF programs. One program can transfer its control flow to the next program via the eBPF tail call. We break the implementation into these six programs because of 1) avoiding breaking the instruction limits in the eBPF verifier (§2.2), and 2) modularity. In the following, we describe each program in detail.

- `tc_broadcast_and_quorum`: This program intercepts outgoing preparation messages. It implements the message broadcasting mechanism (§4.1) and the `tc_ebpf` function in Listing 1 for wait-on-quorums (§4.3). For broadcasting, we generate multiple clones of the preparation packets using the `bpf_clone_redirect()` [45] helper function.
- `xdp_dispatcher`: This program checks the types of incoming messages and calls corresponding message handlers. It only intercepts the ACK (only received on the leader node) and preparation (only received on follower nodes) messages, and calls the corresponding `handle_ACK` and `handle_preparation` programs. It directly forwards other types of messages to user-space applications.
- `handle_ACK`: This program implements the `xdp_ebpf` function in Listing 1 for wait-on-quorums (§4.3). In common cases, it drops most ACK messages, and only forwards the quorum-reaching ACK messages to user-space applications.
- `handle_preparation`: This program implements various checks to detect non-critical path cases where it should forward messages to user-space applications (§4.2). In normal cases (mostly), it will call `write_buffer` to begin `fast_ACK`.
- `write_buffer`: This program stores message/packet data into an in-kernel log for user-space applications to poll and consume. As mentioned earlier, We use the eBPF ring buffer [30] to implement the log data structure. This program then calls the `fast_ACK` program.
- `fast_ACK`: This program reuses and modifies the received packet buffer to create an ACK packet and sent it out. This requires swapping the src-dst IP addresses and filling the corresponding fields of the ACK message.

6 Apply Electrode to Multi-Paxos

Optimizing throughput: We apply the eBPF-based message broadcasting (§4.1) and wait-on-quorums (§4.3) mechanisms to the leader node in the Multi-Paxos protocol. This implies two throughput optimizations: 1) when the leader node sends out preparation messages to follower nodes, it relies on eBPF to broadcast these messages instead of sending them one by one; and 2) when the leader node is waiting for a quorum number of ACK messages from follower nodes, it only needs to process the quorum-reaching ACK message while the other ACK messages are pruned/dropped by the eBPF program. These two optimizations largely reduce the number of user-kernel crossings and kernel networking stack traversing, thus alleviating the CPU bottleneck on the leader node and improving system throughput.

Optimizing latency: We apply the eBPF-based fast acknowledging mechanism (§4.2) to each follower node in the Multi-Paxos protocol. In normal cases (e.g., without packet loss/reordering, and all nodes are alive), the preparation messages from the leader node are quickly buffered and acknowledged by the eBPF program in the follower nodes, bypassing both the kernel networking stack and the user-space Multi-Paxos protocol. This reduces the commit latency of each Multi-Paxos request by twice the time of user-kernel crossing and kernel networking stack traversing.

Detecting non-critical path cases in fast acknowledging: As mentioned in §4.2, there are some non-critical path cases in fast acknowledging where the eBPF program must detect them and forward the incoming packets to the user-space Paxos protocols. To understand why non-critical path cases happen and how to detect them, we first elaborate on the Multi-Paxos/VR protocol shown in §2, following the literature [43]. In the Multi-Paxos protocol, the leader node assigns

each Multi-Paxos request a unique and strictly increasing sequence number, `seq`. Each replica including both the leader node and follower nodes maintains locally a `view` number, a `status`, and its last observed `seq`; each message sent by a replica will piggyback these three variables. The `view` number indicates which (leader) election epoch this replica is in; the `status` indicates if this replica is during a leader election (`status_viewchange`), recovering (`status_recovering`), or normal state (`status_normal`). This protocol requires a follower node to only process a preparation message if the node is in the normal state, and the message has a matched `view` and strictly increasing `seq`; otherwise, the follower node needs to drop the message, or execute a complex view-change or state-transfer procedure [43,54]. Therefore, the non-critical path cases for Multi-Paxos are:

1. the follower is during a leader election or recovering,
2. the follower receives a message with an unmatched `view` that is either (a) stale or (b) newer,
3. the follower receives a message with a non-strictly-increasing `seq` caused by message (a) loss/reordering or (b) duplication.

These cases only happen when replicas fail or join, or messages get lost/reordered, which is less common in data centers [27,61].

To detect these non-critical path cases in eBPF, we maintain an `ebpf_status`, an `ebpf_view`, and an `ebpf_seq` variable in the eBPF program using the eBPF map. In particular, these three variables can be updated by the user-space Multi-Paxos protocols to reflect the current protocol state. Listing 2 shows the detection pseudocode. Line 5 detects case 1, and line 6 detects case 2(a); for these two cases, the eBPF program needs to drop the packet. Line 7 detects cases 2(b) and 3(a), and forwards the packet to the user space to execute the view-change or state-transfer procedure. For case 3(b), i.e., `msg_seq < ebpf_seq + 1`, the eBPF program function replies an ACK (line 11), because it could be a re-transmitted preparation message due to timeout.

Handling the cases 2(a)&3(a) in fast acknowledging is tricky, because it (i.e., forwarding packets to the user space for processing) involves the concurrency between the user-space protocols and the kernel-space eBPF program, while eBPF only supports map-based communication *but not synchronization* between the user and kernel. Our approach is to let the user-space protocols *detach* the eBPF program from the hook while executing the view-change or state-transfer procedure. Specifically, once a user-space protocol receives a preparation message corresponding to the case 2(a) or 3(a), it detaches the eBPF program, then it finishes the view-change or state-transfer procedure, next it updates the `ebpf_status`, `ebpf_view`, and `ebpf_seq` properly, and finally it reattaches the eBPF program. This guarantees the cases 2(a)&3(a) are exclusively handled by the user-space protocol, avoiding the synchronization between the user and kernel. An alternative approach to achieving the same effect as eBPF detach-reattach is to use an eBPF map with a branch testing before any Electrode logic. The first packet in the non-critical path can update this map atomically and let all following packets directly go to the user-space application (i.e., closing Electrode optimizations); later, the user-space application can update this map to reopen Electrode optimizations.

```
1  # pkt     : the packet of the preparation message
2  # msg_view: view piggybacked by the pkt
3  # msg_seq : unique increasing sequence number (from pkt)
4  def detect_non_crit_path_cases(pkt, msg_view, msg_seq):
5      if (ebpf_status != status_normal): drop(pkt)
6      if (msg_view < ebpf_view): drop(pkt)
7      if (msg_view > ebpf_view or msg_seq > ebpf_seq + 1):
8          forward(pkt)
9      if (msg_seq == ebpf_seq + 1):
10         append_log(++ebpf_seq, pkt)
11     reply_ack(pkt)
```

Listing 2: Detecting non-critical path cases during fast acknowledging for Multi-Paxos. Assume the protocol works in a single core, in line with prior Paxos work [40,44,61].

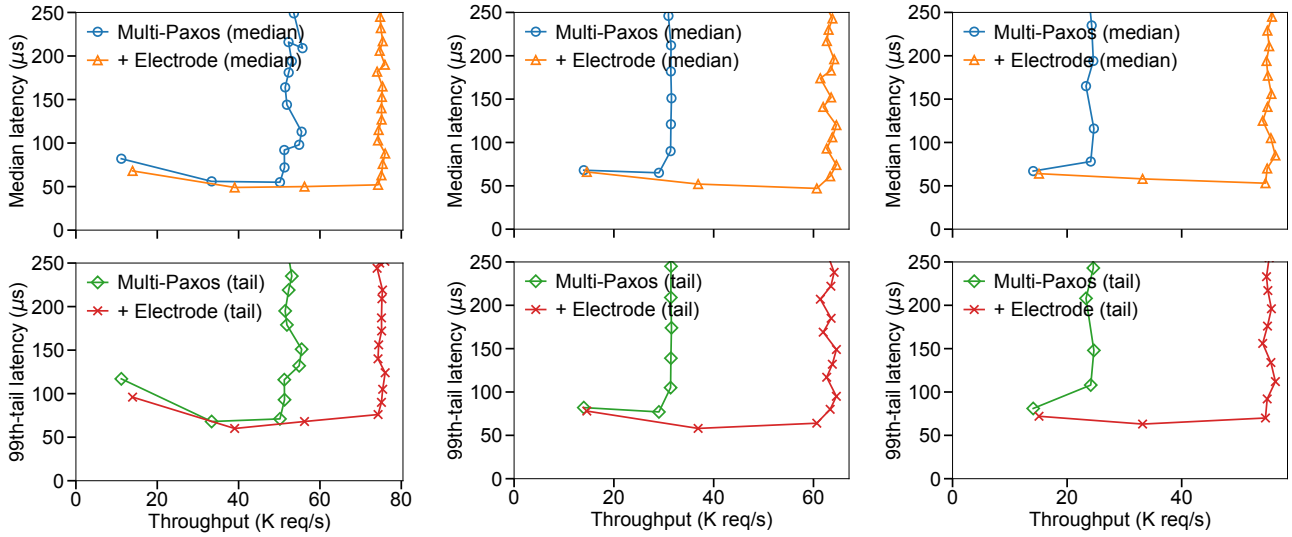
There are a few caveats: 1) After the user-space protocol detaches the eBPF program, it needs to poll the in-kernel ring buffer again, in case the eBPF program still appends a few messages to the ring buffer before detaching. Note that the eBPF map can outlive the eBPF program, as long as the user-space process holds a reference to it, because its lifetime is managed through reference counting [50]. 2) While the user-space protocol is setting the `ebpf_seq` value and is about to reattach the eBPF program, some preparation packets might just pass the eBPF hook location but have not been processed by the user-space protocol, e.g., queued in the socket layer. In this case, the user-space protocol actually has set a smaller `ebpf_seq` value in the map; once the eBPF program gets reattached, it will trigger more case 3(a) (lines 7&8). Our solution to this problem is: after the user-space protocol finishes the view-change or state-transfer procedure, it first sends a `stop_sending_preparation` message to the leader node to stop it from sending preparation messages, then it polls the socket to drain and process any queued packet, next it sets the proper `ebpf_seq` value, finally it sends a `resume_sending_preparation` message to the leader node to resume sending preparation messages, and reattaches the eBPF program. These two messages should be sent using reliable transport like TCP to handle packet loss.

Generalizability: Electrode's eBPF-based optimizations are generic to many more distributed protocols, which normally consist of broadcasting and wait-on-quorums operations. More discussions can be found in Appendix A.

7 Evaluation

This section answers the following questions:

1. How do Electrode and each optimization improve the performance of the Multi-Paxos protocol (§7.1 and §7.2)?



(a) 3 replicas. **(b)** 5 replicas. **(c)** 7 replicas.

Figure 5: Performance comparison of the Multi-Paxos protocol vs. Electrode-accelerated one with different numbers of replicas.

2. How does Electrode improve the performance of real-world Paxos-based applications (§7.3)?
3. How does Electrode save kernel stack overhead (§7.4)?
4. How does Electrode compare to kernel-bypassing (§7.5)?

Testbed setup: We use eight `xl170` servers from CloudLab [12], each of which has a ten-core Intel E5-2640v4 CPU at 2.4 Ghz, 64GB memory, and a Mellanox ConnectX-4 25 Gbps NIC. Each server runs an unmodified Ubuntu 20.04 OS with kernel v5.8.0. All servers are connected using a two-level topology: five Mellanox 2410 as rack switches (each connecting to forty `xl170` servers) and one Mellanox 2700 as the spine switch. One server is dedicated as the client server that generates Paxos requests, and other servers run the Paxos protocol with 3/5/7-replica configurations. By default, we configure each server to use one core for interrupt processing and another core for Paxos processing, following the performance optimizations in [41]. We disable irqbalance to avoid out-of-order packet deliveries as much as possible (which would hurt Paxos performance), in line with prior Paxos work [40,44,61]. Unlike prior Paxos work [32,40,61], we *do not* use IP multicast which requires specialized support from the network (§4.1).

Measurement methodology: The client server runs multiple Paxos/application clients, and each client sends Paxos/application requests in either a closed-loop or open-loop manner. In closed-loop experiments, each client sends the next request once it receives the response of the last request; we vary the number of clients and measure the corresponding throughput, and median and 99th-percentile tail latency, in line with prior Paxos work [40,44,61]. In open-loop experiments, each client sends requests one by one at a specific time interval, such that the overall request rate reaches a specified value; we use enough clients (i.e., they could saturate the Paxos servers), specify different request rates, and measure the corresponding CPU utilization of each replica node.

Comparisons: We use the Multi-Paxos/VR protocol implementation in the SpecPaxos [61] open-sourced code [35] as the baseline, and optimize it using Electrode. We also run a transactional replicated key-value store similar to the one in SpecPaxos [61] atop the baseline Multi-Paxos protocol and Electrode-accelerated Multi-Paxos protocol. All implementation uses the standard UDP stack and socket layer from the Linux kernel.

7.1 Overall Results

Figure 5a, 5b, and 5c show the performance comparison of the Multi-Paxos protocol and the Electrode-accelerated one when using 3, 5, and 7 replicas, respectively. In each figure, we vary the number of clients sending Multi-Paxos requests in a closed-loop manner, and report throughput and median and 99th-percentile tail latency. All curves eventually hit a "hockey stick" in their median or tail latency growth when the system reaches its maximum throughput.

Throughput: the Electrode-accelerated Multi-Paxos protocol achieves 34.9%, 104.8%, and 128.4% higher maximum throughput than the original Multi-Paxos protocol under 3, 5, and 7 replicas, respectively. The large throughput improvements benefit from the eBPF-based broadcasting and wait-on-quorums which reduce the kernel stack overhead significantly on the leader node. With more replicas, the improvement becomes more significant. This is because, for each Multi-Paxos request, the leader node will send more preparation and commit messages, and handle more ACK messages; thus the eBPF-based broadcasting and wait-on-quorums can save more user-kernel crossings and kernel networking stack traversing.

Latency: the Electrode-accelerated Multi-Paxos protocol achieves 12.5%, 20.0%, and 25.6% lower median latency than the original Multi-Paxos protocol with 2 clients (before

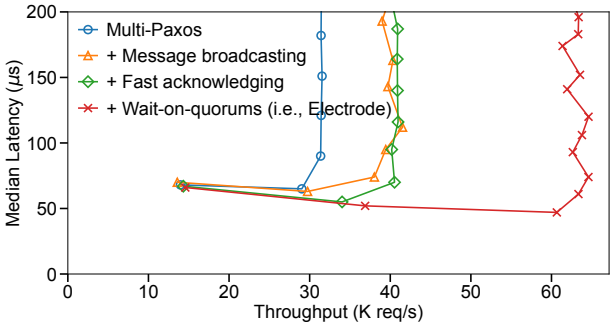
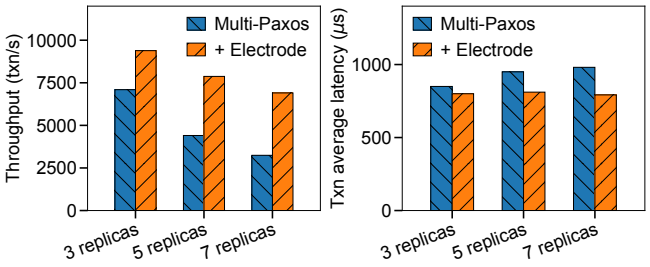


Figure 6: Performance impact of different optimizations for Electrode-accelerated Multi-Paxos protocol (with 5 replicas).



(a) Throughput. **(b)** Latency (one client).

Figure 7: Performance comparison of a transactional key-value store atop the Multi-Paxos protocol vs. Electrode-accelerated one.

the "hockey stick") under 3, 5, and 7 replicas, respectively; the corresponding tail latency is 11.8%, 24.7%, and 41.7% lower. The latency reduction mostly comes from the fast acknowledging in the follower nodes, which, for each Multi-Paxos request, saves the time of two user-kernel crossings, two kernel networking stack traversing, and one wake-up of the user-space process. With more replicas, the latency reduction becomes larger. This is because the fast acknowledging bypasses user-space process scheduling and avoids unpredictable scheduling delays [48] by the OS; for the original Multi-Paxos, with more follower nodes, such unpredictable scheduling delays would raise the chance of follower nodes straggling, thus increasing commit latency. Besides, for Multi-Paxos under 3/5 replicas and Electrode under 7 replicas, their latency curves first decline a bit and arrive at the lowest point, then rise and reach the "hockey stick". This is because, under lower throughput, the Linux scheduler would schedule the Paxos process off the CPU more frequently, while under higher throughput, the Paxos process is mostly scheduled on the CPU.

7.2 Performance Gain Breakdown

Figure 6 shows the performance impact of different optimizations for the Electrode-accelerated Multi-Paxos protocol with 5 replicas. Similar to §7.1, we vary the number of clients sending Multi-Paxos requests in a closed-loop manner, and report the throughput and latency. eBPF-based message broadcasting improves the maximum throughput of the Multi-Paxos protocol by 31.7%; fast acknowledging further reduces the median latency by 4.3%-12.7% (before the "hockey stick"); finally, wait-on-quorums improves the maximum throughput by 57.7%. Overall, we find that the two throughput optimizations (i.e., eBPF-based message broadcasting and wait-on-quorums) have almost no impact on the median latency, while the latency optimization (i.e., fast acknowledging) does not nearly impact maximum throughput. This division of labor demonstrates good modularity of each optimization design in Electrode, and each design can be independently used to accelerate more distributed protocols as shown in Table 4.

7.3 Application Performance

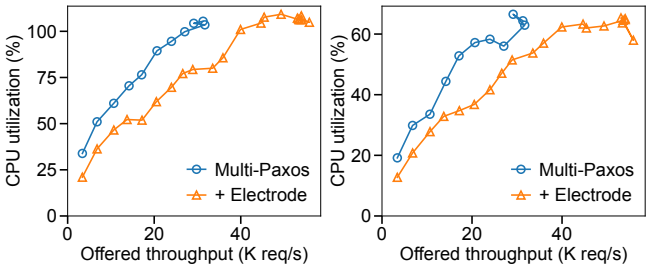
To demonstrate how Electrode can bring benefits to real-world Paxos-based applications, we run a transactional replicated key-value store (similar to the one in SpecPaxos [61]) atop the Multi-Paxos protocol and Electrode-accelerated one. This key-value store supports serializable transactions using two-phase commit and optimistic concurrency control (OCC). Clients use `BEGIN_TXN`, `COMMIT_TXN`, `ABORT_TXN`, `SET`, and `GET` operations to express transactions. We use a synthetic workload derived from the Retwis application [56]—an open-source Twitter clone. This workload consists of four types of transactions with different ratios, and each one issues different numbers of `GET` and `PUT` operations. The workload details can be found in Table 2 of [80]. We vary the number of clients that execute transactions in a closed-loop manner, and measure the maximum throughput these clients can achieve and the average latency under one client.

Figure 7a and 7b shows the maximum throughput and average latency of the key-value store atop the Multi-Paxos protocol vs. Electrode-accelerated one under different numbers of replicas, respectively. Overall, Electrode improves the key-value store throughput by 32.3%-112.9% and latency by 5.9%-19.3%. The improvement becomes larger with more replicas, due to the similar reasons described in §7.1. The latency of the key-value store atop the original Multi-Paxos gradually increases with more replicas, while Electrode-accelerated one's remains relatively stable, because the former is more vulnerable to follower nodes straggling (§7.1).

7.4 CPU Utilization

One design goal of Electrode is to reduce the kernel networking stack overhead (§3) when implementing Paxos protocols. Thus, in this subsection, we study the impact of Electrode on CPU utilizations, which indicates how much kernel stack overhead gets reduced.

Figure 8a and 8b show the CPU utilization of the leader node and follower nodes, respectively, for the Multi-Paxos protocol and Electrode-accelerated one with different offered throughput. The experiments are done in an open-loop manner to control the offered throughput when measuring CPU utilization. The CPU utilization covers both the core handling



(a) The leader node. **(b)** Follower nodes in average.

Figure 8: CPU utilization comparison of the Multi-Paxos protocol vs. Electrode-accelerated one (with 5 replicas).

interrupts and the core running Paxos. With higher offered throughput, the CPU utilization gradually increases, demonstrating the load-aware CPU scaling provided by the kernel networking stack (§1). We note that for DPDK-based Multi-Paxos protocol implementation, the CPU utilization would be always 100% because DPDK busily polls the network interface. Overall, Electrode reduces the CPU utilization by 22.7%-38.0% on the leader node and 16.0%-35.7% on the follower nodes, benefiting from the reduced user-kernel crossings and kernel stack traversing.

7.5 Comparison with Kernel-Bypassing

Electrode still handles client-facing requests/responses and initiates message broadcasting using the Linux kernel networking stack (§3); thus, it will achieve lower performance than pure kernel-bypassing approaches. This subsection compares the performance of Electrode with a kernel-bypassing baseline, aiming to reveal the performance upper bound of kernel-based approaches and identify the possible improvements for future work.

We choose Caladan [15] and use its high-performance DPDK-based UDP stack to implement our kernel-bypassing baseline. Similar to Caladan, our baseline dedicates one CPU core for packet polling and another core for running the Paxos protocol. We also configure the Caladan runtime to never idle the Paxos core even under low request load.

Table 3 compares the latency and throughput of kernel-based Multi-Paxos and the kernel-bypassing one. To exclude the latency incurred by the client-side kernel stack, we tested all three Paxos implementations with a request generator implemented using Caladan. Electrode achieves 1.4-1.6x lower latency and 2.0x higher throughput than vanilla Linux, but it still has 2.2x higher latency and 2.4x lower throughput compared to pure kernel-bypassing. The performance gap between Electrode and kernel-bypassing exists, because there are still substantial Paxos messages going through the kernel networking stack in Electrode. In particular, our profiling shows that, on the leader node, around 59.5% CPU time is spent on `__libc_sendto()` caused by frequent `dev_queue_xmit()` and `sk_buff` clones. Although eBPF-based broadcasting reduces a significant number of user-kernel crossings and sock-

	Lowest median/99p latency	Maximum throughput
Vanilla Linux	59/69 μs	32 K req/s
Electrode	38/49 μs	65 K req/s
Kernel-bypassing	17/22 μs	154 K req/s

Table 3: Performance comparison of kernel-based Multi-Paxos vs. kernel-bypassing one (with 5 replicas).

/UDP/IP layer traversing, it cannot fundamentally optimize how the Linux kernel manages NICs and packet buffers. Finally, we note that Electrode's goal is to provide generic eBPF-based accelerations for distributed protocol implementations that stick to kernel networking stacks because of compatibility, security, isolation, and elastic CPU scaling.

An additional evaluation regarding how the interrupt coalescing feature of modern NICs impacts Electrode is in Appendix B.

8 Discussion and Future Work

Electrode's offloading decisions: Electrode decides to leave four components of the Multi-Paxos protocol to the user space: 1) failure recovery, 2) handling packet loss and reordering, 3) handling client-facing requests/responses, and 4) executing application-specific operations after reaching the consensus. The first two components involve complex operations on the log, e.g., scanning the log and sending inconsistent entries to other replicas, and inserting missing log entries received from others. These operations require accessing dynamic ranges of log entries, which would fail the eBPF static verification. The last two involve complex serialization/deserialization and application-level operations (see §3). We note that it is possible to offload these four components into eBPF by modifying the kernel eBPF subsystem or verifier—we leave this as future work.

How to improve the eBPF subsystem for offloading? Verifying memory accesses more smartly could make more application operations offloadable. The current eBPF verifier only allows accessing static ranges of memory, which hinders many applications with complex memory accessing behaviors. Another useful construct in eBPF would be dynamic memory allocation, which could ease the maintenance of more advanced data structures in eBPF. To avoid memory leaks, a possible solution could be enforcing Rust-style single-owner memory semantics.

io_uring [1] was recently introduced into the Linux kernel to support efficient batching of asynchronous I/Os via shared memory between the user and kernel space, thus reducing the overhead of frequent user-kernel crossings. Therefore, when implementing Paxos protocols using io_uring, it can help reduce the overhead of message broadcasting, which accounts for 12.5% of CPU time based on Table 1. However, each preparation and ACK message still goes through the

full Linux networking stack and wakes up user-space applications, incurring significant overhead; Electrode can be used together with io_uring to reduce such overhead. A recent work XRP [82] shares a similar view regarding io_uring.

Electrode on shared environments: Electrode requires attaching eBPF programs to the network interface, which then processes every packet accordingly. However, multiple Electrode applications might share the same NIC and attach different eBPF programs that might interfere with each other. We can use the SR-IOV (Single Root IO Virtualization) feature that is widely available in modern NICs [2, 9] to avoid such interference. SR-IOV virtualizes a physical network interface into multiple virtualized ones; the Electrode eBPF program can be attached to only one virtualized interface, without impacting others (e.g., used by non-Paxos applications). Besides SR-IOV, Electrode can also check the port numbers of incoming packets in eBPF, and only execute optimizations if the port numbers belong to target Paxos applications.

Accelerating leader-less consensus protocols using eBPF: Electrode targets at leader-based consensus protocols such as Paxos [37] and its variants [36, 43, 54], because they are the most-used ones by modern distributed applications [6, 8, 22]. Electrode's eBPF-based optimizations could also be applied to leader-less consensus protocols, e.g., EPaxos [52], Mencius [4], SD-Paxos [81], etc. For example, replicas in EPaxos could acknowledge preparation messages earlier in an eBPF program before entering the kernel networking stack, thus reducing latency. We leave the exploration of applying Electrode to leader-less consensus protocols as future work.

9 Related Work

Kernel-bypass and hardware offloading: Overheads of the monolithic kernel networking stack have spurred various attempts to design new kernel-bypassed networking stacks like mTCP [24], eRPC [27], Demikernel [79] and more [15, 29, 33, 48, 57, 67], which attempt to eliminate the kernel from the I/O datapath. But all of these solutions are not backward compatible with solutions that already use the standard kernel networking stack, and they incur more costs in terms of CPU cycles and energy during low I/O loads due to busy-polling. Electrode attempts to leverage eBPF to unclog some of the bottlenecks in the kernel networking stack for distributed protocols without completely having to shift to kernel-bypassed stacks.

Similarly, network offload solutions attempt to offload I/O-intensive operations to specialized hardware, e.g., RDMA [11, 28, 76], FPGA [23], SmartNICs [66], and programmable switches [10, 25]. But they come with limited interfaces for programmability and need custom hardware to be installed.

Co-designing distributed systems with networks: There have been attempts to optimize distributed systems by co-designing them with data center networks for improved performance. SpecPaxos [61] attempts to leverage the natural order of packet delivery in data centers to optimize the ordering of messages needed for state machine replication. NoPaxos [40] uses in-network switches to sequence packets for a similar purpose. Eris [39] further applies in-network sequencing to distributed transactions to avoid coordination overhead. These are orthogonal ways to optimize distributed systems and can be used in conjunction with Electrode.

Distributed protocols in data centers: Data centers have a variety of distributed protocols that are deployed for fault tolerance and data consistency. These include replication protocols like Mencius [4], EPaxos [52], chain replication [74], SDPaxos [81], and transaction protocols like TAPIR [80] and Meerkat [72]. Since many distributed protocols share similar patterns of communication like broadcasting and quorum responses, Electrode can be applied to speed up these distributed protocols as well.

eBPF applications: For a long time, eBPF was only used for packet filtering [49], monitoring [3, 63], and load balancing [14] because of its restricted programming model. Now, it is shown to be able to offload small yet critical operations to improve application performance. CCP [53] mentions that it may be possible to leverage the JIT feature of eBPF to gather datapath's congestion measurements for congestion control. BMC [17] uses eBPF to implement an in-kernel cache to accelerate UDP-based Memcached GET requests and achieves significant throughput improvement. Syrup [26] uses eBPF maps to share incoming request information across OS, networking stacks, and application runtimes to enable user-defined scheduling. SPRIGHT [65] employs fast eBPF-based packet forwarding to accelerate sidecar proxies in serverless computing. XRP [82] offloads storage functions (e.g., B-tree lookups) into the kernel using eBPF to reduce kernel storage stack overhead. SynCord [58] leverages eBPF to inject workload-specific and hardware-aware kernel lock policies specified by application developers. Electrode further demonstrates that eBPF can be used to accelerate distributed protocols under the kernel networking stack.

10 Conclusion

Electrode is a system that accelerates distributed protocols using safe in-kernel eBPF-based packet processing before the networking stack. Electrode retains the benefits of using the standard Linux networking stack (e.g., good maintenance, elastic CPU scaling, security, and isolation), while optimizing the performance-critical operations of distributed protocols (e.g., broadcasting, and wait-on-quorums) in a non-intrusive manner. When applying Electrode to a classic Multi-Paxos protocol, we achieve up to 128.4% higher throughput and 41.7% lower latency. We believe that the designs of eBPF-based optimizations in Electrode can motivate more research on improving networked application performance while maintaining the standard Linux networking stack.

Electrode code is available at `https://github.com/Electrode-NSDI23/Electrode`.

Acknowledgments

We thank our shepherd Adam Belay and the anonymous reviewers for their insightful comments. We thank Cloudlab [12] for providing us with the development and evaluation infrastructure. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou is also supported by the Google PhD Fellowship.

References

[1] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[2] NVIDIA Corporation affiliates. Single Root IO Virtualization (SR-IOV) for Mellanox NICs. https://docs.nvidia.com/networking/pages/viewpage.action?pageId=43718746.

[3] The Cilium Authors. Cilium: eBPF-Based Networking, Observability, Security. https://cilium.io/.

[4] Catalonia-Spain Barcelona. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of USENIX OSDI*, 2008.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[6] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of USENIX OSDI*, pages 335–350, 2006.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[8] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[9] Intel Corporation. Single Root IO Virtualization (SR-IOV) for Intel NICs. https://www.intel.com/content/www/us/en/support/articles/000005722/ethernet-products.html.

[10] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at Network Speed. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 1–7, 2015.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[13] Facebook. Facebook's Branch of Apache Thrift, Including a New C++ Server. https://github.com/facebook/fbthrift/blob/main/thrift/doc/cpp/cpp2.md#options.

[14] Facebook. Katran: A High-Performance Layer 4 Load Balancer. https://github.com/facebookincubator/katran.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of ACM SOSP*, pages 29–43, 2003.

[17] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[18] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of USENIX NSDI*, pages 1249–1266, 2022.

[19] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[20] The Tcpdump Group. tcpdump. https://www.tcpdump.org/.

[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David

Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of ACM CoNEXT*, pages 54–66, 2018.

[22] Michael Isard. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[23] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX NSDI*, pages 425–438, 2016.

[24] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of USENIX NSDI*, pages 35–49, 2018.

[26] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[29] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[30] The Linux kernel development community. BPF Ring Buffer. https://www.kernel.org/doc/html/latest/bpf/ringbuf.html.

[31] The Linux kernel development community. struct sk_buff. https://docs.kernel.org/networking/skbuff.html.

[32] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *Proceedings of EuroSys*, pages 1–17, 2020.

[33] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of USENIX ATC*, pages 863–880, 2019.

[34] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[35] UW Systems Lab. Speculative Paxos Open Source. https://github.com/UWSysLab/specpaxos.

[36] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[37] Leslie Lamport. The Part-Time Parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[38] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of ACM PODC*, pages 312–313, 2009.

[39] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of ACM SOSP*, pages 104–120, 2017.

[40] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of USENIX OSDI*, pages 467–483, 2016.

[41] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of ACM SoCC*, pages 1–14, 2014.

[42] John C Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings of IEEE INFOCOM*, volume 96. Citeseer, 1996.

[43] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. 2012.

[44] Xuhao Luo, Weihai Shen, Shuai Mu, and Tianyin Xu. DepFast: Orchestrating Code of Quorum Systems. In *Proceedings of USENIX ATC*, pages 557–574, 2022.

[45] Linux Programmer's Manual. bpf-helpers(7). https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[46] Linux Programmer's Manual. bpf(2). https://man7.org/linux/man-pages/man2/bpf.2.html.

[47] Linux Programmer's Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[48] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[49] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[50] Paul E McKenney. Overview of Linux-Kernel Reference Counting. *N2167*, pages 07–0027, 2007.

[51] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Turquois: Byzantine Consensus in Wireless Ad hoc Networks. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 537–546. IEEE, 2010.

[52] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of ACM SOSP*, pages 358–372, 2013.

[53] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[54] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of ACM PODC*, pages 8–17, 1988.

[55] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *Proceedings of USENIX ATC, FREENIX Track*, pages 183–191, 1999.

[56] VMware Inc. or its affiliates. Spring Data Redis - Retwis-J. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[57] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[58] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

[59] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[60] Valentin Poirot, Beshr Al Nahas, and Olaf Landsiedel. Paxos Made Wireless: Consensus in the Air. In *EWSN*, pages 1–12, 2019.

[61] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI*, pages 43–57, 2015.

[62] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of Interrupt Coalescence on Network Measurements. In *International Workshop on Passive and Active Network Measurement*, pages 247–256. Springer, 2004.

[63] The IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[64] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[65] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[66] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM SOSP*, pages 740–755, 2021.

[67] ScyllaDB. SeaStar High Performance Server-Side Application Framework. https://github.com/scylladb/seastar.

[68] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source Routed Multicast for Public Clouds. In *Proceedings of ACM SIGCOMM*, pages 458–471. 2019.

[69] Gráinne Sheerin. gRPC and Deadlines. https://grpc.io/blog/deadlines/.

[70] Alberto Spina, Julie McCann, Michael Breza, and Anandha Gopalan. *Reliable Distributed Consensus for Low-Power Multi-Hop Networks*. PhD thesis, Master's thesis, Imperial College London, 2019.

[71] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland.

The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of VLDB*, page 1150–1160. VLDB Endowment, 2007.

[72] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of EuroSys*, pages 1–14, 2020.

[73] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing Storage Performance with Calibrated Interrupts. *ACM Transactions on Storage (TOS)*, 18(1):1–32, 2022.

[74] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of USENIX OSDI*, volume 4, 2004.

[75] Ed. W. Eddy. RFC 9293: Transmission Control Protocol (TCP). https://datatracker.ietf.org/doc/html/rfc9293.

[76] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[77] IJsbrand Wijnands, E Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. RFC 8279: Multicast Using Bit Index Explicit Replication (BIER). https://www.rfc-editor.org/rfc/rfc8279.

[78] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of ACM SIGCOMM*, pages 126–138, 2020.

[79] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[80] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[81] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *Proceedings of ACM SoCC*, pages 68–81, 2018.

[82] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

Types	Protocols	Applying message broadcasting	Applying fast acknowledging	Applying wait-on-quorums
Replication	Primary-backup	The primary broadcasts requests to backups.	Each backup buffers messages in the kernel and quickly responds to the primary.	The primary waits for responses from all backups.
	Chain	None	Each replica (except for the last one) buffers write requests in the kernel and forwards them to the next replica.	None
Concurrency control	Two-phase locking	A transaction coordinator broadcasts `LOCK` and `UNLOCK` requests to all shards.	Each shard maintains a lock table in the kernel and directly handles lock acquiring and releasing.	A transaction coordinator waits for responses from all shards.
	OCC	None	Each shard checks in the kernel if the committing transaction's timestamp conflicts with all other running ones.	None
Atomic commitment	Two-phase commit	A transaction coordinator broadcasts `PREPARE` and `COMMIT` requests to all shards.	Each shard buffers `PREPARE` messages in the kernel and responds to the coordinator, and handles `COMMIT` requests by polling the buffered messages.	A transaction coordinator waits for responses from all shards

Table 4: Applying Electrode to more distributed protocols.

APPENDIX

A Electrode Generalizability

Table 4 summarizes how the classic replication, concurrency control, and atomic commitment protocols can leverage Electrode optimizations. For example, the primary-back replication, two-phase locking, and two-phase commit protocols follow the pattern of sending requests to multiple nodes and waiting for a quorum number of responses; thus they naturally fit well with the eBPF-based message broadcasting and wait-on-quorums. Together with the above protocols, the chain replication [74] and opportunistic concurrency control (OCC) [34] protocols include some critical-yet-simple operations like storing messages in memory, maintaining a lock table, and checking timestamp conflicts; these operations are also suitable for offloading to the eBPF following the fast acknowledging mechanism.

B Impact of Interrupt Coalescing

During benchmarking, we noticed that the interrupt coalescing [62] (IC) feature of modern NICs has a big impact on the measured performance. In IC, after an incoming packet triggers an interrupt, the kernel networking stack waits until a threshold of packets arrives or a timeout gets triggered, aiming to amortize the interrupt cost. In our scenarios, we find it significantly hurts latency and performance predictability in our settings; similar results are also reported in [73]. Thus, in all our experiments, we disable IC by default.

Figure 9 shows the performance impact of IC on the Multi-Paxos protocol and Electrode-accelerated one, by varying the number of open-loop clients. With IC, load-latency curves become unpredictable with two "hockey stick"s. The second "hockey stick" is because the extremely high load triggers coalescing/batching much more packets in one interrupt. Overall, IC does not nearly impact the maximum throughput for the Multi-Paxos protocol and Electrode-accelerated one, but it increases the latency by 57.4%-129.2% and 9.1%-246.8% with 1-3 clients (before the first "hockey stick"). Moreover,

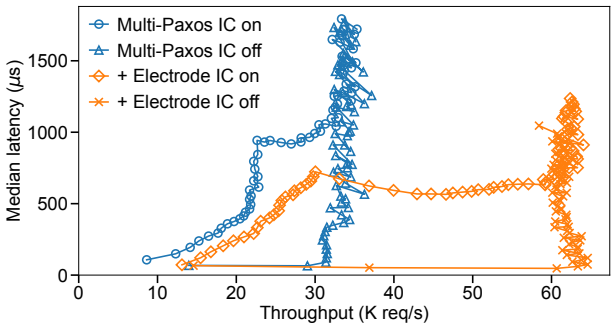


Figure 9: Performance impact of interrupt coalescing (IC) on the Multi-Paxos protocol vs. Electrode-accelerated one (with 5 replicas).

enabling IC decreases the one-client throughput by 38.3% and 10.1% for the original Multi-Paxos and Electrode-accelerated one, respectively.

Electrode performance with IC: Electrode accelerates the maximum throughput of the Multi-Paxos protocol by 81.4% and latency by 32.7% with 1 client (before the first "hockey stick") when IC is on.

Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes

Zhenyuan Ruan Seo Jin Park Marcos K. Aguilera [‡] Adam Belay Malte Schwarzkopf [†]
MIT CSAIL *[‡]VMware Research* *[†]Brown University*

Abstract. Datacenters waste significant compute and memory resources today because they lack resource *fungibility*: the ability to reassign resources quickly and without disruption. We propose *logical processes*, a new abstraction that splits the classic UNIX process into units of state called *proclets*. Proclets can be migrated quickly within datacenter racks, to provide fungibility and adapt to the memory and compute resource needs of the moment. We prototype logical processes in Nu, and use it to build three different applications: a social network application, a MapReduce system, and a scalable key-value store. We evaluate Nu with 32 servers. Our evaluation shows that Nu achieves high efficiency and fungibility: it migrates proclets in ≈100μs; under intense resource pressure, migration causes small disruptions to tail latency—the 99.9th percentile remains below or around 1ms—for a duration of 0.54–2.1s, or a modest disruption to throughput (<6%) for a duration of 24–37ms, depending on the application.

1 Introduction

Compute and memory are valuable and expensive resources in datacenters today, but they are inefficiently utilized [46, 76]. A key reason for this inefficiency is a lack of *fungibility*—the ability to reassign resources quickly and without disruption between different users and across different machines. Without fungibility, resources are stranded and over-provisioned for fear of running short, even as resource consumption naturally fluctuates in datacenter applications [2, 7, 18, 34, 39].

Existing systems fail to provide fungibility because current abstractions for compute work and memory state (VMs, containers, processes) are too coarse-grained (§2). To address this problem, we introduce the abstraction of a *logical process*. Logical processes provide fungibility, while retaining a familiar programming model similar to traditional processes. A logical process consists of many smaller *proclets*, atomic units of state and compute that can be independently migrated under resource pressure to achieve fungibility. Like a traditional process, a logical process has its own address space, isolated from other processes. But unlike a traditional process, a logical process can spread across many machines in datacenter racks as a result of the migration of its proclets. Intuitively, logical processes break down the monolithic nature of traditional processes into many proclets. A proclet consists of a heap (state) and a set of user-level threads and their execution contexts (stacks and register values). A runtime system that manages the logical process responds to spikes in load by migrating proclets quickly to a machine with spare resources.

To realize logical processes and proclets, we had to address three challenges. First, proclet migration must be fast and react to resource pressure before resources are exhausted. Second, communication between proclets and migration of proclets must impose little overhead or disruption on the application, especially if migration itself consumes resources when they are short. Third, the programming model of logical processes and proclets must support practical datacenter applications.

We respond to these challenges as follows. First, we divide process state into proclets, which are small relative to an entire process, so they can be migrated orders of magnitude faster than VMs or processes. Second, we optimize our software stack to take full advantage of modern datacenter networks (at 100–400 Gbit/s). This pushes performance far enough for proclets to migrate in ≈100μs. We also scale proclets across machines with minimal communication overheads by using a single program image across machines and an optimized RPC stack. Third, we use a global address space to provide a programming model that is process-like and intuitive. This makes it possible to statically check types, and enables computation shipping by passing function pointers between proclets.

We prototyped logical processes and proclets in Nu, a system that provides a C++ class API and a Caladan-based user-level threading and kernel-bypass networking runtime [28]. Nu targets environments with tens of racks: hundreds of machines connected with an overprovisioned network that provides high full-bisection bandwidth (100–400 Gbit/s) and low latency (10–20 μs). We implemented three applications using Nu. The first is a version of the DeathStarBench social network application[29], originally implemented using microservices. The Nu version of this application is simpler, shorter, and has an order of magnitude better performance than the microservice version, while preserving scalability. The second application is k-means clustering on Phoenix MapReduce [63], which represents a compute-intensive workload with high parallelism. Phoenix MR originally supported thread parallelism in a single NUMA machine, but the Nu version scales across multiple machines while also delivering comparable single-machine performance. The third application is a scalable key-value store implemented in Nu as a hash table whose buckets are distributed across multiple proclets.

We evaluate Nu in a setup of 32 servers with 100 GbE NICs that are connected through a top-of-rack switch. Our evaluation shows that Nu achieves high efficiency and fungibility: it reacts quickly to resource pressure and migrates

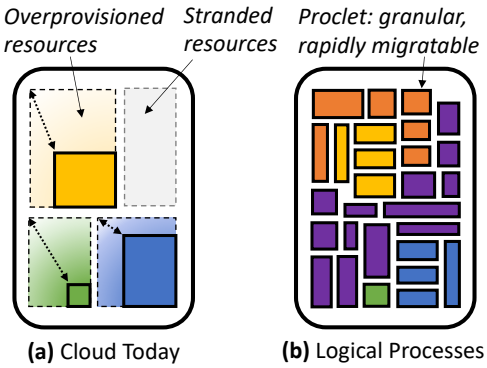


Figure 1: (a) Resources are wasted as they are overcommitted for peak use (gradient) or stranded as additional tasks do not fit (gray). (b) Proclets permit tighter packing, which fits more tasks (orange, purple) that can be migrated away quickly under resource pressure.

proclets without disruption to the application workload. Nu migrates proclets in ≈100μs and its migration exceeds the rate at which the Linux kernel can allocate memory (≈ 7GB/s), so Nu handles even intense resource pressure. Under this memory pressure, the social network app adds 122μs to the 99.9th-percentile client latency for a period of 0.86s; the key-value store app adds 52μs for 2.1s; and k-means loses 2.9% throughput for 37ms. Under intense compute pressure, disruption is higher, but still short-lived: the key-value store adds 1,053μs for 0.54s; and k-means loses 5.8% throughput for 24ms. Finally, Nu's logical processes are efficient in the absence of resource pressure, and match or exceed the performance of strong baselines on one or more servers.

Nu has some limitations. First, logical processes require developers to structure applications as proclets. This may not be feasible for every application (§3.3), but we have shown that it is feasible for three very different applications. Second, Nu currently considers only two resources, memory capacity and compute load. We expect other resources can be added (network, caches, memory bandwidth, etc.), but that remains future work. Third, Nu targets deployments where network bandwidth is plentiful and latencies low.

Nu is available as open-source software [66].

2 Motivation: Resource Fungibility

Cloud computing originally promised to deliver utility computing, with fine-grained, pay-per-use sharing of compute resources, rather than fixed-size machines that customers must purchase and own [4, 31]. But, almost two decades later, the operational reality is different: although end-users can readily rent resources, cloud providers still provision and offer these resources in fixed-size units and over long time horizons.

We argue that a key problem in this setting is the lack of fungibility—the ability to reassign resources quickly and without disruption between different users and across different machines. Users today submit requests for fixed allocations (number of cores, memory, etc.) as determined by so-called "instances" (or "slots", "tasks"). These allocations tend to overestimate actual resource use, which fluctuates at sub-second time scales. Providers bin-pack instances onto the available servers [33, 35, 71, 76, 77]. This is inefficient because users must size instances for peak rather than typical usage, leaving substantial resources idle most of the time. Providers can reclaim some of these wasted resources by overbooking and scheduling best-effort instances in them [2, 44, 76, 84]. But this practice is disruptive, as machines can get intermittently overloaded, leading to performance degradation (e.g., high tail latencies), which is particularly problematic for latency-sensitive workloads [28, 44]. In response, the cluster manager must kill some best-effort instances to free up resources. But doing so is also disruptive because the work done by a killed instance can be wasted and may need to be redone. Moving the instance usually is not an option as it requires an expensive VM or process migration that can take seconds or minutes because the state to be moved is large, and it requires the cluster manager to find a destination machine that has sufficient resources to take over the entire (indivisible) instance.

In other words, today's cloud is not fungible (Figure 1(a)). Resources can only be reassigned on fairly long timescales, larger than the timescales over which resource consumption fluctuates. The underlying reason for this problem is that current abstractions for compute work and memory state—VMs, containers, and processes—are too coarse-grained.

A more efficient design would avoid disruption and reassign resources quickly and at fine granularity. This would make it easy for providers to increase utilization by densely packing instances across machines while rebalancing and migrating work as necessary, instead of killing instances under resource pressure. In addition, this would eliminate the burden on users to predict and specify peak per-machine resource usage for each instance, allowing them to instead pay for resources as they are used.

Our approach to fungibility. To provide fungibility, we revisit the *process*, a core OS abstraction that dates back to the 1960s. Traditionally, a process is an instance of a computer program that runs on one machine, consisting of memory and a set of threads. Our work extends this idea across machines to provide a similar abstraction called a *logical process*.

Logical processes are inspired by logical disks [59, 74]. Much like a logical disk, a logical process combines together disparate physical resources—in this case, machines rather than disks. A logical process automatically scales to use additional machines when more capacity is needed, and can recover from machine failures.

A logical process achieves fungibility through two key ideas (Figure 1(b)). First, a logical process divides program state into fine-grained partitions called *proclets*. Second, proclets are migrated quickly between machines in response to memory or compute resource pressure. Each proclet runs on one machine at a time, and proclets communicate with each other through efficient message passing.

Because proclets are fine-grained, migrations complete

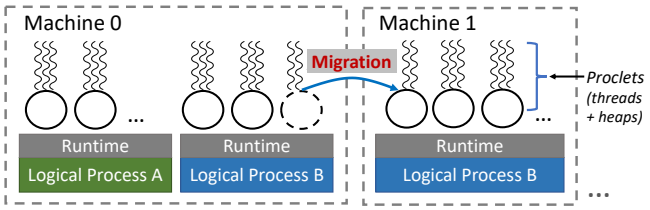


Figure 2: Logical processes spread across machines. Each logical process is comprised of proclets that include a heap (shown as a circle) and threads (shown as squiggles). Proclets rapidly migrate to other machines in response to resource pressure.

quickly, causing minimal performance disruption. Inspired by prior work that shows that decomposition into small units simplifies placement [51, 57], proclets' fine granularity makes packing them onto machines simple and avoids complex and time-consuming bin-packing on allocation or migration.

Alternative approaches. There are a few other approaches to improving fungibility, but they have drawbacks. One can migrate VMs [20, 78], containers [22], or processes [49], but migration is slow due to their state size. An alternative that maintains the process abstraction is to use distributed shared memory (DSM) to spread a normal process across machines [83]. But DSM systems experience high coherence overheads with shared memory, leading to poor performance. PGAS [5, 17, 54] is a type of DSM that can avoid such overheads, but its applicability is limited to parallel applications.

Another approach to fungibility is to adopt new programming models to distribute the application into smaller units, as with distributed objects [6, 13, 21, 30, 70, 82], microservices, and serverless functions. These models depart significantly from the familiar process abstraction, and they are built on top of traditional, coarse-grained instances that limit their fungibility. They also have high RPC messaging overheads (and cold start delays for serverless functions [72]) that grow in cost as their units become smaller. Alternatively, parallel programming frameworks [8, 15, 23, 26, 50, 81] partition work via rigid compute patterns (e.g., partition-aggregate, actors). This constrains the programming model and requires data to be statically placed on machines.

Finally, some techniques provide fungibility, but in limited form. Far memory systems [32, 67, 79] can incrementally extend the memory of a process, but they perform well only when the remote memory is cold. Request load balancing can make compute fungible, but it is mostly suited for stateless or read-only services. These two techniques are complementary to logical processes and can be combined with them.

3 The Logical Process Abstraction

A *logical process* exists across one or several machines and contains a collection of proclets. Proclets are fine-grained partitions of program state that form units of migration. Proclets can be individually migrated between machines to relieve resource pressure (Figure 2).

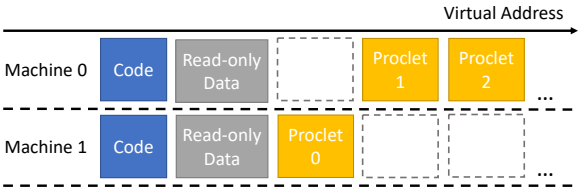


Figure 3: The address space layout of a logical process running on two machines. Read-only code and data is mapped everywhere, while proclets are mapped in exactly one machine at a time.

A proclet consists of a heap and a set of threads that can access the heap concurrently via shared memory. A proclet never shares its heap memory directly with other proclets. Instead, each proclet has an associated *root object*, which defines a remote method interface that other proclets use to access its state. This approach allows developers to build full programs from proclets in a natural, object-oriented way. The root object may store references (pointers) to ordinary local objects stored on the proclet's heap.

The number of machines allocated to a logical process can change over time in response to shifts in the resources available on each machine. Each machine handling logical processes runs a separate runtime instance. The runtime provides location-transparent communication between proclets, detects resource pressure, migrates proclets between machines, and cleanly handles failures.

Developing software for logical processes is similar to normal UNIX processes. Code can spawn threads, use synchronization primitives to coordinate access to shared memory, and allocate memory using standard APIs like `malloc` or `new`. But there are two major differences. First, developers must partition their program state into proclets. Second, in most cases, developers must use runtime APIs instead of making system calls or performing I/O directly. We describe the logical process abstraction in more detail in the following.

3.1 Address Spaces and Cache Coherence

A logical process uses an identical address space layout on each machine. This simplifies migration, as pointers remain valid across machines without swizzling. Runtime instances coordinate to keep their layout synchronized during initialization and whenever new proclets are created.

Figure 3 shows an example address space layout for a logical process running on two machines. Code and shared data segments are mapped read-only on all machines. Consequently, the machines must be binary-compatible, but not necessarily identical architectures (e.g., AMD and Intel x86 CPUs). Read-only data can store large static arrays, tables, and other inputs that all proclets might need. Proclets' heaps, on the other hand, are readable and writable, only mapped on one machine at a time, and only ever accessible by the owning proclet. (This contrasts with distributed shared memory [3, 10, 25, 40, 52, 68, 69], which typically provides cache coherence across machines.) In other words, *no proclet can share*

```
1  struct Accumulator {
2    Accumulator(int val) : val_(val) {}
3    void Add(int n) { std::scoped_lock l(mu_); val_ += n; }
4    int Get() { std::scoped_lock l(mu_); return val_; }
5    std::mutex mu_; int val_;
6  };
7
8  void mainfunc() {
9    // Creates two proclets with root class Accumulator.
10   auto p1 = make_proclet<Accumulator>(10);
11   auto p2 = make_proclet<Accumulator>(10);
12   // Invokes Get() on p1; prints 10.
13   std::cout << p1.Run(&Accumulator::Get);
14   // Invokes a closure on p1; prints 15.
15   std::cout << p1.Run(+[](Accumulator &a) { a.Add(5);
         return a.Get(); });
16   // Invokes Get() asynchronously; prints 25.
17   auto f1 = p1.RunAsync(&Accumulator::Get);
18   auto f2 = p2.RunAsync(&Accumulator::Get);
19   std::cout << f1.get() + f2.get();
20   // Adds p2's value to p1 by invoking a closure on p1.
21   p1.Run(+[](Accumulator &a, proclet<Accumulator> p) {
22     auto v = p.Run(&Accumulator::Get); a.Add(v); }, p2);
23   // Arguments statically type checked; DOESN'T COMPILE!
24   // p1.Run(&Accumulator::Add, 10, 20);
25   // Proclets are freed when mainfunc() gets out of scope
26 }
```

Figure 4: Code sample for logical processes. Proclets have a root class with methods containing the app logic. Proclets can create other proclets, run methods synchronously or asynchronously, and run closures. Closures can take proclets as arguments to chain execution.

memory with another proclet. Instead, proclets communicate via remote method invocation, which passes arguments by copying if the proclets are co-located on a machine or by network transfer if they are on different machines.

This design avoids writable shared memory across machines and aligns well with current datacenter networks, which provide high throughput and low latency, but lack hardware support for cache-coherent memory across machines. Additionally, this design enables fault isolation, as it allows one proclet to fail independently from others on different machines. A failure can cause a proclet's memory to disappear at any time, and these errors can be cleanly reported via return codes of remote methods. This allows us to use standard distributed systems techniques (e.g., replication) to make critical proclets fault-tolerant.

Proclet migrations occur atomically and each proclet runs on exactly one machine at a time. Consequently, cache coherence is available within proclets, but not across proclets. This design allows for a normal programming environment inside proclets, including synchronization across threads (via spinlocks, mutexes, etc.) when they access shared memory within a single proclet's heap.

3.2 Programming Model

Developers write an application as a set of proclet root classes. As in traditional object-oriented programming, each class defines methods and fields. Methods implement the proclet's application logic and expose the API for the proclet to be invoked by other proclets. Fields specify state internal to the proclet, although additional state can be allocated dynamically in the heap at runtime. Figure 4 shows a running example in C++.[1] Lines 1–6 define `Accumulator` as the root class for a simple proclet that keeps a value `val_` and exposes two methods `Add` and `Get` to increment and retrieve the value. Here, the methods are one-liners, but in real applications they constitute most of the code.

When a logical process starts up, the runtime launches a `main` proclet. This proclet typically creates other proclets by calling function `make_proclet` with their root classes and constructor parameters. In the example, the `main` proclet invokes function `mainfunc` (for brevity we do not show the `main` proclet, only `mainfunc`), which in lines 10–11 creates two proclets with root class `Accumulator`.

Proclets communicate only via remote method invocations and closures. With remote method invocations, a proclet calls the methods of the root object of another proclet, either synchronously using function `Run()`, or asynchronously using function `RunAsync()`, which returns a future. Lines 13 and 17–18 show a synchronous and two asynchronous invocations of `Get` on proclets. The two asynchronous invocations run concurrently to hide latency.

With closures, a proclet can implement function shipping [36, 38, 65, 67, 79, 80], and ship a function that invokes methods—interspersed with its own processing logic—on the root object of another proclet. Line 15 shows a closure that invokes `Add` and `Get` on the same proclet. This execution incurs a single roundtrip to the server hosting the proclet, even though it invokes two methods. Shipping code to data in this manner can greatly improve efficiency.

The remote runtime may execute methods and closures on the same proclet concurrently on different threads. Hence, the example uses a mutex `mu_` to protect `val_` against concurrent execution of `Add` and `Get`.

Naming and reference counting. Proclets need to know about each other before they can communicate. We adopt a proclet naming scheme based on smart *proclet pointers*. These pointers provide safety, convenience, and reference counting through a interface similar to C++'s `shared_ptr`.

Unlike standard RPC frameworks, remote methods or closures can take proclet pointers as arguments. Thus, code can pass handles to proclets to other proclets by passing them as parameters, similar to delegating capabilities. This feature permits a remote method or closure to chain together the execution of multiple proclets while performing computation in between. For example, line 22 shows a closure on proclet `p1` that takes proclet `p2` as a parameter; the closure first calls `p2`'s `Get` method, followed by `p1`'s `Add` method.

Proclet pointers are valid within the entire logical process, even across machines, and the runtime frees a proclet when it loses its last reference. In the example, proclets `p1` and `p2` are freed automatically when `mainfunc()` goes out of scope.

We considered using global strings as proclet names, but never needed them in building applications. A logical process

[1] A logical process can be implemented in other languages too.

is tightly coupled, and we found that passing smart pointers is more convenient than hard-coding strings. Typically, initialization code creates several proclets, and passes around their smart pointers, so the code hands over access directly.

Type checking. Because a logical process uses an identical program image across machines, static type checking of argument types is sufficient for remote invocations. This contrasts with standard RPC frameworks (e.g., Thrift or gRPC), which additionally have to perform dynamic type checking, incurring runtime overheads and requiring extra error handling. Line 24 thus fails to compile because of too many arguments.

Raw pointers into a proclet's heap are never allowed as arguments; we made this choice to discourage incorrect code that attempts to share memory between proclets. On the other hand, smart pointers are supported, and passing them as arguments causes the objects they own to be copied.

Unlike standard RPC frameworks, proclet invocations allows remote methods and closures to take function pointers and closures as arguments. This is possible as all machines in the logical process map the code segment at the same address.

Network I/O outside a logical process. Logical processes perform their I/O through abstractions provided by the runtime, rather than POSIX syscalls and I/O abstractions. This allows proclets to be machine-independent and migrate between machines without having to move hard-to-migrate local kernel state (e.g., the TCP state machine). In particular, the runtime maintains TCP network connections to clients, which can be either other logical processes or normal processes. These connections allow clients to communicate with specific proclets inside a logical process—or to spread load across groups of stateless proclets—and will forward client requests if the destination proclet has migrated. Similar to existing libraries for distributed request routing [1, 53], the runtime informs client libraries about the proclet's new location, so that the client knows to expect the response on another network connection and to send future requests there. In our datacenter setting, client and server code are under the control of the same entity, and custom I/O libraries (e.g., for request routing and load balancing) are commonplace [1, 11, 73].

In rare cases, developers can pin proclets that need to use local resources directly to a machine. Such proclets lose their ability to migrate and reduce resource fungibility, so developers should pin proclets only if absolutely necessary.

3.3 Porting Applications to Logical Processes

In principle, any application that can partition its state into fine-grained units can be ported to a logical process (each unit becomes a proclet). This aligns well with existing cloud applications that already partition their state (e.g., microservices, FaaS, distributed frameworks, etc.), though sometimes at a coarser granularity than proclets. There are two main considerations when dividing a logical process into proclets: the proclet granularity and its scope.

Proclet granularity. Choosing the right size for proclets is important. If proclets are too large, resource fungibility suffers. If they are too small, communication overheads increase as remote invocations become more frequent. Developers must choose a sweet spot that provides sufficient fungibility without significant overheads. §6.4.2 shows empirical measurements of proclet performance at different state sizes and invocation compute intensities; in practice, proclets of a few MiB state size work well.

Proclet scope. The next consideration is how to decide what functionality goes into a proclet. One approach is *functional splitting*, which equates a proclet to a logical functional unit in the application (a module, a microservice, a package, etc). Well-known software engineering practices suggest how to choose appropriate units [47, 56]: the unit should include functionality that is intuitively related, that can be described simply, and that can be encapsulated through a compact and easy-to-understand API. The latter property ensures that the interface between proclets is also compact. Another approach is to use *sharding*. Since a functional unit may be much larger than the ideal proclet size, it may help to shard (partition) the unit. For example, consider a large chaining hash table. Each hash bucket of this data structure becomes a separate proclet and stores the proclet pointer in the hash array. To operate on a key in the hash table, the code makes the appropriate method invocation to the corresponding proclet. This results in a distributed key-value store, as proclets are spread across machines, but maintains the hashtable API.

Limitations. Some applications are hard to decompose into proclets, such as applications that manipulate large amounts of state that is not easily divisible (e.g., video encoders, architecture simulation, sorting, or graph processing). For these examples, decomposition may still be possible, but it requires new algorithmic approaches [27, 41, 45, 48, 64].

Other applications may require functionality that is tied to physical hardware resources, such as a GPU or an FPGA. In these cases, proclets that interact with the hardware may need to be pinned, thus reducing the logical process's fungibility.

3.4 Security and Threat Model

A logical process has the same isolation properties as a UNIX process—viz., its memory is isolated from other processes, but its threads share an address space—but applies this model across multiple machines. Even though proclets lack shared memory, there is no hardware memory isolation (e.g., via the MMU) between the proclets *within* a logical process to enforce this. We made this choice for performance reasons and because it matches the isolation model of UNIX processes. On the other hand, memory isolation is guaranteed across *different* logical processes: each local logical process instance runs in a different UNIX process and is isolated from other logical process instances on the machine.

Address space layout randomization (ASLR) and stack ca-

naries are important defenses against buffer overflow attacks. Although ASLR might at first glance seems incompatible with logical processes' global address space, it works as long as the loader maintains the same randomized address space layout on each machine. Stack canaries also work, as proclets cannot share stack memory and the implementation can maintain a different secret canary value for each proclet.

Finally, logical processes trust the network to provide confidentiality and integrity. This is necessary to make remote method invocation and migration efficient by sending raw data and pointers. Modern datacenter NICs have hardware encryption engines that ensure these properties.

3.5 Fault Tolerance

A proclet may be replicated to tolerate failures. Replication creates backup copies of the proclet's heap, which the runtime places at the same virtual address in different machines. To keep the backup heaps in sync, the primary replica serializes the invocation requests and forwards them to the backup replicas. (This requires proclet operations to be deterministic.) Operations on a replica are totally ordered without overlap within each proclet—a choice that trades off some performance for strong consistency. To reduce replication latency, the primary overlaps execution with the backups, but the primary only returns from an invocation once the backups finish.

When the system detects the failure of the primary (e.g., due to an RPC time out), it atomically promotes a backup to the primary. To keep the same replication factor, it also adds a new backup by pausing the proclet and copies its heap from the new primary to the new backup replica.

4 The Nu Runtime System

We built Nu, a prototype runtime that provides the logical process abstraction and runs inside a normal Linux environment. Nu shares some architectural and implementation building blocks with Caladan [28]. Caladan was a good fit for Nu because it provides a user-level threading package with overheads low enough to hide microsecond-scale latency. For example, if a thread blocks waiting for a remote proclet invocation to return, the runtime can quickly context switch to another runnable thread with little overhead. Caladan uses work-stealing to balance these threads across cores, which reduces tail latency [62]. Caladan also provides an optimized kernel-bypass, user-level TCP/IP networking stack to further reduce proclet communication and migration costs.

Nu adds ≈10,000 lines of C++ code to Caladan. This includes efficient communication infrastructure, a new memory management layer to handle multiple heaps, a well-optimized proclet migration system, and a controller to track the location of proclets. In the following, we describe these components.

4.1 Serialization and Communication

Nu serializes arguments to remote invocations using `cereal`, an efficient, header-only library for serialization [16]. Cereal has a compact binary serialization format that supports most STL types, but prohibits raw pointers and references (`shared_ptr` and `unique_ptr` are still supported). We modified `cereal` so that it can serialize function and proclet pointers. To optimize use of `cereal`, Nu maintains a buffer pool for serialized outputs and eliminates extra data copies.

Nu uses C++ templates to internally produce code at compile time for serialization and deserialization of remote method arguments. This contrasts with RPC frameworks like Thrift, which require code generation and an interface description language. As a result, developers call remote methods without boilerplate, and they benefit from static type checking.

We took several steps to optimize remote method invocations. First, Nu opens one TCP connection on each core for each outgoing machine. These connections use specific 5-tuples, so they have flow-level affinity matched with the core they are associated with, enabling cache-aware steering [42, 60]. This design increases the number of open connections, but Caladan easily scales to 10,000 connections, much more than needed for our target environment. Second, Nu applies adaptive batching to combine remote method invocation payloads (requests and responses) into larger TCP transfers without impacting latency [9]. We modified Caladan to use jumbo frames to increase the benefit of this batching. Third, each connection operates as a closed queuing system, limiting the maximum number of requests in flight. This provides flow control and prevents unbounded memory consumption under overload. Finally, when the caller and callee proclets are in the same machine, Nu substitutes the RPC with a fastpath: a local call without any RPC overheads.

4.2 Memory Management

Nu uses a custom slab allocator to manage each proclet's heap. It includes a per-core object cache to increase scalability, similar to most modern multicore memory allocators [12, 14]. C++ allows a custom definition of `operator new()` that Nu uses to override memory allocations. Nu keeps track of which proclet each thread is associated with and directs allocations to the correct heap. In the future, we plan to explore specialized proclet allocators too. For example, an arena allocator could benefit short-lived proclets because it need not free objects until the proclet terminates, reducing overheads.

4.3 Migration

Nu migrates proclets across binary-compatible machines under resource pressure. Nu separates migration mechanism from policy.

Mechanism. To migrate a proclet, the runtime first sets a migration flag, causing method invocations to the migrating proclet to be rejected and retried. Next, it preemptively pauses and saves register state for all the proclet's running threads to ensure that the data is not mutated during migration. Then, it moves proclet data, including heap, stack, and register state, to the new destination. Finally, the runtime clears the migration flag and contacts the controller to update the location of the

proclet, ensuring pending and future method invocations are routed to the new destination (§4.4). We co-designed Nu's RPC layer with migration, and it routes the results of method invocations on migrated proclets back to the caller.

We optimized Nu's migration datapath. To improve TCP throughput, we use parallel connections and jumbo frames. We found that Linux's `mmap` (used for creating the proclet space at the destination machine) was a bottleneck, so we modified the Linux kernel to pre-zero freed pages. After this optimization, Nu can migrate at line rate on 100GbE. When we tried 200GbE, `mmap` again became a bottleneck—in this case due to the Linux kernel's physical frame allocation speed. As a workaround, Nu instead uses `mmap` to pre-fault a small pool of memory at the destination server. Then, on migration, Nu performs `mremap` on that memory to reuse prior frame allocations. Future Linux kernel optimizations might avoid the need for this remapping.

The CPU overhead of migration is moderate in our current prototype: it takes three hyperthreads to saturate 100GbE and five hyperthreads to saturate 200GbE.

Policy. Nu provides an extensible migration policy interface that dictates which proclets to move and where to move them under resource pressure. Many sophisticated policies are possible, including policies that react to several types of resource pressure (e.g., CPU load, cache pressure, memory capacity, memory bandwidth, network bandwidth, etc.), and policies that co-locate frequently communicating proclets to improve locality. Currently, our prototype ignores locality and focuses only on CPU load and memory capacity, two resources often subject to pressure, but we plan to extend it in the future.

Because Nu's migration is fast, we found that even the simplest policies work well (§6). In particular, Nu needs no sophisticated algorithms to predict future resource use, but rather simply migrates proclets at the last moment, when resources are nearly exhausted. To determine when migrations are needed, a monitoring thread in the runtime polls resource use. For memory, it monitors the amount of free memory and begins migrating once it falls below a threshold (e.g., 1 GiB). For CPU, it monitors system core utilization and begins migrating when a threshold of cores are busy. A better alternative might track the queueing delay of runnable threads, allowing Nu to distinguish actual overload from cases where all cores are busy but not overloaded [19]. We plan to investigate this in the future.

Nu migrates one proclet at a time until resource pressure is eliminated. To determine which proclet to migrate, Nu uses this formula (where P is the set of proclets on the machine):

$$\arg\max_{p \in P} \left(\frac{RESOURCE_USE(p)}{MIGRATION_TIME(p)} \right)$$

$RESOURCE_USE()$ measures a proclet's use of the resource under pressure, and $MIGRATION_TIME()$ models the migration time of a proclet by considering the size of its heap, as well as the number of threads it must pause and transfer, and the size of thread stacks. This maximizes the pressure alleviation rate and helps Nu optimize for response speed. Nu's runtime collects metrics in real time to estimate this rate.

To determine the migration destination, Nu queries a global cluster controller, which monitors resource use across servers and returns possible destinations (described next).

4.4 Controller

Nu has a controller that makes cluster-wide decisions, such as proclet placement and virtual address allocation, and tracks information, such as proclet location and resource use. Nu assumes that the controller is highly available. Although our prototype controller is centralized, high availability can be achieved through primary-backup replication or simple recovery: the controller keeps only soft state, so it can always restore its state by querying the servers.

Placing proclets. The controller periodically probes servers' available resources. It uses this information to decide where to place a proclet on creation or migration. Currently, it uses a simple policy that spreads proclets evenly across machines.

Allocating virtual address segments. Proclets must use non-overlapping virtual addresses. Therefore, Nu divides the virtual address space into an array of segments. These segments are large enough (4 GiB by default) to leave room for a proclet's heap to grow. The controller keeps lists of allocated and unallocated segments. On proclet allocation, the local runtime contacts the controller to obtain an unallocated segment.

Resolving proclet location. The controller keeps a location map from the starting logical address of each proclet to the IP of the machine hosting the proclet. Each local runtime maintains a cache of the location map that contains the proclets it has recently accessed. This eliminates the need for method invocations to communicate with the controller in the common case, moving the controller off the critical path for the steady-state application traffic. When a proclet migrates, the controller updates the map. This causes caches to become stale, so a local runtime may send a method invocation to the wrong machine. When this happens, the remote machine returns an error. The local machine then handles the error by invalidating its cache entry and contacting the controller to find the new machine location.

4.5 Replication

Nu optionally provides traditional primary-backup replication for proclets. This works by forwarding proclet operations from a primary to backup replicas, akin to traditional state machine replication (SMR). One challenge specific to Nu is that a proclet operation can invoke sub-operations on other proclets. The backup replicas will invoke the same sub-operations as the primary, but side-effect causing invocations must occur only once, and replicas must see the same results as the primary's operations. Nu supports a variant [58] of RIFL [37]'s duplicate detection. Proclets assign a unique

Workload	# proclets	Memory	Compute Intensity [time/invoc.]	Proclet size
SocialNetwork	12 (microservices) + 65,536 (hashtable)	113 GiB	1–100 μs (variable)	31 KiB–8 MiB
In-memory KVS	65,536	138 GiB	1 μs (low)	2 MiB
Phoenix k-means	720 (workers) + 1,024 (hashtable)	8.4 GiB	map: 4.6 ms, reduce: 677 μs	31 KiB–19 MiB

Figure 5: Characteristics of the three case study applications (KVS is an in-memory key-value store, and Phoenix is a MapReduce framework).

ID of the form ⟨proclet id⟩+⟨epoch⟩+⟨sequence number⟩ to each proclet-to-proclet invocation. Primaries forward their sub-operation invocation results to the replicas, and replicas reuse the results (identified by the unique ID). Returning the saved results instead of re-executing sub-operations ensures all replicas have the same heap state. As with an unreplicated system, if a primary crashes in the middle of an operation, its sub-operations are re-executed if the unfinished operation is retried.

When Nu's controller detects a failed primary, it promotes a backup to be the new primary and updates its location map with the new primary. However, runtimes may have the old primary in their caches, which could cause a "split brain" situation if the old primary continues to serve requests. A standard epoch-based approach [43, 55] can help Nu avoid this problem: each reconfiguration increments an epoch counter and backup proclets reject operations with outdated epochs.

4.6 Limitations

Our Nu prototype has some limitations. It requires the use of C++, and though the runtime provides many OS services (timers, external and internal network I/O, synchronization, threads, memory allocation, etc.), it does not yet support all services. Despite these limitations, we ported three very different applications to run on Nu.

5 Application Case Studies

We implemented three applications on Nu, which cover a range of proclet sizes, communication patterns, and compute intensities (Figure 5). All applications use a Nu-enabled hashtable library. The hashtable partitions the key space with a hash function and uses proclets as data shards. A root proclet has a vector of proclet pointers to these shards and shares them with client proclets to allow direct communication.

SocialNetwork (from the DeathStarBench suite [29]) is a multi-tier, interactive web service, originally built as 12 microservices. Its overall complexity is high, with a fan-out communication pattern and many microservices that have low compute intensity, making it sensitive to both tail latency and RPC overheads. We ported SocialNetwork to a logical process, turning each microservice into a proclet. However, we found that its compute intensity was sometimes too low and that it lacked autoscaling support; both limit its overall scalability. Therefore, we also built a version of SocialNetwork that is better structured for a logical process: this version merges SocialNetwork's small, stateless microservices into a single root class, and scales by spawning it as proclets across machines. Both versions have ≈1,000 LOC, compared to 6,843 LOC in the original, which highlights the simplifications afforded by logical processes. Our implementation replaces the external stores used by microservices (Memcached and Redis) with a backend based on our hashtable, and leverages proclet closures to support Redis-like local operations. We also modified our external I/O subsystem to interact with unmodified Thrift-based clients. This is possible because any root proclet can handle any request, as root proclets are stateless.

KV Store is a key-value store composed of the Nu-enabled hashtable library and an additional 200 LOC. It is a stateful application that is latency-sensitive and uses significant memory, making it hard to migrate. On each machine, the Nu runtime's external I/O subsystem receives requests from external clients and steers them to the right proclets. The key-value store has low compute intensity (1μs/invocation), but large proclet state (2 MiB/proclet).

K-means is a workload from Phoenix MapReduce [63]. Phoenix MR is a NUMA-oriented, shared-memory MapReduce framework designed for single-machine operation. We run k-means—an algorithm that requires multiple iterations—in a Nu-based Phoenix MR port, using proclets to scale across machines. We modified Phoenix's task scheduler to replace worker threads with worker proclets, ship closures to the workers, and shuffle data between mappers and reducers via our hashtable (changing 548 out of Phoenix's 3,066 LOC). K-means is compute-intensive (0.7–4.6ms/invoc.), but has smaller proclet state (31 KiB–19 MiB/proclet). Overall, we found it easy to modify Phoenix MR to work in a distributed setting. Our version follows the same partition-aggregate communication pattern that makes distributed MapReduce frameworks sensitive to stragglers in k-means.

6 Evaluation

We evaluate Nu with these three applications, as well as microbenchmarks that measure the impact of specific design decisions. Our evaluation seeks to answer four questions:

1. Can migration in Nu prevent performance disruption during intense resource pressure? (§6.1)
2. How does porting applications to Nu impact their performance? (§6.2)
3. How well does Nu scale with the number of servers? (§6.3)
4. What is the effect of compute intensity, as well as that of our key design decisions, on Nu's performance? (§6.4)

Setup. Except §6.4.2, all other experiments run on a cluster of 32 physical servers in CloudLab [24]. The servers are `c6525-100g` instances (24-core AMD 7402P at 2.80GHz,

# of Machines	Controller	Proclet servers	Clients
SocialNetwork	1	26	5
KV Store	1	15	16
K-means	1	30	1

Figure 6: Allocation of machines for each application.

128 GB RAM, Mellanox ConnectX-5 NIC), connected by a 100 GbE network. We run §6.4.2 on `c6525-100g` servers, `c6525-25g` servers (i.e., the variant with 25GbE NICs), and our local servers with a 200 GbE network.

Servers run Ubuntu Linux 20.04 with kernel v5.10 patched to pre-zero free pages (§4.3). We disable ASLR, as Nu does not support it yet.

6.1 Application Performance under Resource Pressure

Nu's proclet-centric design enables fine-grained, rapid migration. The key goal of this design is to achieve high application performance even under resource pressure. To evaluate this, we expose Nu and our applications (§5) to compute and memory resource pressure and measure the application performance as proclets migrate to other machines. We skip SocialNetwork for compute pressure, as this application can handle it with a standard front-end load balancer. We run experiments using 32 machines, one of which serves as the controller (§4.4). The remaining machines are either proclet servers or clients, and we partition them appropriately for the application (Figure 6). To evaluate Nu's ability to manage disruptions under demanding load conditions, we generate enough client load to use ≈70% of CPU capacity across all proclet servers. Then, we induce resource pressure on one proclet server, causing it to migrate its proclets to the other servers.

In these experiments, memory pressure comes from an antagonist process that allocates memory as fast as Linux's virtual memory subsystem permits (≈ 7 GB/s measured in our machine with 4K pages). Once the memory usage of the machine goes above the threshold, Nu starts to migrate proclets to free memory. A good result would show Nu migrating proclets sufficiently quickly to keep up with the allocation rate of the antagonist, without disrupting application performance. To assess the benefit of rapid migration, we compare Nu against a baseline that emulates MigrOS [61], a recent RDMA-based live migration system. To emulate MigrOS, we throttle Nu's migration speed to 600 MB/s on average with a 200 ms initial delay. Since migration speed is slower than the antagonist's memory allocation, the machine starts swapping. We swap to a fast device: Linux `brd`, a block device backed by RAM. (The common alternative—killing processes—is even more disruptive, wastes work, and yields no meaningful baseline.)

Figure 7a shows the 99.9th percentile latency of client requests in the SocialNetwork application. At t=3.9s, the antagonist starts allocating memory, and once Nu's runtime detects that the free memory size goes below 1 GiB (a configurable threshold) at 4.9s, it starts migrating proclets to another machine. During the migration, client-perceived latency increases by less than 19%. At t=5.7s, all proclets have migrated and latency recovers. Figure 7b shows the same experiment with the baseline (Nu emulating MigrOS's migration speed). Since it migrates memory slower than the antagonist requests, memory runs out at t=5s and Linux starts swapping. Thus, the 99.9th latency increases from 639μs to 206ms. The antagonist finishes at t=10s, and latency eventually recovers as memory use drops. Figure 8 summarizes the results for the same experiment on KVS and k-means, which show a similar trend (graphs in §A.1).

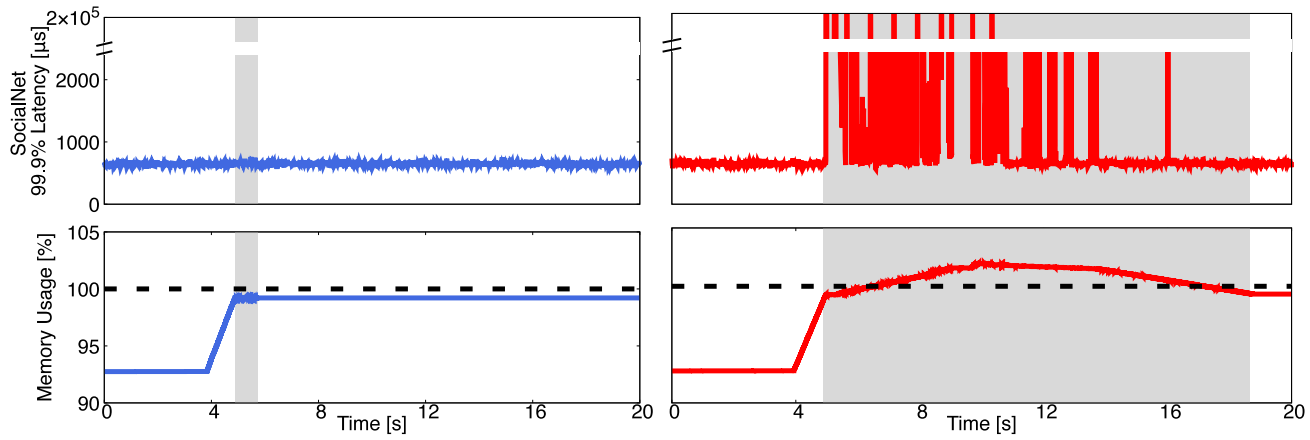
Compute pressure is harder to handle well than memory pressure as the CPU use can spike instantly. Figure 9 shows that Nu experiences a higher performance impact when faced with an antagonist that suddenly uses half the available CPU cores. However, disruption is still short-lived as Nu resolves pressure rapidly through fast migration. By contrast, the performance impact on the baseline lasts ≈15× longer.

These results show that Nu frees resources quickly under pressure, migrating proclets faster than Linux can allocate memory. Consequently, the pressuring workload (here, the antagonist) neither runs out of resources nor slows down, and the applications experience only modest tail latency increases. This means that Nu-based applications can use spare resources without risk: Nu can always migrate proclets if other workloads need the resources.

6.2 Comparison with Existing Implementations

Nu seeks to provide logical processes that match or exceed the performance of current architectures even in the absence of resource pressure. Although Nu allows distributed operation, local proclet invocations would ideally match the performance of computing on a single machine. We therefore compare the performance of Nu-based applications to baseline implementations without logical processes on a single machine. We measure tail latency under varying load for long-running services (SocialNetwork and KVS), and throughput for k-means. A good result would show Nu matching the baseline on NUMA-optimized, compute-intensive applications (e.g., Phoenix k-means), and it would outperform the baseline on RPC-based applications because Nu's fastpath avoids RPC overheads.

Figure 10 shows the results. Nu matches or exceeds the baseline's performance in all cases. For SocialNetwork (Figure 10a), Nu serves about 850k requests/second with sub-millisecond 99.9th percentile latency. The baseline implementation, which runs microservices in Docker containers and uses Thrift RPCs, scales to only 8,000 operations/second, with a 9–60 ms 99.9th-ile latency (very left of the graph). Nu outperforms the baseline because its fastpath avoids the overheads of loopback RPCs (serialization and network syscalls) with a single machine. For KV Store, Nu outperforms memcached on Linux by 15×, serving 12M operations/second to mem-



(a) Nu (fast migration).

(b) Baseline (slow migration).

Figure 7: SocialNetwork runs alongside a memory antagonist that starts at 3.9s. When the memory usage reaches the high watermark, Nu starts migrating proclets rapidly (the gray window). By matching the allocation speed of the antagonist, Nu keeps the memory usage flat and resolves the pressure in 0.86s. SocialNetwork's 99.9th-ile latency is unaffected. By contrast, the baseline fails to migrate fast enough and starts swapping, which leads to 206ms latency (322×). After the antagonist finishes, the memory usage and the latency gradually return to normal.

Workload [Disruption Effect]	**Baseline (Slow migration)**		**Nu (Fast migration)**	
	Duration	Effect	Duration	Effect
SocialNetwork 99.9th Lat. [μs]	9.14s	206ms (322×)	0.86s	761μs (1.2×)
KV Store 99.9th Lat. [μs]	60.94s	1.64s (>10,000×)	2.10s	85μs (2.6×)
K-Means Tput. [# iters/s]	0.73s	3.25 (-33%)	37ms	4.71 (-2.9%)

Figure 8: Under memory pressure, proclet migration with Nu sees shorter disruption and better performance during disruption than migration at state-of-the-art process live-migration speed (baseline).

Workload [Disruption Effect]	**Baseline (Slow migration)**		**Nu (Fast migration)**	
	Duration	Effect	Duration	Effect
KV Store 99.9th Lat. [μs]	7.96s	874μs (26.5×)	0.54s	1086μs (32.9×)
K-Means Tput. [# iters/s]	0.65s	2.41 (-50.3%)	24ms	4.57 (-5.8%)

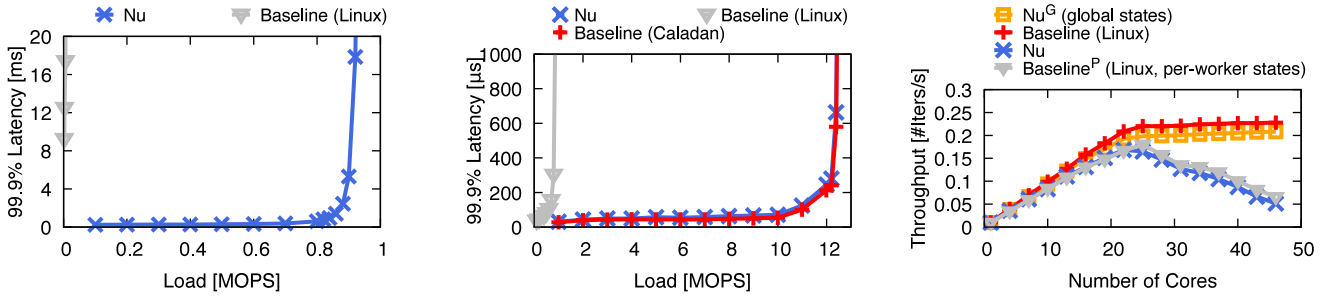
Figure 9: Under compute pressure, Nu sees short disruption and acceptable performance during proclet migration.

cached's 800k at sub-millisecond latency (Figure 10b). Crucially, Nu performs as well as the same KV Store running on Caladan [28], which also uses kernel-bypass networking and a user-level threading runtime. Finally, Nu matches Phoenix MR's performance (Figure 10c). Phoenix MR is designed for scalability on a single NUMA machine, and exploits shared memory for performance, so it is a strong baseline. The k-means workload requires sharing the intermediate clustering result across all workers. In a shared-memory setting, this shared state can be a global variable (as in the baseline), but in a distributed framework would involve per-worker copies. Since Nu supports migration, it must be prepared to operate distributedly and keep per-worker (per-proclet) copies, which amplifies the application's cache footprint on a single machine. We therefore compare two Nu setups: per-worker copies (label Nu) and global state (Nu^G), and add a modified baseline with per-worker states (Baseline^P). Nu^G measures the overhead of Nu's infrastructure with pinned (unmigratable) proclets. The overall results show that Nu's proclet invocations on a single machine are fast enough to match the performance of single-machine baselines.

6.3 Scalability

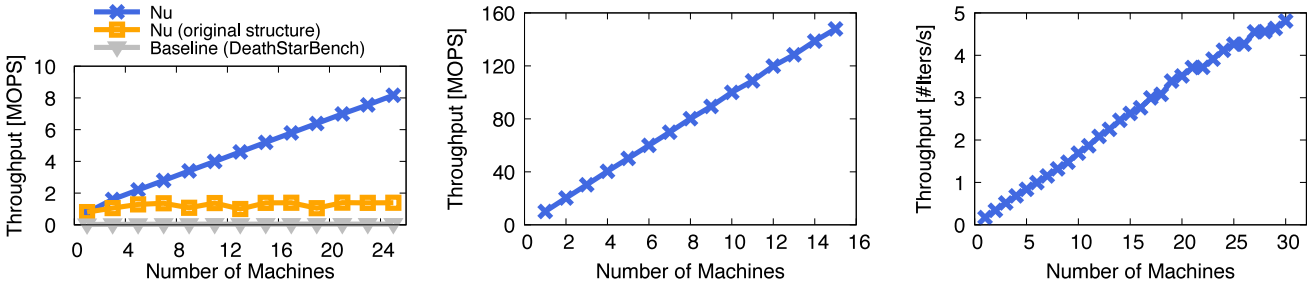
Next, we consider how Nu scales as the proclets of a logical process are spread across many machines. For each of our three applications, we run an experiment where the runtime assigns its proclets round-robin across servers. We consider two versions of the SocialNetwork application: the one from §6.2, which we wrote with logical processes and proclet decomposition in mind; and a second version that mirrors the exact microservice decomposition in DeathStarBench. We measure throughput for equal-sized input, i.e., a strong-scaling setup. Because Nu's local proclet invocation is faster than remote invocation, the single-machine setup has a substantial efficiency advantage, which makes linear scalability difficult to achieve. An ideal result would therefore show scalability close to linear as the number of machines increases.

We show the results in Figure 11. Nu scales well in all three applications, and achieves nearly linear scalability for KV Store and k-means. The SocialNetwork application is the most challenging to scale (Figure 11a). A direct port from the original microservice architecture to Nu (where each microservice becomes one proclet) results in many proclets with methods that have low compute intensity. When invoked remotely, calls to these methods can be costly, while the additional resources of a remote machine speed up the more compute-intensive invocations. On balance, Nu's throughput



(a) SocialNetwork **(b)** KV Store **(c)** K-means

Figure 10: Nu-based applications match or outperform baselines on a single machine. For SocialNetwork, Nu's fastpath helps it outperform the expensive RPCs in the baseline; in KV Store, Nu matches Caladan's [28] performance; and for k-means, Nu matches the baseline depending on how state is represented: as a global shared array (typical in a NUMA setting) or as per-worker arrays (as in distributed settings).



(a) SocialNetwork **(b)** KV Store **(c)** K-means

Figure 11: Nu scales well as the number of machines increases. The Nu-native SocialNetwork application, which merges the baseline's stateless microservices, scales better than the direct port of the baseline because its proclets better amortize the cost of remote invocations. K-means scales sub-linearly as the overhead of broadcasting each iteration's intermediate results increases with the number of machines.

still increases with the number of machines—from 786k to 1.37M ops/sec—but less so than when the application is decomposed into proclets that have sufficient compute intensity (786k to 8.44M ops/second). However, both Nu-based implementations perform one to two orders of magnitude better than the DeathStarBench baseline (45k ops/sec). The KVS implementation on Nu scales very well (Figure 11b) as it relies on client-side request steering (in response to hints from Nu's runtime) to direct clients' requests to the right machine, which then makes a local proclet invocation. K-means (Figure 11c) has high compute intensity, which makes scaling easy.

We conclude from these results that Nu's logical processes scale well when proclets are distributed across machines if a proclet's methods have sufficient compute intensity. §6.4.1 evaluates the impact of compute intensity on Nu's efficiency.

6.4 Design Drill-Down

6.4.1 Impact of Compute Intensity

We now examine the efficiency of Nu's mechanism for proclet method invocation (§3.2). Intuitively, the more compute an invocation does, the easier it is to amortize the overheads of the invocation (serialization, networking, and threading); yet, the lower these overheads are, the better Nu's performance becomes. Our experiment is a sensitivity analysis in which we vary the compute duration in a proclet's method between 0.1 and 50μs, and we measure the aggregate invocation throughput. We use sufficient threads to maximize throughput, saturating the machine that runs the proclet. We consider two cases for Nu: two proclets in the same machine (local), and proclets in different machines (remote). We compare the performance of Nu against three common mechanisms to invoke a task: (*i*) a function call in a Linux process; (*ii*) an RPC using Thrift, a popular open-source RPC framework [75]; and (*iii*) an RPC using a modified Thrift that uses Caladan [28] to reduce TCP and threading overheads. We measure throughput in a closed-loop setting. A good result would show performance of Nu close to local function calls for local invocations, and at least as good as Thrift for remote invocations.

The results in Figure 12 show that when the invocation is local, Nu's performance tracks closely that of Linux function calls. This happens because of Nu's fastpath for local invocations. When invocation is on a remote proclet, compute intensity (invocation duration) matters. For short invocations (0.1μs), Nu is $\approx$13$\times$ worse than local function calls, but 2.4$\times$ better than Caladan-based Thrift (and 29.4$\times$ better than Thrift on Linux). As the invocation becomes more compute-intensive, these gaps close: for a 10μs task, Nu's remote invocation achieves 85% of local function call throughput. We conclude that locality matters for remote proclet invocations with low compute intensity, but that Nu delivers near-single-

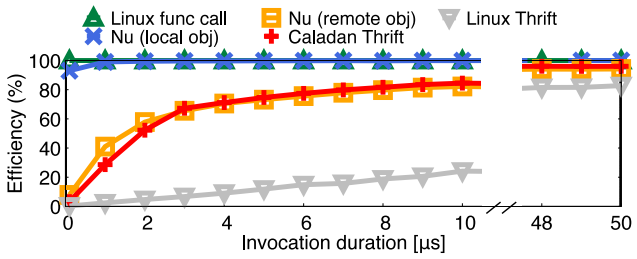


Figure 12: Efficiency (y-axis) of Nu invocations as a function of compute intensity (invocation duration), normalized to Linux function call throughput. Nu's local proclet invocation matches the performance of a function call, and Nu outperforms Linux Thrift by 2.4–29.4× for remote invocations when compute intensity is low.

machine performance for tasks with compute intensity as low as 10μs.

6.4.2 Migration Time and Bandwidth

We now measure the time it takes to migrate a proclet in Nu. The experiment migrates proclets of varying sizes to another machine and measures the migration time. We vary the test proclet's memory size by adjusting its heap size, from 64 KiB to 16 MiB. The proclet has a single thread with a small stack (64 bytes). A good migration latency would be ≈100μs for modest-sized proclets—orders of magnitude faster than traditional resource balancing mechanisms. For larger proclets, we expect the latency to approach network transfer time.

Heap Size	Migration Time [μs]		
	25 GbE	100 GbE	200 GbE
64 KiB	21	21	9
1 MiB	343	111	61
2 MiB	683	216	108
16 MiB	5,452	1,512	771

Figure 13: Nu migrates proclets with different heap sizes (64 KiB–16 MiB) faster with increasing network speeds (25/100/200GbE).

Figure 13 shows the results. With 100 GbE (i.e., the network setting used for all other experiments), Nu migrates small proclets (up to 1 MiB) in under 125μs. This corresponds to a bandwidth of 3–9 GB/s. For larger proclets (2 MiB–16 MiB), the latency varies from 200μs to 1,500μs, which corresponds to a bandwidth of ≈11 GB/s, close to the 100 GbE line rate. The results of 25 GbE and 200 GbE show similar trends. Proclet migration benefits from higher network bandwidth; for example, with 200 GbE, Nu only takes ≈100μs to migrate a 2 MiB proclet. We conclude that Nu migrates proclets quickly and that its migration uses the network efficiently.

6.4.3 Controller Performance

To understand whether Nu's controller can become a performance bottleneck, we benchmark it as a standalone component to measure its capacity. Depending on the type of control message, the controller achieves 0.79-0.96 million msg/s. This is two to three orders of magnitude higher than the real load demand (542–21,450 msg/s) we measured in the end-to-end experiments (§6.1). This makes sense as Nu's runtime caches the proclet location resolution result, thereby moving the controller off the critical path of steady-state application traffic. The controller is only involved in the control plane of initial proclet location resolution and migration.

6.4.4 Proclet Replication

Nu allows replicating proclets for fault-tolerant operation. Replication imposes overhead because it forwards all invocations of a proclet to a backup in a different machine (§3.5). We measure the invocation throughput of calling 8,192 remote replicated proclets, as we vary the compute intensity as in §6.4.1. These invocations do not have sub-operations. The baseline is the same setup without replication. A good result would show a modest loss of throughput with replication.

	Compute Intensity [μs]				
Throughput [MOPS]	0.1	1	10	20	30
with replication	13.18	10.56	2.52	1.52	1.12
without replication	21.04	14.86	3.56	1.97	1.37

Figure 14: Replicated proclets achieve 63–82% of unreplicated throughput, depending on compute intensity.

Figure 14 shows the results. Throughput drops by 37% with a 0.1μs compute intensity, but this drop gradually shrinks to 18% as compute intensity grows to 30μs. Replication adds ≈1.2μs to each operation to invoke the backup proclet, an overhead that gets amortized at larger compute intensities. This result shows that fault-tolerance for critical proclets is feasible and need not come at severe performance cost.

7 Conclusion

We presented logical processes, a new abstraction that decomposes an application into proclets, which are small units of state and compute. Logical processes and proclets solve a key hindrance to increasing datacenter resource utilization: the lack of microsecond-granularity fungibility in resource use.

We found that logical processes and our Nu prototype improve fungibility by making applications granular and migrating proclets quickly under resource pressure. For three applications, Nu matches the performance of strong baselines, scales well, and migrates their proclets within hundreds of microseconds with little disruption to application performance.

Nu is available as open-source software [66].

Acknowledgements

We thank our shepherd Dejan Kostić, the anonymous reviewers, Irene Zhang, Akshay Narayan, and members of the MIT PDOS group for their helpful feedback. We appreciate Cloudlab [24] for providing the experiment platform. This work was funded in part by a Facebook Research Award, a Google Faculty Award, the DARPA FastNICs program under contract #HR0011-20-C-0089, the NSF under award CNS-2104398, and VMware.

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. "Slicer: Auto-Sharding for Datacenter Applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[2] Pradeep Ambati, Iñigo Goiri, Felipe Vieira Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[3] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. "Treadmarks: Shared memory computing on networks of workstations". In: *IEEE Transactions on Computers (TC)* 29.2 (1996).

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. "A View of Cloud Computing". In: *Communications of the ACM (CACM)* 53.4 (2010).

[5] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. "UPC++: A High-Performance Communication Framework for Asynchronous Computation". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019.

[6] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. "Orca: a language for parallel programming of distributed systems". In: *IEEE Transactions on Software Engineering (TSE)* 18.3 (1992).

[7] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing locality and independence with logical regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 2012.

[9] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. "The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane". In: *ACM Transactions on Computer Systems (TOCS)* 34.4 (2017).

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-specific Memory Coherence". In: *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1990.

[11] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. "The CacheLib Caching Engine: Design and Experiences at Scale". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[12] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. "Hoard: A Scalable Memory Allocator for Multithreaded Applications". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2000.

[13] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. "The development of the Emerald programming language". In: *ACM SIGPLAN conference on History of programming languages*. 2007.

[14] Jeff Bonwick and Jonathan Adams. "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources". In: *USENIX Annual Technical Conference (ATC)*. 2001.

[15] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. "Orleans: cloud computing for everyone". In: *ACM Symposium on Cloud Computing (SoCC)*. 2011.

[16] *cereal: A C++11 library for serialization*. 2021. URL: https://github.com/USCiLab/cereal (visited on 09/20/2022).

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing". In: *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2005.

[18] Yue Cheng, Zheng Chai, and Ali Anwar. "Characterizing Co-Located Datacenter Workloads: An Alibaba Case Study". In: *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*. 2018.

[19] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. "Overload Control for μs-scale RPCs with Breakwater". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[20] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live Migration of Virtual Machines". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2005.

[21] *Common Object Request Broker Archictecture (CORBA)*. URL: https://www.omg.org/spec/CORBA (visited on 09/20/2022).

[22] *Checkpoint/Restore In Userspace (CRIU)*. URL: https://www.criu.org (visited on 09/20/2022).

[23] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2004.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. "The Design and Operation of CloudLab". In: *USENIX Annual Technical Conference (ATC)*. 2019.

[25] Michael J. Feeley, William E. Morgan, Frédéric H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. "Implementing Global Memory Management in a Workstation Cluster". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 1995.

[26] Message P Forum. *MPI: A Message-Passing Interface Standard*. Technical report. University of Tennessee, 1994.

[27] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing using Thousands of Tiny Threads". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[28] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. "Caladan: Mitigating Interference at Microsecond Timescales". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[29] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[30] Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. "A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications". In: *Proceedings of the VLDB Endowment (PVLDB)* 13.5 (2020).

[31] Martin Greenberger. *Management and the Computer of the Future*. Wiley, 1962.

[32] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with Infiniswap". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[33] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

[34] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek R. Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. "PerfIso: Performance Isolation for Commercial Latency-Sensitive Services". In: *USENIX Annual Technical Conference (ATC)*. 2018.

[35] *Kubernetes*. URL: https://kubernetes.io (visited on 09/20/2022).

[36] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. "Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[37] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. "Implementing linearizability at large scale and low latency". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2015.

[38] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[39] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. *First-generation Memory Disaggregation for Cloud Platforms*. 2022.

[40] Kai Li and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems". In: *ACM Transactions on Computer Systems (TOCS)* 7.4 (1989).

[41] Yilong Li, Seo Jin Park, and John Ousterhout. "MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2021.

[42] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[43] Barbara Liskov and James Cowling. *Viewstamped Replication Revisited*. Technical report. Massachusetts Institute of Technology, 2012.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. "Heracles: Improving resource efficiency at scale". In: *International Symposium on Computer Architecture (ISCA)*. 2015.

[45] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. "GraphLab: A New Framework for Parallel Machine Learning". In: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*. 2010.

[46] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. "Imbalance in the cloud: An analysis on Alibaba cluster trace". In: *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017.

[47] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 2008.

[48] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. "Graphite: A distributed parallel simulator for multicores". In: *IEEE Symposium on High Performance Computer Architecture (HPCA)*. 2010.

[49] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. "Process migration". In: *ACM Computing Surveys* 32.3 (2000).

[50] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[51] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. "Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2021.

[52] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-tolerant Software Distributed Shared Memory". In: *USENIX Annual Technical Conference (ATC)*. 2015.

[53] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. "Scaling Memcache at Facebook". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2013.

[54] Robert W. Numrich and John Reid. "Co-Array Fortran for Parallel Programming". In: *SIGPLAN Fortran Forum* 17.2 (1998).

[55] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *USENIX Annual Technical Conference (ATC)*. 2014.

[56] John Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 2018.

[57] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[58] Seo Jin Park. "Achieving both low latency and strong consistency in large-scale systems". PhD thesis. Stanford University, 2019.

[59] David A. Patterson, Garth A. Gibson, and Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". In: *International Conference on Management of Data (SIGMOD)*. 1988.

[60] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert Tappan Morris. "Improving network connection locality on multicore systems". In: *European Conference on Computer Systems (EuroSys)*. 2012.

[61] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. "MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications". In: *USENIX Annual Technical Conference (ATC)*. 2021.

[62] George Prekas, Marios Kogias, and Edouard Bugnion. "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[63] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. "Evaluating MapReduce for Multi-core and Multiprocessor Systems". In: *IEEE Symposium on High Performance Computer Architecture (HPCA)*. 2007.

[64] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. "TritonSort: A Balanced Large-Scale Sorting System". In: *ACM Transactions on Computer Systems (TOCS)* 31.1 (2013).

[65] Zhenyuan Ruan, Tong He, and Jason Cong. "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive". In: *USENIX Annual Technical Conference (ATC)*. 2019.

[66] Zhenyuan Ruan, Seo Jin Park, Adam Belay, Marcos K. Aguilera, and Malte Schwarzkopf. *Nu: Logical Processes for Resource Fungibility*. URL: https://github.com/Nu-NSDI23/Nu (visited on 09/20/2022).

[67] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. "AIFM: High-Performance, Application-Integrated Far Memory". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[68] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. "Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1996.

[69] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. "Fine-grain Access Control for Distributed Shared Memory". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1994.

[70] Malte Schwarzkopf. "Operating system support for warehouse-scale computing". PhD thesis. University of Cambridge Computer Laboratory, 2016.

[71] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters". In: *European Conference on Computer Systems (EuroSys)*. 2013.

[72] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *USENIX Annual Technical Conference (ATC)*. 2020.

[73] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. "FlightTracker: Consistency across Read-Optimized Online Stores at Facebook". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[74] David Teigland and Heinz Mauelshagen. "Volume Managers in Linux". In: *USENIX Annual Technical Conference (ATC)*. 2001.

[75] *Apache Thrift*. URL: https://thrift.apache.org (visited on 09/20/2022).

[76] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: The next Generation". In: *European Conference on Computer Systems (EuroSys)*. 2020.

[77] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. "TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters". In: *European Conference on Computer Systems (EuroSys)*. 2016.

[78] *VMware VirtualCenter User's Manual*. 2003. URL: https://www.vmware.com/pdf/VirtualCenter_Users_Manual.pdf (visited on 09/20/2022).

[79] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. "Semeru: A Memory-Disaggregated Managed Runtime". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[80] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. "Ship Compute or Ship Data? Why Not Both?" In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2021.

[81] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[82] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. “Customizable and Extensible Deployment for Mobile/Cloud Applications”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[83] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. “GiantVM: A Type-II Hypervisor Implementing Many-to-One Virtualization”. In: *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 2020.

[84] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. “CPI2: CPU Performance Isolation for Shared Compute Clusters”. In: *European Conference on Computer Systems (EuroSys)*. 2013.

A Appendix

In this appendix, we include the end-to-end performance results under resource pressure that were not included in §6.1 due to the space constraint.

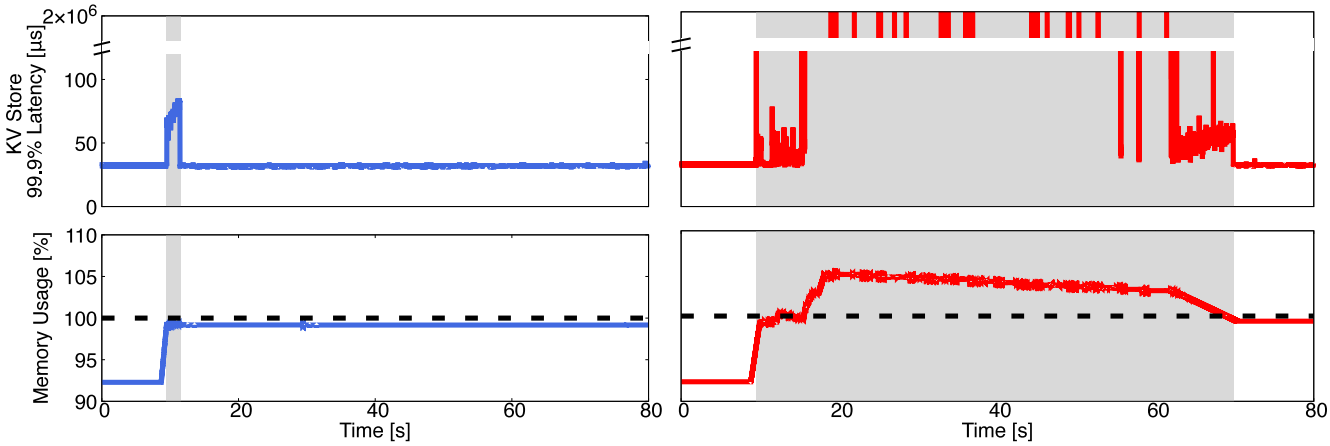
A.1 Application Performance Under Memory Pressure

Figure 15 presents the 99.9th tail latency of KV store under memory pressure. The results are similar to the SocialNetwork results (Figure 7). Figure 16 shows the K-Means performance under memory pressure using the throughput metrics as it is a batch application. Nu achieves 97% throughput during migration, whereas the baseline only achieves 67% throughput. Here we do not show the memory utilization as K-Means has a tiny per-machine memory footprint. The baseline has lower performance mainly because of the long task pausing time caused by slow migration.

A.2 Application Performance Under Compute Pressure

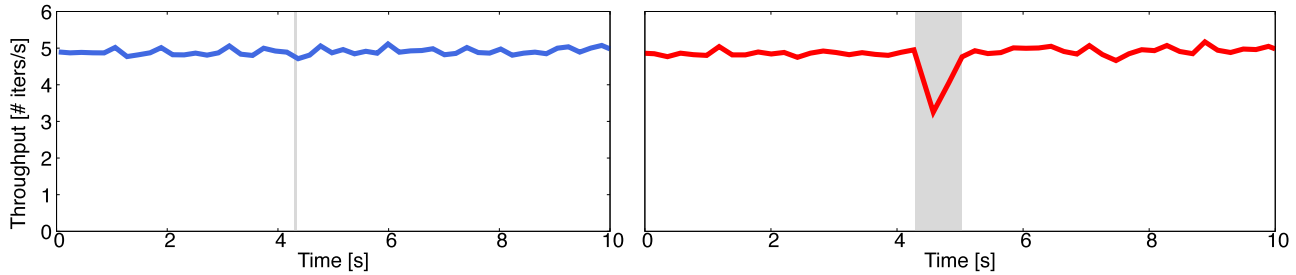
In the next experiment, we evaluate Nu's performance under compute pressure. Compute pressure is harder to handle well than memory pressure, since Nu's solution to resource pressure—proclet migration—itself consumes compute resources. The antagonist process in this experiment is a synthetic CPU-spinning workload that occupies half the CPU cores on the machine, reducing the compute resources available both to the application and to Nu's proclet migrations. In addition, the CPU load of the antagonist can spike instantly; this is different from the memory load which only increases gradually. Therefore, we would expect a higher impact on application performance than when Nu migrates proclets under memory pressure. A good result for Nu would show that the application still achieves acceptable performance, even if degraded for some (ideally short) time.

Figure 17a shows Nu's results. At t=4.9s, the compute pressure starts on one machine, taking away half of the application cores, and Nu immediately starts migrating proclets to reduce load on the machine. 99.9th latency increases from 33 μs to 1086 μs. This latency spike makes sense as the machine's compute resources are degraded by 50% and Nu needs additional compute to migrate proclets. The latency gradually decreases as proclets migrate and the other machine starts serving client requests, and soon recovers back to 33 μs as the migration ends at t=5.44s. In contrast, for the baseline, the latency disruption lasts 7.96s, which is 15X of the Nu's 0.54s duration. Figure 18 shows the result of K-means. Nu takes 24ms to resolve the pressure and achieves 94.2% throughput



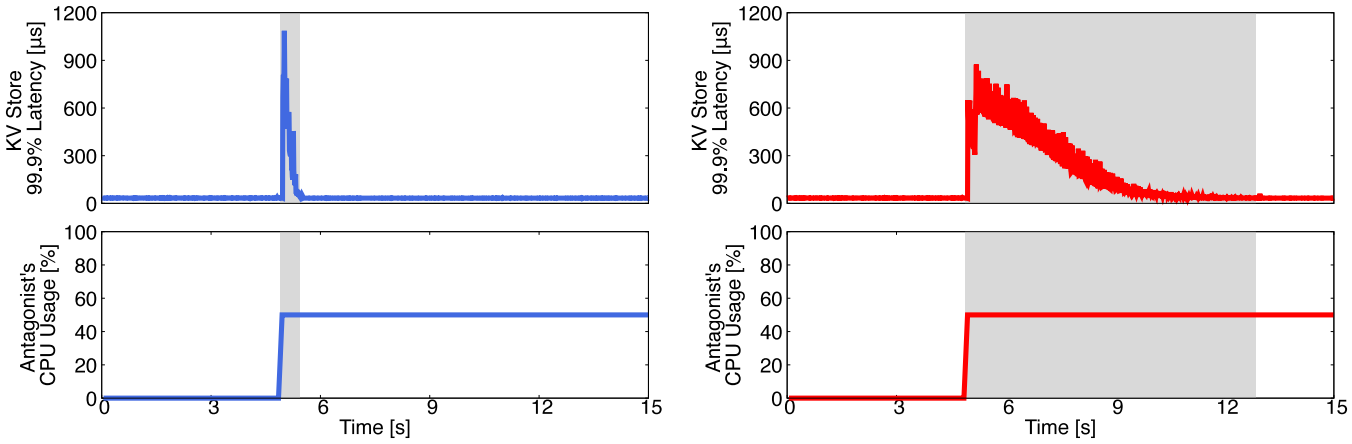
(a) Nu (fast migration). **(b)** Baseline (slow migration).

Figure 15: For KV store under memory pressure, Nu is able to maintain 99.9th tail latency under 85μs as it migrates proclets faster than the memory allocation speed of the antagonist. In contrast, the baseline suffers from poor tail latency ($\approx 2\times10^6\mu$s) since it cannot keep up with the allocation speed and has to swap memory.



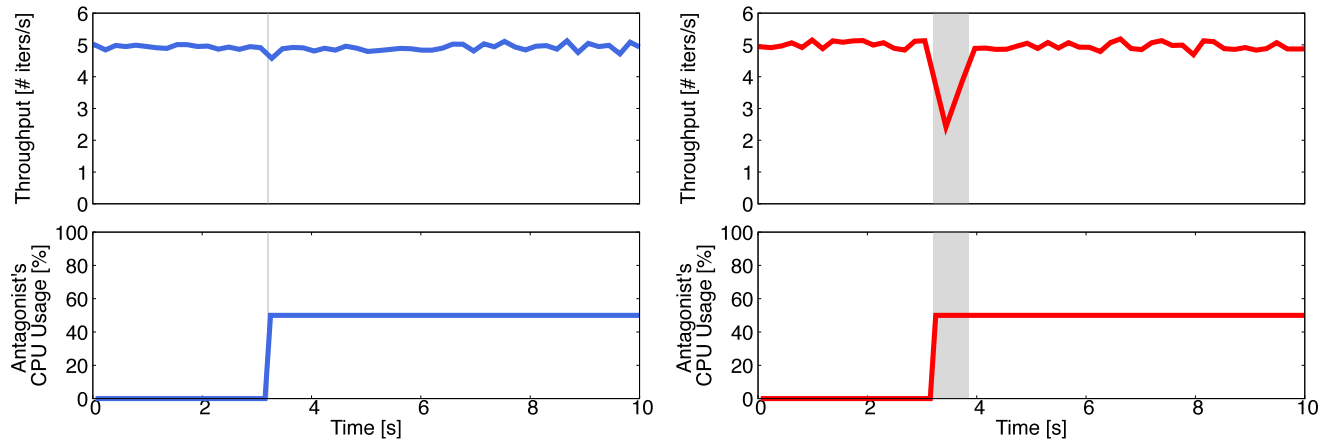
(a) Nu (fast migration). **(b)** Baseline (slow migration).

Figure 16: For K-means under memory pressure, Nu maintains stable throughput during migration, whereas the baseline only achieves 67% throughput. We do not show the memory utilization here as K-means has a tiny per-machine memory footprint.



(a) Nu (fast migration). **(b)** Baseline (slow migration).

Figure 17: Under compute pressure, the KV store server becomes overloaded and the client-perceived 99.9th tail latency spikes from 33 μs to $\approx$1 ms. Nu only takes 0.54 s to fully recover the performance, while the baseline requires 7.96 s ($\approx$15X).



(a) Nu (fast migration). **(b)** Baseline (slow migration).

Figure 18: For K-means under compute pressure, Nu takes 24ms to resolve the pressure and achieves 94.4% throughput during migration. In contrast, the baseline takes 0.65s and only achieves 49.7% throughput.

during migration. In contrast, the baseline takes 0.65s and only achieves 49.7% throughput.

These results demonstrate that Nu's logical processes react quickly to CPU pressure. Some performance degradation is unavoidable, but the impact is short-lived: after a sub-second delay, proclet migrations relieve the resource pressure.

Enabling High Quality Real-Time Communications with Adaptive Frame-Rate

Zili Meng[1,2], Tingfeng Wang[1,2,3], Yixin Shen[1], Bo Wang[1,4], Mingwei Xu[1,4],
Rui Han[2], Honghao Liu[2], Venkat Arun[5], Hongxin Hu[6], Xue Wei[2]
[1]*Tsinghua University,* [2]*Tencent Inc.,* [3]*Beijing University of Posts and Telecommunications,*
[4]*Zhongguancun Laboratory,* [5]*Massachusetts Institute of Technology,* [6]*University at Buffalo, SUNY*

Abstract

Emerging high-quality real-time communication (RTC) applications stream ultra-high-definition (UHD) videos with a high frame rate (HFR). They use edge computing, which enables high bandwidth and low latency streaming. Our measurements, from the cloud gaming platform of one of the largest gaming companies, show that, in this setting, the queue at the client-side decoder is often the cause of high latency that hurts the user's experience. We, therefore, propose an Adaptive Frame Rate (AFR) controller that helps achieve ultra-low latency by adaptively coordinating the frame rate with fluctuating network conditions and decoder capacity. AFR's design addresses two key challenges: (1) queue measurements do not provide timely feedback for the control loop; and (2) multiple factors control the decoder queue, and different actions must be taken depending on why the queue accumulates. Both trace-driven simulations and large-scale deployments in the wild demonstrate that AFR can reduce the tail queuing delay by up to 7.4× and the stuttering events measured by end-to-end delay by 34% on average. AFR has been deployed in production in our cloud gaming service for over one year.

1 Introduction

Emerging network technologies like 5G have gotten both academia and industry excited about high-quality real-time communication (RTC) applications with ultra-high definition (UHD), high frame rate (HFR), and reduced delays. Examples include cloud gaming [41, 82], virtual reality [37, 88, 63] and 4K video conferencing [40, 49]. Some high-quality RTC services have already been deployed in production (e.g., cloud gaming from Google [3], Microsoft [1], Nvidia [5]). For example, the market share of cloud gaming reached one billion dollars in 2020, with an expected growth rate of 30% [19].

To achieve a satisfactory user experience, those applications need to stream with high resolution, high frame rate, and a low delay (§2). For example, cloud gaming services deliver content with a resolution of ≥1080p [3] and frame-rate of 60fps [61], while requiring a tail end-to-end delay of less than 100ms [43]. Streaming like this significantly improves users' experience and enables new applications.

This paper argues that, in addition to modulating bitrate to match network capacity, a high-quality RTC system must regulate the queuing at the decoder queue. For traditional standard quality RTC, the time required to decode a frame is much shorter than the interarrival time of frames. Thus, the decoder queue is not a bottleneck and a traditional RTC service only needs to adjust the bitrate to match the network bandwidth. However, in high-quality RTC, the high frame rate reduces the time between the arrival of frames at the client, while the high resolution increases the decoding delay for each frame. At the decoder queue, the frame arrival rate frequently exceeds the departure rate, leading to a long queue, as shown in Figure 1. The video delivery is required to not only adapt the bit-rate to the network bandwidth but also coordinate with the decoder queue capacity. From measurements of our production cloud gaming service, Tencent Start [4], we find that video delivery without coordinating the queue capacity could introduce a non-negligible queuing delay at the client-side decoder queue. Moreover, such a queuing delay accounts for a large proportion of delayed frames in satisfying the much tighter delay requirement of high-quality RTC, especially when the network delay has been reduced with recent infrastructure developments (e.g., edge computing [57]). According to our measurements, among all frames with a total round-trip delay of >100ms, 57% of them have been delayed at the decoder queue for >50ms (§3.1). Our survey finds that the future demands of UHD and HFR video will further exacerbate the problem, even with the evolution of decoding hardware (§3.1). Therefore, for high-quality RTC, to reduce the end-to-end delay, it is essential to reduce the queuing delay at the decoder.

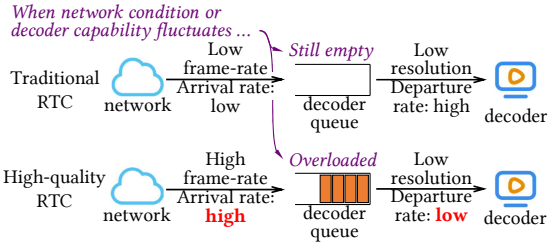


Figure 1: Comparison of the decoder queue between traditional and high-quality RTC applications. Due to the high frame rate and resolution, when network condition or decoder capability fluctuates, high-quality RTC applications may overload decoder queues, leading to high delay at the tail.

Not all interventions are effective at regulating the queuing at the decoder queue (§3.2). For instance, decoding delay is not affected much by bitrate. It is affected by resolution, but adjusting the resolution requires the client to request a new key frame. This consumes bandwidth and incurs several extra frame intervals of delay. Discarding a frame at the client also requires a new key frame, which incurs the same cost. Hence, we introduce an *adaptive frame-rate (AFR)* controller, which controls the frame rate at the *encoder*. Reducing frame rate gives the decoder more time to process frames. Hence, it is effective at reducing the queue length. Further, edge streaming services offer short RTTs, which means the control loop to adjust the encoder's frame rate is short.

Note, there have been previous efforts to adapt the frame-rate

(e.g., CU-SeeMe [38] decades ago). However, the development of decoding hardware had made it redundant in the recent decade, and traditional RTC in the recent decade is mostly bottlenecked in the network. In this paper, we show how high-quality RTC, with UHD resolution, HFR, and stringent delay requirements, has changed this. We further improve upon these proposals in two ways. First, existing control mechanisms are based on delay or queue length [60, 34, 77], which are slow to react since they need to wait for the queue to build up. AFR instead relies on the arrival and service processes in addition to the queue length to adjust the frame rate. Second, not all increases in decode queuing delay need to reduce the frame rate. For instance, when queuing delay increases due to a transient burst of arriving packets. Hence, AFR uses two control loops that adjust the frame rate at different time scales.

We implement the AFR controller on both simulators and the production of the cloud gaming service from Tencent Start [4]. Trace-driven simulations and deployments in the wild demonstrate that AFR could effectively reduce the tail queuing delay by up to 7.4×, and consequently reduce the ratio of frame stutters measured by total delay by up to 2.2× (§6.1 and §6.5) with negligible overhead. AFR has been deployed on Tencent Start since February 2021, serving millions of sessions. We will release the collected traces and the simulation code.

We make the following contributions:

- We carry out a month-long measurement campaign to motivate the significance of controlling queuing delay at the decoder queue, and identify the unique challenges from high-quality RTC with stringent delay requirements (§3).
- We design a hierarchical frame-rate controller, AFR, to control the decoder queue towards an ultra-short delay under different scenarios for high-quality RTC (§4).
- We evaluate AFR with both trace-driven simulations and large-scale deployments in production in the wild (§5). Our evaluation shows that both queuing delay and total end-to-end delay could be significantly improved (§6). AFR has been used in deployment for over one year.

2 Background: High-Quality RTC

High-quality RTC applications are attracting attention from the industry and academia. A series of high-quality RTC products have been released recently, including cloud gaming [3, 1, 5], virtual reality (VR) [12, 11, 6], and 4K videoconferencing [8]. For example, by generating high-quality content and streaming to the user via Internet, users can enjoy better video quality with low-cost devices. Specifically, the high-quality RTC has the following features standing out from traditional RTC applications:

- *High frame-rate*. Traditional RTC usually delivers content with a low frame rate (LFR) of 24fps [9]. However, high-quality RTC requires a frame rate of up to 60fps, some of which even require a frame-rate of 240fps [73].
- *High resolution*. Most existing RTC applications are delivered at SD resolutions by default (e.g., 360p for Google Meet [7]). In contrast, high-quality RTC applications require a resolution of 1080p to 4K or higher [62].
- *Stringent delay requirement*. Furthermore, high-quality RTC applications also have stringent latency requirements. For example, videoconferencing requires a round-trip interaction delay of 150ms [9] and gaming for 100ms [43].

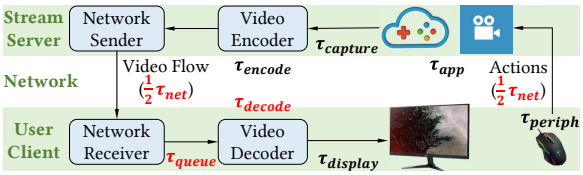


Figure 2: A general delivery pipeline of RTC services. We highlight the major contributing components in the tail end-to-end delay of high-quality RTC according to our measurements in red.

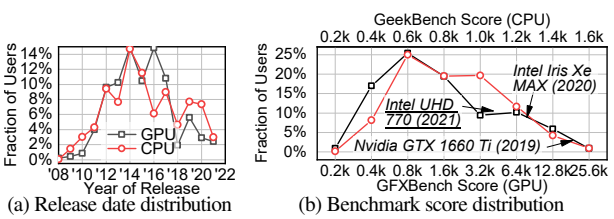
Existing delivery pipeline. To better understand the bottleneck of high-quality RTC, we present the key components of the existing RTC delivery pipeline in Figure 2. First, the video encoder captures the contents generated from video sources (e.g., gaming applications [23, 57]) and encodes them into video frames. Then, encoded frames are sent over the network from the streaming server to user clients. After that, on the client side, upon receiving new frames from the network, the decoder will decode those frames. Finally, decoded video frames will be displayed on users' displays.

Optimization goal: low tail delay. With the intelligence from each community, the delay of each component has been intensively optimized in recent research efforts. To reduce the network delay, existing providers either deploy stream servers at the edge [57, 74], introduce low-latency congestion controllers [16, 25], or suggest users use wired connections. For example, recent measurements unveil that cloud gaming services could deliver the RTC streams with an average round-trip network delay of 20ms [57, 26]. Similarly, streaming encoders are optimized for low latency to satisfy the stringent delay requirements in high-quality RTC services [58, 69, 34].

Meanwhile, optimizing the *tail* performance is also critical for user's experience for high-quality RTC [56]. The increase in tail delay will result in frame stuttering or freezing, degrading the user's experience. Quality of experience assessment frameworks in video streaming usually individually calculate the stuttering time as a penalty to the user's experience [33, 80]. Considering the high frame rate of high-quality RTC, further tails of 99th or 99.9th percentiles need to be focused on. For example, at the frame rate of 60fps, even the 99.9th percentile delay could happen every 16 seconds. Especially for applications such as cloud gaming, such a delay might lead to the loss of the game (e.g., stalls when the gamer is discovered by the opponent in a shooting game) [67, 43]. Therefore, it is essential to control the tail delay and reduce frame stutters for high-quality RTC.

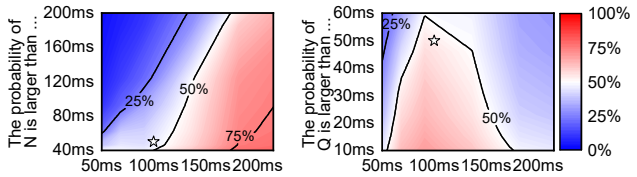
3 Motivations and Challenges

In this section, we first explain the formulation of drastic queuing delay in high-quality RTC (§3.1). We then present our thinking over the design choice of adjusting frame rate (§3.2). We



(a) Release date distribution (b) Benchmark score distribution

Figure 3: Release year and benchmark score distribution of user devices in production. We use the single-core score in GeekBench [15] for the CPU benchmark and Aztec Ruins Normal Tier score in GFXBench [13] for the GPU benchmark.



(a) Network delay. (b) Queuing delay.

Figure 4: While network delay should usually be blamed when the total delay is above 200ms, queuing delay plays a dominant role among all frames with a total delay of more than 100ms. The color indicates the conditional probability $P(X > X_{th} | T > T_{th})$ for $X \in \{N, Q\}$. Stars denote X_{th}=50ms, T_{th}=100ms.

further analyze the unique challenges of effectively achieving an ultra-short queue (§3.3).

3.1 Motivation: Drastic Queuing Delay

Observation: decoder queuing delay is a critical contributor to the total delay at the tail. We profile the delay of each frame at each stage in the delivery pipeline in Figure 2. We measure the Tencent Start cloud gaming service for a month in 2021, containing tens of thousands of users, with thousands of different CPU and GPU models. We present release dates and benchmark scores of CPU and GPU in Figure 3 and list top models in Appendix B.1. Unless other specified, all measurements in this paper are analyzed from this dataset.

According to our measurements, among all components in the pipeline, the network, queuing (at the decoder queue), and decoding delay are >10ms at the 99th percentile. We highlight them in red in Figure 2. The tail of the application and encoding delay is light since they are processed on commercial servers, which are stable compared to networks and heterogeneous clients. Therefore, we focus on the network, queuing, and decoding delay in the following discussion. We leave the measurement results to Appendix B.2.

We investigate how these three components contribute to the increase of total delay at the tail. For each frame, we denote N, Q, D, and T as the network, queuing, decoding, and total end-to-end delay. We then calculate the conditional probability of $P(X > X_{th} | T > T_{th})$ for each $X \in \{Q, D, C\}$ from our measurements, where X_{th} and T_{th} are thresholds for statistics. A high conditional probability suggests that the component is more likely the cause of $T > T_{th}$. We calculate the conditional probability with different thresholds, and present the results for

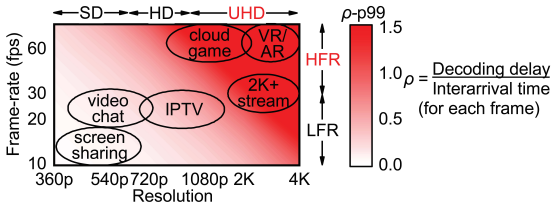


Figure 5: Illustration of the 99th percentile of the utilization ρ of the decoder queue. For high-quality RTC applications (in the top-right corner), the decoder queue is heavily loaded at the tail (shaded red), resulting in an increase of queuing delay at the tail.

network delay and queuing delay in Figure 4.

As we can see, when analyzing the root causes of frames with T>200ms for traditional RTC services, network delay has a high probability (shaded red) to be blamed. However, when analyzing the frames with T>100ms, queuing delay dominates the increase of total delay. Our measurements show that among all frames with an end-to-end total delay of more than 100ms, queuing delay increase happens more frequently than all other component delays: 57% of them have a queuing delay of more than 50ms (stars in Figure 4). Considering the stringent delay requirement of ∼100ms for high-quality RTC, the increase in queuing delay plays a dominant role.

Root cause: The UHD resolution and HFR jointly contribute to the increase in queuing delay. Compared to LFR streaming, HFR increases the arrival rate of the decoder queue by reducing the interarrival time between frames. Also, UHD decreases the departure rate compared to SD streaming by increasing the decoding delay of each frame.

Specifically, we illustrate how the frame rate and resolution could affect the load of the decoder queue by presenting the 99%ile queue utilization in Figure 5. We scale the distribution of interarrival time and decoding delay from our measurements to other frame rates and resolutions. As we can see, for traditional RTC services (the down-left corner), due to their low frame rates and resolutions, the decoder queue still has a utilization of $\rho \ll 1$ at the tail. However, for high-quality RTC applications (the up-right corner), the decoder queue would be heavily loaded, leading to a drastic queuing delay.

The issue is the inconsistency of the decoder's performance *on average* and *at tail*. In fact, many of the hardware decoders that we measured claim to support UHD and HFR videos (e.g., Nvidia GTX series in Table 4). However, according to our measurement, supporting UHD and HFR does not really mean *consistently* supporting. For example, the decoding delay can fluctuate due to numerous reasons including overheating at the client [64], CPU scheduling (§5.1), and the prediction errors [48], all of which are difficult to control for an application. From our measurement with devices in production, the decoding delay is 18ms at the 99th percentile even with hardware acceleration (Appendix B.2). Note that at the frame rate of 60fps, the interarrival time between frames is 16.7ms, resulting in a heavily loaded decoder queue at the tail.

We further analyze the necessity and sufficiency between the

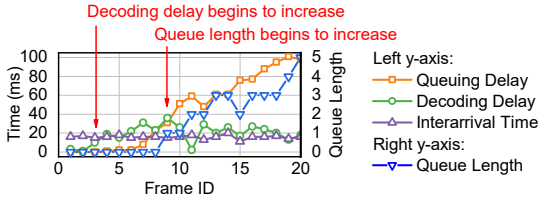
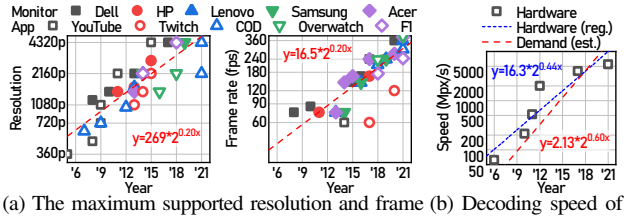


Figure 6: A trace for the accumulation of decoder queue. Note that this is an illustrative example – the distribution of all traces can be found in Appendix B.4.



(a) The maximum supported resolution and frame rate for the top 5 monitor vendors, two streaming platforms (YouTube and Twitch) and three games (Call-of-duty, Overwatch, and F1) [10].

(b) Decoding speed of existing hardware and required decoding speed from demands.

Figure 7: Decoding hardware cannot keep pace with the rapid increase of demands of videos with high resolution and frame rate. Note that the required decoding speed from demands is the frame rate times the *square* of resolution times the aspect ratio.

increase of other components and total delay in Appendix B.3 and figure out that the minor fluctuation of decoding delay leads to the increase of queueing delay. From the queuing theory, when the queue is heavily loaded, the queuing delay will drastically increase [32]. This is because while the decoding delay is continuously fluctuating, the queuing delay is accumulating all the fluctuations of precedent frames. Especially in heavy traffic, a minor fluctuation of the decoding delay could result in a magnitude increase in queuing delay. We refer the readers to [32] for more theoretical analysis. Illustratively, we present a trace from our production service in Figure 6. In the trace, the interarrival time is 16ms, and the decoding delay is 18ms, while the queuing delay is 54ms on average. The continual increase of the decoding delay, although not much by magnitude (18ms) and not long by duration (20 frames, approximately 0.3s), leads to a drastic queuing delay. If such a trace happens with a probability of 1%, we will have a 99th percentile decoding delay of 18ms, and a 99th percentile queuing delay of 55ms. In this case, the tail queuing delay is much higher than the decoding delay, which also contributes to more than half of the end-to-end stutters as analyzed in §3.1.

Trend: hardware decoders cannot keep pace with the increasing demands of UHD and HFR video. User demands for video have increased sharply, as shown in Figure 7(a). For example, the highest supported resolution and frame rate of YouTube have increased from 360p@30fps (7Mpx/s) in 2005 to 8K@60fps in 2015 (2Gpx/s), doubling every 14 months on average. Emerging services at 16K [85, 62] or 240fps [73] further indicate the future demands of UHD and HFR streaming.

However, the decoding speed of the hardware is not increasing as fast. We summarize the decoding speed of state-of-the-art video decoders from recent academic papers [53, 30, 90, 91, 89, 85]. As shown in Figure 7(b), the decoding speed of the state-of-the-art decoding hardware doubles only approximately every 27 months (blue dotted line). Meanwhile, we also calculate the required decoding speed from the existing demands of videos by multiplying the estimated resolution and frame rate from Figure 7(a) and plot the estimation in red in Figure 7(b). The required decoding speed from demands, doubling every 20 months, (red dashed line) increases much faster than the development of decoding hardware (blue dotted line), indicating the continuous incapability of decoding hardware for UHD and HFR videos.

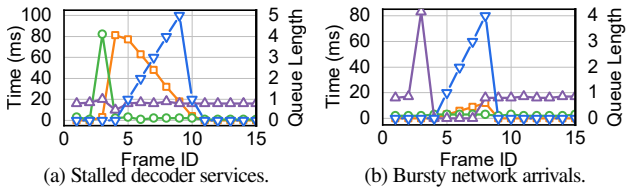
In addition to the state-of-the-art hardware, there are still a considerable number of low-end and mid-end devices in our users. User devices, even in the same generation, could also be very heterogeneous. For example, in Figure 3, notice that the performance of Intel Iris Xe is 2× better than Intel UHD 770 even though the latter is more recent. Thus, there is heterogeneity in user devices even in the same generation. Moreover, new video codecs (e.g., H.265), although with a higher compression ratio, even slow down the decoding speed by up to 60% [24, 55, 21]. In this case, the mismatch between the decoder and UHD and HFR videos will further exacerbate, making the queuing delay at the tail a lasting issue.

3.2 Choice: Controlling Proper Parameters

We motivate the need to adjust the frame rate. For an encoder, there are three parameters that could be independently set, including the frame rate, bit rate, and resolution. The encoder will automatically optimize other parameters (e.g., quantization parameters) based on current contents to achieve the target frame rate, bit rate, and resolution. We refer readers to [17] for more details on video codec.

We first analyze how these parameters could affect the delay of different components. When the bit-rate increases, the network delay will increase due to the congestion. When the resolution increases, since the decoder needs to decode frames with larger pixels, it needs a longer time to decode. The queuing delay depends on the enqueue rate (i.e., frame-rate) and the dequeue rate (i.e., decoding delay). In contrast, for example, if the bit-rate decreases, yet the resolution is kept the same, the decoding delay for each frame will hardly decrease due to the hardware design of the codec, which we further measure in Appendix B.4. Thus, relying on the total delay (e.g., Salsify [34]) would lead to ambiguity in taking effective actions to reduce the delay.

Therefore, we need to individually control respective parameters to reduce different delays. In response, we adjust the frame rate to control the queuing delay for high-quality RTC. When the fluctuations of the decoder and network result in an increase of queuing delay, it is essential to adjust the encoding parameters to reduce the queuing delay. In this case, after collecting measurements from the client and network, the encoder at the server could accordingly adjust the frame rate for the following frames. We could dynamically specify certain timestamps where new frames are encoded.



(a) Stalled decoder services. (b) Bursty network arrivals.

Figure 8: Two traces of transient fluctuations of the decoder queue from online traces. Legends are the same as Figure 6.

We further discuss several potential solutions and concerns of adapting frame rates in Appendix A. In summary, adjusting the resolution or dropping frames is impractical due to the significant overhead of bandwidth. Statically choosing the frame rate based on the client model is also insufficient due to the fluctuation of decoding delay in the runtime. Moreover, since applications have limited control over users' systems, it is also impractical to control the user's system (e.g., pinning the application to a CPU core) for a large-scale production-level service [14]. In terms of frame-rate adaption, note that there are previous efforts in the adaption of frame-rate (e.g., CU-SeeMe [38] decades ago). However, as we discussed in §3.1, with the increase in resolution and frame-rate, and the stringent delay requirements, we need to reemphasize the significance of adapting frame rate now. We also show that it is timely enough to control the frame rate over the Internet.

3.3 Challenges

Achieving an ultra-short queue. To achieve an ultra-short queuing delay for the decoder queue, it is challenging to pick the appropriate indicator to inform the controller when it needs to take action. Existing signals (queue length [60] or queuing delay [77, 34]) fail to achieve an ultra-short queuing delay. Since the accumulation of the decoder queue is the consequence of the fluctuation of the arrival or departure process, both the queue length and queuing delay can only be observed when the queue has already been built up. For the example in Figure 6, while the decoding delay starts to increase at the 3rd frame, a non-zero queue length can only be observed by the 9th frame. We also evaluate baselines based on queue length and queuing delay in §5.2.

In response, we want to capture the *earliest signal* to perceive the potential queuing delay. Therefore, instead of measuring the queuing delay, we want to estimate the potential increase of queuing delay predictively. For example, inspired by recent advances in congestion control [36, 50], a straightforward way is to measure the dequeue rate of the decoder queue to estimate the potential increase of the queuing delay.

However, in terms of tails, the arrival process is also fluctuating, which could also lead to an increase in queuing delays. For example, the network delay might increase by ten times at the 99th percentile than the median [25]. In response, to precisely avoid queue accumulation, we extend the designs of [36, 50]: AFR comprehensively measures the arrival and departure process and controls the queuing delay based on queuing theory. We introduce the design in §4.2, and evaluate the necessity of measuring the arrival process in §5.2.

Handling various events. Furthermore, the reason behind the formulation of the decoder queue in high-quality RTC is complex. As we introduced in §3.1, the stationary degradation of decoding capacity could lead to the accumulation of the decoder queue, e.g., the traces in Figure 6. Besides, the decoder queue could also be accumulated due to transient contingencies. For example, from our experiences in production, the decoder might contingently experience a sudden decoding lag of $\sim$100 milliseconds (e.g., the 3rd frame in Figure 8(a)). The sudden interference in wireless channels might also lead to the bursty arrival of several frames (e.g., the 4th to 8th frames in Figure 8(b)). In both cases, the decoder queue will be accumulated. Since these transient fluctuations happen suddenly, it is challenging for the controller to react by measuring enqueue and dequeue rates.

Algorithm 1: Hierarchical AFR control.

Input: Enqueue process $\{A_n\}$, dequeue process $\{S_n\}$, queue states Q. (A_n denotes the interarrival times, and S_n denotes the decoding delays of frames $\{n\}$.)
Output: Target frame rate f.
1 $f_0 =$ StationaryController($\{A_n\}$,$\{S_n\}$)
2 $\alpha =$ TransientController(Q)
3 $f = \alpha f_1$

Thus, AFR differentiates the causes of queue accumulation and reacts respectively to fluctuations at different time scales. We design a stationary controller to avoid queue accumulation in heavy traffic (§4.2), and a transient controller to reduce the queuing delay in contingencies (§4.3).

4 Design – Adaptive Frame-Rate (AFR)

We first analyze the overall workflow of AFR in §4.1, and then present the two controllers of AFR (§4.2, §4.3).

4.1 Workflow Overview

The workflow of AFR is presented in Algorithm 1. Specifically, the stationary controller (§4.2) maintains the queue around an ultra-short target based on dynamics of enqueue and dequeue processes. By measuring the statistics of both processes, AFR calculates the expectation of the queuing delay based on queuing theory. The frame rate can therefore be optimized towards a given queuing delay target (line 1). The transient controller observes the queue states Q (queue length and queuing delay) and calculates the discounting factor $\alpha \leqslant 1$ (line 2) to further decrease the frame rate when the queue formulates. The final frame-rate is the stationary frame-rate f_0 discounted by α (line 3). In this case, AFR can react to various scenarios of queue accumulation.

4.2 Stationary Controller

As introduced above, we measure the arrival and service processes and control the expected queuing delay of the queue. Specifically, we use the Kingman formula as an approximation of the expectation of queuing delay. Kingman formula is a widely adopted approximation formula of queuing delay [45] for G/G/1 queues. Compared to other approximation methods, in this paper, we adopt the Kingman formula to estimate the queuing delay since its estimation is from *both arrival and departure processes*

without relying on queue states, which could provide the earliest signal for the potential queuing delay. According to the Kingman formula, the expectation of queuing delay τ_{queue} follows:

$$\mathbb{E}(\tau_{queue}) \approx \left(\frac{\rho}{1-\rho}\right)\left(\frac{c_a^2+c_s^2}{2}\right)\mu_s \quad (1)$$

where

$$c_a = \sigma_a/\mu_a, \quad c_s = \sigma_s/\mu_s, \quad \rho = \mu_a/\mu_s \quad (2)$$

(μ_a, σ_a) and (μ_s, σ_s) are the mean and standard deviation of the arrival and service processes:

$$\mu_a = \mathbb{E}\{A_n\}, \sigma_a = \sqrt{var(A_n)}, \mu_s = \mathbb{E}\{S_n\}, \sigma_s = \sqrt{var(S_n)} \quad (3)$$

From Eq. 1, the queuing delay is related to the following factors:

- Queue utilization ρ. The queuing delay will increase when the queue is overloaded ($\rho \to 1$). The current frame rate and decoding delay determine the queue utilization.
- Arrival and service fluctuations c_a and c_s. When the arrival or the service processes fluctuate, the queuing delay will also increase.
- Service time μ_s. Finally, the queuing delay scales with the average decoding delay.

Therefore, we control the expected queuing delay by controlling the right-hand side (RHS) of Eq. 1. We set $\mathbb{E}\{\tau_{queue}\}$ to a pre-defined queuing delay target W_0. Consequently, the target frame-rate f_0 could be calculated as:

$$f_0 = \rho/\mu_s = 1/\left(\mu_s \cdot \left(1 + \frac{\mu_s}{W_0} \cdot \frac{c_a^2+c_s^2}{2}\right)\right) \quad (4)$$

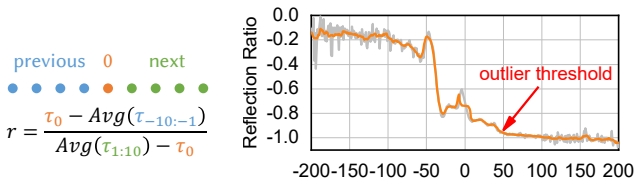
Discussion: Approximation method. The AFR mechanism supports any approximation formula by design. There are other research efforts to control the queue. For example, recent efforts in congestion control [36, 50] directly set the target utilization (e.g., setting $\rho = 0.95$) and calculate the enqueue rate. In this paper, we adopt Kingman formula to capture *both the arrival and departure processes*, as discussed in §3.3. We also evaluate the performance of other baselines in §6.1.

Measurements of queuing dynamics. According to Eq. 4, we need to measure the mean and variance of the arrival and service processes. Similar to the RTT measurements in TCP [44], we adopt the exponentially weighted moving average (EWMA) and exponentially weighted moving variance (EWMV) to estimate the $\mu_s, \sigma_s, \mu_a, \sigma_a$ in Eq. 1 and 2.

$$\begin{aligned} \hat{\mu}_n &= \xi_\mu x_n + (1-\xi_\mu)\hat{\mu}_{n-1} \\ \hat{\sigma}_n &= \sqrt{\xi_\sigma (x_n - \hat{\mu}_n)^2 + (1-\xi_\sigma)\hat{\sigma}_{n-1}^2} \end{aligned} \quad (5)$$

where x_n denotes interarrival time A_n or service time S_n. $\hat{\mu}_n$ and $\hat{\sigma}_n$ are the EWMA and EWMV. ξ_μ and ξ_σ are the discounting factors for the measurement of mean and standard deviation, trading off between precision and sensitivity.

However, due to bursty arrival or stalled services (§4.1), both the arrival and service processes could have significantly deviated value. For example, the 3rd frame in Figure 8(a) has a decoding time of 82ms while other frames are below



(a) Illustration (b) Measurements in production

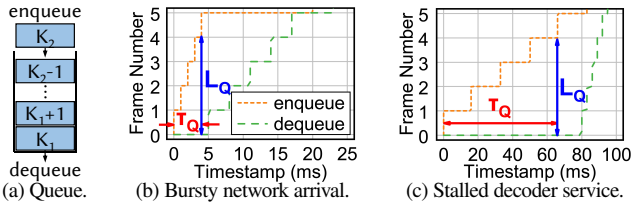
Figure 9: Reflection in outlier removal. Figure 9(b) presents the frequency of frames with $r \in [\frac{1}{C}, C]$. Measurement details in §5.2.

4ms. Such outliers will significantly deviate the estimation of stationary statistics for a long period. In fact, as we discussed in §4.1, these contingent events are designed to be handled by the transient controller. Therefore, we need to filter those outliers out to precisely estimate the stationary status of arrival and service processes. Due to the highly skewed distribution of decoding delay, existing outlier removal mechanisms based on standard deviation (e.g., the three-σ rule [65, 68]) suffer from differentiating stationary state transitions from outliers.

To capture the transitions of the status of decoders while eliminating the influence of the contingent outliers, we introduce an outlier removal mechanism based on priori knowledge from measurements in production. The key intuition is that *decoding delay differences* $(S_n - S_{n-1})$ are related to the probability of being outliers. For example, an increase of 20ms on decoding delay is probably the transition between stationary states (Figure 6). However, a sudden increase of 80ms on decoding delay is likely to indicate that decoding delay is an outlier, which is usually the scenario of contingent stalls in Figure 8(a). This is because commercial decoders are usually able to decode frames at the frame rate of 24fps on average. According to our measurements, when the decoding delay difference is above 50ms, the possibility of being an outlier for that frame is 95%. Thus, we remove frames with a decoding delay difference of >50ms in the stationary controller, and leave the control of those frames to the transient controller.

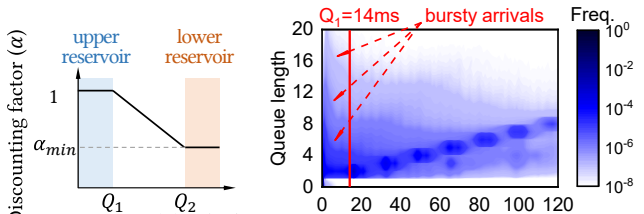
We further characterize our observation based on measurements in production. As shown in Figure 9(a), we quantify the outlier with *reflection ratio r*, which illustrates the recovery of decoding delay before and after the potential outlier. The numerator is the difference between the current decoding delay (τ_0) and the average decoding delay of the previous 10 frames $(\tau_{-10:-1})$, and the denominator is the difference between τ_0 and future decoding delay. For outliers of contingent stalled service (e.g., the 3rd frame in Figure 8(a)), their reflection ratios would approach -1. This is because previous frames and subsequent frames have similar decoding delays, while the outlier has a much higher decoding delay $(\tau_0 \gg \tau_{-10:-1} \approx \tau_{1:10})$.

We then plot the relationship between the difference of decoding delay $(\tau_0 - \tau_{-1})$ and the average reflection ratio (r) of all frames with the same difference from our measurements in Figure 9(b). When the decoding time difference is larger than 50ms (marked with a red arrow), the average reflection ratio is less than -0.95, indicating that most frames in this scenario are outliers. Therefore, the stationary controller in AFR does not calculate the



(a) Queue. (b) Bursty network arrival. (c) Stalled decoder service.

Figure 10: Differences between bursty network arrivals and stalled decoder services. The y-axis is the accumulated enqueue/dequeue frames. For example, the enqueue curve in Figure 10(b) increases from 1 to 2 at 1ms, indicating that frame #2 enqueues at 1ms.



(a) α-τ_Q mapping. (b) L_Q-τ_Q frequencies.

Figure 11: Illustrations and measurements of the transient controller. A series of linearly distributed dark blue clusters in Figure 11(b) indicate that L_Q and τ_Q are linearly correlated.

frames with a decoding delay difference larger than 50ms.

Convergence time analysis. To help operators to better understand the behavior of the stationary controller, we investigate the convergence of the stationary controller during state transitions of the service process. We want to answer the following question: During the transition from stationary state (μ_1,σ_1) to (μ_2,σ_2), how long will the stationary controller take to converge to the new frame-rate and drain the potential accumulation of the queue due to the transition?

We outline the main conclusion here and leave the detailed analysis in Appendix E. When the control loop (round-trip delay) of AFR is τ frames, the convergence time T_0 is bounded w.r.t. τ and W_0, and is acceptable for most scenarios. For example, when the average control loop of AFR is the interarrival time of one frame (τ=1), and W_0=2ms, the stationary controller could converge to the new stationary state within 2 frames. We illustrate the convergence time of the stationary controllers with more settings in Appendix E.

4.3 Transient Controller

The transient controller is designed to handle the contingent queue accumulations (§4.1). Therefore, we need to first understand how we should react to these queue contingencies.

Understanding queue contingencies. As shown in Figure 8(a) and 8(b), both stalled decoder services and bursty network arrivals will cause a sudden increase in queue length. We illustrate the enqueue and dequeue events of two contingencies in Figure 10. In Figure 10(b), 5 frames arrive at the client together within 4ms, resulting in a queue length of 4 when the 5th frame arrives and observes, as illustrated with the L_Q (blue arrow). In Figure 10(c), the decoder takes 80ms to decode the 0th frame, when queued frames cannot be dequeued to the decoder. Therefore, upon the arrival of the 5th frame, it also observes a queue length of 4.

However, the bursty network arrivals and stalled decoder services should be handled separately. In the scenario of bursty network arrivals, the bottleneck of total delay is still in the network due to its long network delay. As long as the decoder is functional, even if multiple frames arrive at the queue simultaneously, they could be processed efficiently (Figure 8(b)). In this case, the queue will be drained in a short time, and we do not need to reduce the frame rate. In contrast, the stalled decoder service will drastically increase the queuing delay of subsequent frames and needs adaption (Figure 8(a)). Thus, we need to differentiate between the two scenarios.

Since both scenarios result in an increase in queue length, they cannot be effectively differentiated with queue length only. Our insight is that we can differentiate them with the sojourn time of the first frame in the queue. As shown in Figure 10(a), at the arrival of frame K_2, the sojourn time τ_Q of the first frame K_1 and queue length L_Q observed by K_2 are:

$$\tau_Q = t_{enq}^{(K_2)} - t_{enq}^{(K_1)}, \qquad L_Q = K_2 - K_1 \tag{6}$$

where $t_{enq}^{(i)}$ is the enqueue timestamp of frame #i, and frame #K_1 is the frame at the head of the queue. For bursty network arrivals, since frames arrive at the decoder queue simultaneously, when the last frame of the burst arrives, the first frame has only been queued for a short time. For example, τ_Q in Figure 10(b) is 4ms (marked red). In contrast, for stalled decoder service, the head frame has been blocked for a long time, leading to a high τ_Q of 66ms in Figure 10(c). Therefore, we use τ_Q to adjust the frame rate in the transient controller.

Feedback control. For the transient controller, the design space is to find out a mapping between the discounting factor α and the queuing delay τ_Q. Since the transient controller is designed to reduce the frame rate based on the results of the stationary controller, the possible range of α satisfies:

$$f_{min}/f_{max} = \alpha_{min} \leqslant \alpha \leqslant 1 \tag{7}$$

where f_{min} and f_{max} are the lower and upper bounds for frame rate required by the application. Since longer τ_Q indicates a more severe load of the queue, the discounting factor should decrease with the increase of τ_Q. Besides, the α-τ_Q mapping should also have the following properties:

First, avoid overreactions. As we discussed above, for bursty network arrivals, τ_Q will also slightly increase due to the volumetric arrived frames. However, since such a transient queue accumulation will be cleared quickly as long as the decoder is functional (Figure 10(b)), we should not decrease the frame rate. Therefore, we need to introduce an upper reservoir (as shown in Figure 11(a)) to avoid overreactions. In the upper reservoir, when a non-zero but small τ_Q is observed ($0 \leqslant \tau_Q \leqslant Q_1$), the transient controller will not decrease the frame rate. The reservoir threshold Q_1 should be set based on measurements. We measure the observed L_Q and τ_Q from frames and present the results in Figure 11(b). Peaks near the left axis (marked by red dashed

arrows) represent frames with a long L_Q yet with a short τ_Q, which are due to the bursty network arrivals. Therefore, we set Q_1 to filter out those bursty arrival-related peaks (e.g., Q_1=14ms in our deployment, the red line in Figure 11(b)).

Second, respond timely. Due to the stringent delay requirements of high-quality RTC applications, a long queuing delay will drastically degrade the users' experiences. Therefore, we need to control the slope of the mapping in Figure 11(a) to effectively reduce the queuing delay. Since α is lower bounded, we could control the slope of the mapping by introducing a *lower reservoir*, as shown in Figure 11(a). We set Q_2 as the maximum tolerable queuing delay:

$$Q_2 = \max(Q_1, Deadline - \tau_{network} - \tau_{decode}) \quad (8)$$

where $\tau_{network}$ is the round-trip network delay, and τ_{decode} is the decoding delay μ_s. *Deadline* is the requirement for the total delay of the application. Based on users' experiences in the human-machine interaction and our operational experiences, we set *Deadline* to 100ms in our deployments [43].

5 Implementation

We implement the AFR with a frame-level trace-driven simulator, and deploy the AFR onto a production high-quality RTC service in the wild. In this section, we present the design of our simulator (§5.1), introduce the simulation setup (§5.2) and the deployment setup (§5.3).

5.1 Simulator Design

To faithfully compare and replay the traces for different queue control algorithms, we design a simple simulation environment that models the dynamics of RTC. The simulator maintains the decoder queue and replays the traces collected from online services, where the traces contain the decoding delay, network delay, original queuing delay, and also the arrival timestamp for each frame. Specifically, frames arrive at the decoder queue according to timestamps in traces, wait in the decoder queue for dequeuing, and are decoded according to decoding delays in traces. To avoid frequently sending frame-rate adjustment requests to the servers, frame rates are quantized at the level of 5fps, which is also followed by our online deployment. We implement the potential interference from CPU time-slicing: since the fetching of frames to decoders depends on the CPU, there are possible cases where fetching the frame from the queue to the decoder needs waiting to be scheduled by the CPU by up to several milliseconds [20]. Therefore, we further profile such a delay in the traces and introduce the scheduling waiting time in our simulator. We also implement the response time of the encoder between the new frame-rate actions and new frames generated with the updated frame rate, according to our measurements in §6.4. Please refer to Appendix C for implementation details.

5.2 Simulation Setup

Traces. We measure the frame-level statistics of our cloud gaming service (introduced in §3.1) on two types of clients (Windows and MacOS) and access networks (Ethernet and WiFi). We profile each step of received frames in one of our production clusters for 24 days in December 2020. This results in a dataset with 7.73 billion frames and 41.7k hours of playtime (Table 1), which is the largest frame-level dataset for interactive streaming to the best of our knowledge.

	Category	Session	Frame	Playtime
(1)	Windows+Ethernet	29.7k	6.35 B	34.2k hours
(2)	Windows+WiFi	6.4k	1.12 B	6.2k hours
(3)	MacOS+Ethernet	0.4k	40.9 M	0.2k hours
(4)	MacOS+WiFi	2.1k	216 M	1.1k hours
	Total	38.1k	7.73 B	41.7k hours

Table 1: Distribution of our traces on the client type.

Parameter settings. There are several parameters in AFR to be determined. Except for the parameters related to the transient controller (§4.3), we set W_0 in the stationary controller to 2ms and the discounting factors in EWMA $\xi_{arrv} = 0.033$ and $\xi_{serv} = 0.25$. We discuss the sensitivity of those settings and their influence on the performance in §6.3.

Metrics. In the evaluation, we mostly measure the delays (including the queueing delay and the end-to-end total delay). As we discussed in §2, the delay in interactive streaming is orthogonal to other video quality metrics (e.g., PSNR [2] or SSIM [76]). The delay, which represents interactivity, is the main optimization goal in this paper. We demonstrate that AFR has negligible degradation on the video quality in §6.4.

Baselines. To evaluate the performance of AFR, we implement existing frame control mechanisms as follows:

- `DropTail` is the frame control mechanism in WebRTC [60]. When frames overflow the queue, the client will first clear the queue, then request a new key frame, and finally drop all frames until the next key frame arrives. We set the queue capacity to 16 frames.
- `QLen-S` observes the current queue length, skips frames from the content generator before the encoder if the queue length is $\geqslant$1, and resumes if the queue length is $<$1.
- `QWait-S`. We migrate the frame control mechanisms from existing academic efforts in our simulator [34, 77], and replace the signal from total delay to queuing delay to better reduce the queuing delay. Since these baselines are not designed for stringent delay requirements of 100ms, we also finetune their parameters with our traces. `QWait-S` skips frames before the encoder if the queuing delay is $\geqslant$32ms, and resumes if the queuing delay is $<$4ms.

Besides, to evaluate the effectiveness of different components in AFR, we also different variants of AFR:

- `AFR-QLen`. We demonstrate the insufficiency of controlling the frame rate with queue states with a feedback algorithm based on current queue length: it observes the current queue length at the arrival of each frame, and maps the queue length of$\{0, 1+\}$ to frame-rate$\{60, 24\}$fps.
- `AFR-QWait`. A feedback algorithm maps current queuing delay of$\{(0, 4), (4, 8), (8, 12), (12, \infty)\}$ms to frame-rate of $\{60, 48, 36, 24\}$fps. The parameters have also been finetuned with our traces.

- `AFR-TX`. To demonstrate the effectiveness of measuring both the arrival and service process, we further implement a dequeue rate-based algorithm. `AFR-TX` measures the dequeue rate and sets the target frame-rate with $\rho = 0.8$, where ρ has been tuned with our traces. The dequeue rate is the reciprocal of decoding delay.
- `AFR-Kingman`. Moreover, we individually evaluate the stationary controller of AFR to further illustrate the effectiveness of the transient controller.
- `AFR`. Finally, we put all optimizations in this paper (both the stationary and transient controller) together.

We present how we tune the parameters, and evaluate the trade-offs between frame rate and queuing delay in §6.3.

5.3 Deployment setup

We finally deploy AFR onto our cloud gaming service. The gaming service X employs the H.264 codec to increase the coverage of hardware decoding and adaption towards heterogeneous clients[1], and customizes the codec performance for the optimization of gaming. Tencent Start currently supports 13 production-level games, including action-adventure, first-person shooter, and real-time strategy games. To optimize the network delay, the service is accelerated with multi-access edge computing similar to [74, 57, 86]: Users are split into tens of operation regions with a geographical diameter of hundreds of kilometers. Cloud gaming servers are deployed on clusters in each operation region, resulting in an average round-trip network delay of 15ms (Appendix B.2).

The frame-rate adaption algorithms are implemented on the client side. The AFR controller continuously measures the statistics of the decoder queue, and sends requests to edge servers to adjust the frame rate when necessary. The edge server then forwards the frame-rate adjustment requests to both the video encoder and the gaming application. New frames will be generated following the new inter-frame interval. We evaluate the response timeliness and overhead of video encoder and gaming application in §6.4.

6 Evaluation

We evaluate the AFR controller in the following aspects:

- **Delay improvements.** We present the performance improvements: The ratio of frames with long queuing delay and total delay of AFR has been improved by 2.1×-26× and 13%-2.2× against existing baselines (§6.1).
- **Frame-rate maintenance.** We then demonstrate that AFR introduces negligible impacts on the metrics related to frame-rate (§6.2).
- **Parameter sensitivity.** Our evaluation shows that parameters in AFR have a wide range of settings to gain performance improvements against finetuned baselines (§6.3).
- **Microbenchmarking.** We further demonstrate that the timeliness, overhead, and image quality of frame-rate adjustments are satisfactory for online deployment (§6.4).

[1]Hardware decoding has a shorter decoding delay than software decoding and supports higher frame rates. H.264 has a higher coverage of hardware decoding support compared to other advanced codecs [54].

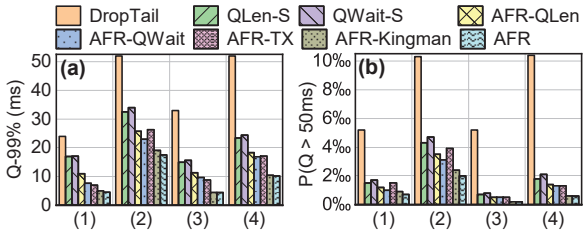


Figure 12: Simulation results of queuing delay (the 99%ile and the ratio of frames with >50ms queuing delay).

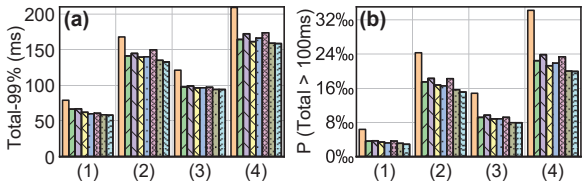
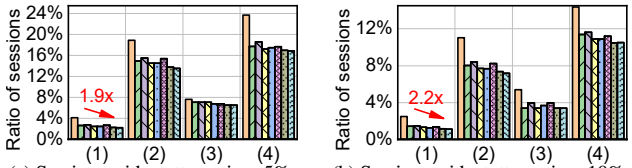


Figure 13: Simulation results of total delay (the 99%ile and the ratio of frames with >100ms total delay).



(a) Sessions with stutter ratio >5%. (b) Sessions with stutter ratio >10%.

Figure 14: Ratio of sessions with different stuttered frames.

- **Deployment in the wild.** Finally, we report the A/B test results and the deployment progress of AFR on our cloud gaming service online (§6.5).

6.1 Delay Improvements

We compare the queuing delay and the total delay of each frame with AFR and baseline algorithms in four sets of traces (Table 1). We measure the queuing delay in two dimensions: we present the 99th percentile queuing delay and the ratio of frames with a queuing delay >50ms in Figure 12. We first analyze the results of AFR against three existing mechanisms (`DropTail`, `QLen-S`, and `QWait-S`). AFR could reduce the 99%ile queuing delay by 1.9× to 7.4×, and the ratio of severely queued frames by 2.1× to 26× on different sets of traces against three baselines. In this case, the 99%ile queuing delay could be squeezed to 6.9ms. This indicates that AFR could effectively achieve an ultra-short queuing delay. AFR also demonstrates satisfactory performance improvements on the *total end-to-end delay*, which is directly related to users' experiences. AFR improves the 99%ile total delay by 27% to 36%, and the ratio of severely delayed frames (total delay >100ms) by 1.6× to 2.2× in all traces. We also measure the *session stutter ratio*, i.e. the ratio of frames with a total delay of >100ms in a session, for each session. We then measure the ratio of sessions with a session stutter ratio of >5% and >10%, which indicates how many users suffer from unsatisfactory experiences and present the results in Figure 14. For the major population of our service (Cat. (1), Table 1), AFR reduces the stuttered sessions by 17% and 21% compared to the best of the three baselines. For other categories, the ratio of

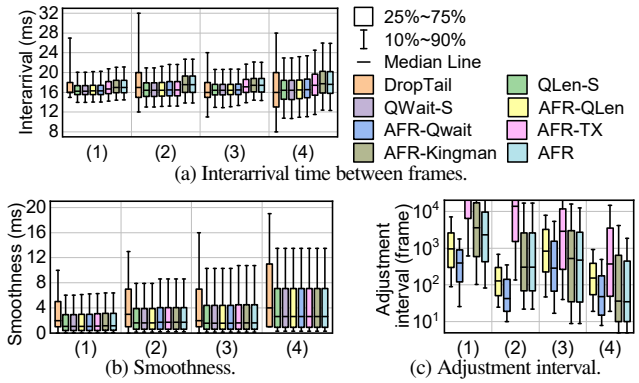


Figure 15: Frame-rate maintenance. Better viewed in color.

stutter sessions has also been reduced by 5% to 37%. AFR could significantly improve experiences for high-quality RTC.

We further understand the performance improvements with the comparisons among different variants of AFR. Compared to `DropTail`, baselines based on queue states (`AFR-QLen`, `AFR-QWait`) could effectively reduce the queuing delay, indicating the necessity of actively controlling the queuing delay (§3.1). Compared to `QLen-S` and `QWait-S`, controlling the frame rate achieves better performance than skipping frames from the encoder. This is because skipping frames would drastically degrade the tail frame rate, for which the parameters of baselines are tuned (§6.3). `AFR-TX` could further reduce the queuing delay than the queue state-based baselines, indicating that observing the service process could know the potential degradation in advance and effectively take actions, validating our analysis in §3.3. `AFR-Kingman` further improves the performance by 10% against `AFR-TX`, demonstrating that the fluctuating arrival of the high-quality RTC could also affect the estimation of the decoder queue. `AFR` finally reduces the tail queuing delay by 2-4% against `AFR-Kingman`, indicating the necessity of the transient controller to handle contingencies.

Besides, we also find that AFR has larger performance improvements when the network is better. The performance improvements on two sets of Ethernet traces (55% and 37% for Cat. (1) and (3)) are larger than the on WiFi traces (35% and 27% for Cat. (2) and Cat. (4)). Considering the ongoing deployment of next-generation access networks with better network conditions (e.g., 5G and WiFi 6), the necessity of controlling the decoder queue would be more significant.

6.2 Frame-rate Maintenance

Besides, we also measure the effect of AFR on the frame rate. We first measure the interarrival time between frames at the arrival of each frame on the client. For example, a frame rate of 60fps should result in an interarrival time of around 16.7ms. We tune the parameters of each algorithm to keep the 99th percentile of their interarrival time at the same level (details in §6.3). Therefore, for 10-90th percentiles, as shown in Figure 15(a), most algorithms except for `DropTail` are comparable. Compared to the existing deployed mechanism `DropTail`, AFR even improves the tail user-perceived frame rate due to its better management of frame

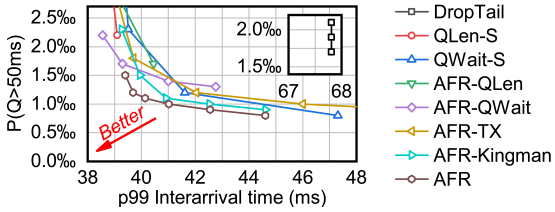


Figure 16: The trade-off between the tail interarrival time and queuing delay. We tune the parameters for baselines and AFR to illustrate the capability of each algorithm in the trade-off.

drops. AFR slightly decreases the median frame rate by 3%-9%, which brings the negligible quality of experience (QoE) degradation to users considering the improvements on delay [71, 81].

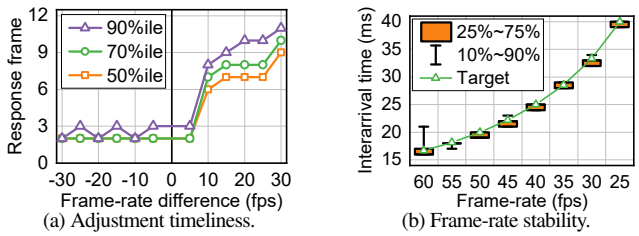
We further measure the smoothness of frame-rate, which might also have potential effects on users' experiences [33]. We measure the *differences of interarrival time* as an indicator of the smoothness of frame rate and present the results in Figure 15(b). Except for `DropTail`, all baselines and AFR have similar interarrival differences and are better than `DropTail`. This is mainly because that frame drops in `DropTail` will introduce a sudden increase of interarrival differences. Moreover, we also measure the frame adjustment interval and present the distributions in Figure 15(c). The median adjustment interval of AFR is hundreds to thousands of frames, which is much longer than the response time of frame-rate adjustment (§6.4).

6.3 Parameter Sensitivity

We then evaluate the sensitivity of parameters in AFR and other baselines. We tune parameters of all baselines in §5.2: thresholds for skipping frames for `QLen-S` and `QWait-S`, mappings for `AFR-QLen` and `AFR-QWait`, ρ for `AFR-TX`, and W_0 for `AFR-Kingman` and AFR. We present the ratio of frames with queuing delay >50ms (P(Q>50ms)) and the 99th percentile of interarrival time on Cat. (1) traces in Figure 16. The down-left corner indicates the algorithm has a satisfactory trade-off between the queuing delay and the frame rate.

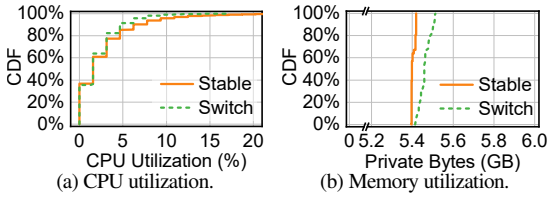
As we can see, AFR outperforms all other baselines in a wide range of settings, achieving a better trade-off between the queuing delay and frame rate. `QLen`-based algorithms are challenged in achieving ultra-short queuing delay: with the extremest parameters (skipping/decreasing frame-rate as long as queue length is non-zero), `QLen-S` and `AFR-QLen` could only achieve a P(Q>50ms) of 2.2‰ and 1.7‰, much higher than other baselines. This follows our analysis in §3.3 that queue length is too coarse-grained as a signal to control the queue with an ultra-short target. Meanwhile, skip-based algorithms could achieve lower queuing delay compared to frame-rate-based algorithms, yet with higher interarrival time. The parameters of all algorithms are tuned according to Figure 16 by aligning the 99th percentile interarrival time.

We also evaluate how different percentiles of queuing delay and total delay are affected by the setting of W_0 in Appendix D.3. The performance of AFR reacts sensitively to the setting of W_0, indicating that operators could effectively balance the total delay and frame rate by adjusting W_0. We further evaluate



(a) Adjustment timeliness. (b) Frame-rate stability.

Figure 17: Effectiveness of frame-rate adjustment.



(a) CPU utilization. (b) Memory utilization.

Figure 18: Frame-rate adjustment overhead.

the sensitivity of the discounting factors ξ of the EWMA and EWMV in the transient controller (§4.3) in Appendix D.3, demonstrating how operators should set these parameters to balance between the precision and sensitivity.

6.4 Microbenchmarking

We also benchmark AFR in a testbed of our cloud gaming service.

Effectiveness of frame-rate adjustment. We first measure the responsiveness and precision of frame-rate adjustment at the video encoder. We enumerate all frame-rate switching within{25, 30, ···, 60}fps, and measure how many frames the encoder needs to take to steadily output video streams at the new frame rate. The response time measured by the unit of frame (i.e. *response frame*) is presented in Figure 17(a). For each group of settings, we repeat the experiments 100 times to eliminate the randomness. When decreasing the frame rate, the 90%ile response frames is less than 3 frames, indicating the encoder and gaming application could decrease the frame-rate timely. This could effectively alleviate the overload of the decoder queue. When significantly increasing the frame rate, the frame rate might be slightly delayed to change. This is because the frame rate at the client side follows the bucket effect. Either encoder or the gaming application decreases the frame rate will lead to a decrease of the final frame rate, while the increase of frame rate needs an increase from both components. Even so, the tail response frame is $<$10 frames, which is much less than the adjustment interval (Figure 15(c)).

We then measure the fluctuation of the frame rate of the output of the streaming encoder. We set the frame rate to several levels as above, and measure the interarrival time between each frame. For each frame rate, we measure the interarrival time for 30,000 frames and present the distribution in Figure 17(b). The interarrival time between frames largely falls around the target frame rate. Therefore, unlike the fluctuating bit-rates in video streaming [42], frame-rate could be precisely controlled by the encoder.

Frame-rate adjustment overhead. We further measure the potential processing overhead of frame-rate adjustment at the edge server. To magnify the overhead, we change the frame rate

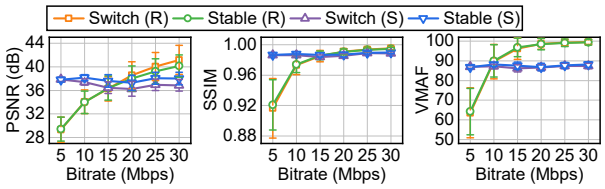


Figure 19: The image quality differences of AFR and the original video tested in a running scene (R) and stable scene (S). The error bar represents the standard deviation.

from 60fps to 30fps and back to 60fps every 6 frames, which is much shorter than the usual adjustment interval. We then measure the CPU and memory utilization of the cloud gaming application and encoder by sampling the CPU processing time and application private bytes with the `typeperf` [66] every 1 second. We measure for 30 minutes to eliminate the randomness. We compare the scenario with a stable frame-rate of 60fps (`stable`) and a frequently switching frame-rate (`switch`) in Figure 18. For CPU utilization, both scenarios have a similar distribution from 0% to 20%. `switch` is a little better than `stable` since producing a lower frame rate takes fewer CPU resources for the gaming application. As for memory utilization, the major memory consumption is from the gaming application. Frame-rate switching slightly increases the utilization of private bytes since frequently resetting the encoder requires allocation of memory. Nonetheless, the increase of memory utilization is less than 1.8% even at the 99%ile, which is negligible and could be even lower in the case of normal frame-rate adjustments.

Image quality degradation. We also investigate the potential image quality degradation caused by AFR. We record two raw videos from games, one in a running scene (R) and another in a standing scene (S). For each video, we switch the frame rate every 100 frames 15 times and measure the video quality for the following 400 frames. We investigate three video quality metrics, peak-signal-to-noise-ratio (PSNR) [2], structural similarity index (SSIM) [76], and video multimethod assessment fusion (VMAF) [51], and present the results in Figure 19. `stable` and `switch` denote the scenarios where the frame-rate remains unchanged or frequently switched. Results demonstrate that frequently switching the frame rate will not affect the video quality: the video quality of two videos on three metrics are comparable in all cases.

6.5 Deployment in the Wild

Finally, we evaluate the performance of AFR by deploying it onto Windows clients of our cloud gaming service, Tencent Start, in one of its production clusters. Before the deployment of AFR, our cloud gaming service follows the frame control strategy in WebRTC (i.e., `DropTail`). To make a clean and controlled comparison, we only present the results from online A/B tests in our production clusters, when all other implementations and settings are kept the same. The A/B test is conducted from January 8, 2021, to January 14, 2021, resulting in 5369 Ethernet sessions and 1467 WiFi sessions. The parameter settings of AFR remain the same as the simulation (§5.2). We randomly enable

Cat. (1)	Q99	Q>50ms	T99	T>100ms	Session
DropTail	54ms	1.11%	101ms	1.03%	7.30%
AFR	22ms	0.51%	80ms	0.68%	5.82%
Cat. (2)	Q99	Q>50ms	T99	T>100ms	Session
DropTail	64ms	1.83%	174ms	3.00%	24.00%
AFR	37ms	0.54%	160ms	2.11%	21.17%

Table 2: Performance of deployment in the wild. Metrics are the 99%ile of queuing delay (Q99), the ratio of frames with Q>50ms, the 99%ile of total delay (T99), and the ratio of the stuttered frame (T>100ms). `Session` is the ratio of sessions with stutter ratio >5%. Cat. (1) and (2) are Ethernet and WiFi on Windows clients.

(or disable) AFR with a probability of 50% for each session, and present the results in Table 2. Similar to the simulation results, the ratio of stuttered frames measured by total delay (P(T>100ms)) in both categories has been improved by 34% and 30%, which significantly improves users' experiences in interactive streaming. The stuttered sessions (with the same metric as Figure 14(a)) have also been reduced by 17% on average, indicating these users could be alleviated from stuttering streaming experiences. Therefore, the online deployment also demonstrates significant benefits of AFR for high-quality RTC users. AFR has already been deployed onto all production clusters of Tencent Start for over one year, serving thousands of users each day.

7 Discussions

In this section, we discuss the potential limitations of AFR.

Application scenarios. In this paper, we mainly evaluate the performance of AFR on traces or production clusters of our cloud gaming service. However, as we introduce in §1 and §2, the overload of decoder queue generally exists in many high-quality RTC scenarios, such as VR streaming or 4K live streaming, as long as they stream high frame-rate and high bit-rate video onto commercial clients. We evaluate AFR with cloud gaming due to access to the real-world traces and production services X. We leave the deployment of AFR over other scenarios as our future work.

Coexistence of multiple control loops. There are other control loops that work simultaneously in the RTC system. For example, the underlying congestion controller will also control the bit-rate of the video based on network conditions [25]. The video codec will also adjust the quantization parameter based on the scenes to encode [17]. As we discussed in §3.2, these parameters are affected by different causes (network congestion, decoder degradation, scene variation), which are orthogonal to each other. Therefore, the adaption of the frame rate is orthogonal to the other controllers in the RTC system. In §6.5, we evaluate the performance of AFR with all these controllers in our real production in the wild. We leave the coordination of different controllers on the joint optimization over the user's experience for the future.

8 Related Work

There has been little prior work on the decoder queue for high-quality RTC. We survey the following three aspects.

Frame controls in RTC. As we discussed, besides the `DropTail` mechanism in WebRTC, there are a series of research efforts in the active control of RTC frames. For example, some work [38, 77, 34] maintains a certain number of in-flight frames based on total delay [77] or frame-level acknowledgement mechanisms [34]. AFR differentiates from them in two aspects. First, the end-to-end control introduces ambiguity in taking effective actions, as discussed in §3.2. AFR takes effective actions to reduce the queuing delay. Second, existing control strategies are based on queue lengths or queuing delay. In contrast, measuring the arrival and service processes in AFR could help high-quality RTC to achieve lower queuing delays. Furthermore, researchers also proposed to co-design the codec and network [35, 34, 79, 75] or even redesign new decoding ASICs [90]. These cannot be accelerated with commercial hardware and are hard to deploy in practice.

Adaptive Bit-rate Control. There have already been a series of research efforts on the optimization of low-latency streaming. Different congestion control [78, 83, 25] or rate adaption algorithms [87, 84] have been proposed to enable the low-latency transport for real-time communications. However, as we discussed above, changing the bit rate without changing the frame rate will not alleviate the load of the decoder queue. Since bit-rate and frame-rate are independent in video streaming, bit-rate adaption is orthogonal to frame-rate adaption and could be integrated with AFR to control the network delay and queuing delay together.

Cloud gaming. As a recently emerging application, cloud gaming also attracts the attention of researchers. Researches include the optimization of the renderer [23, 52, 27] and streaming codec [72], which are independent of the optimization at the transport level. There are also investigations on the user experience of cloud gaming [81, 71, 28], which could be integrated with our work by better customizing the optimization goal. Recent efforts also try to investigate the performance of production-level cloud gaming services from the client side [26], which are limited in scale and completeness. To the best of our knowledge, we are also the first piece of work to investigate the performance of production-level cloud gaming services from the server side, with the scale of tens of thousands of hours of playtime.

9 Conclusion

In this paper, we propose AFR to reduce the queuing delay of the decoder queue for high-quality RTC by dynamically adjusting the frame rate. AFR introduces a stationary controller and a transient controller to respectively mitigate the stationary heavy traffic and contingent arrivals and services. We further evaluate the performance of AFR with trace-driven simulations and deployments in the production clusters. Experiments demonstrate that AFR could significantly reduce the stuttering ratio and tail total delay.

This work does not raise any ethical issues.

Acknowledgements. We sincerely thank our shepherd KyoungSoo Park, anonymous reviewers, and labmates in the Routing Group from Tsinghua University for their valuable feedback. This work is sponsored by the National Natural Science Foundation of China (No. 62002196, 61832013, and 62221003) and the Tsinghua-Tencent Collaborative Grant. Bo Wang and Mingwei Xu are the corresponding authors.

References

[1] Cloud gaming (beta) with xbox game pass — xbox. https://www.xbox.com/en-US/xbox-game-pass/cloud-gaming, 2020.

[2] Peak signal-to-noise ratio - wikipedia. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio, 2020.

[3] Stadia - one place for all the ways we play. https://stadia.google.com/, 2020.

[4] Start - tencent cloud gaming. https://start.qq.com/, 2020.

[5] Your games. your devices. play anywhere — nvidia geforce now. https://www.nvidia.com/en-us/geforce-now/, 2020.

[6] Facebook 360 video. https://facebook360.fb.com/, 2021.

[7] Google meet and default video resolution - google meet community. https://support.google.com/meet/thread/58039897/google-meet-and-default-video-resolution, 2021.

[8] Huawei video conferencing platform — huawei enterprise. https://e.huawei.com/en/solutions/enterprise-collaboration/videoconferencing-platform, 2021.

[9] Meeting and phone statistics – zoom help center. https://support.zoom.us/hc/en-us/articles/202920719-Meeting-and-phone-statistics, 2021.

[10] Trtx 2080 ti vs rtx 3080 ti game performance benchmarks (i7-8700k vs core i9-10900k) - gpucheck united states / usa. https://www.gpucheck.com/compare/nvidia-geforce-rtx-2080-ti-vs-nvidia-geforce-rtx-3080-ti/, 2021.

[11] Vr-interactive – we are interactive. https://vr-interactive.at/, 2021.

[12] Youtube vr - home - youtube vr. https://vr.youtube.com/, 2021.

[13] Gfxbench - unified graphics benchmark based on dxbenchmark (directx) and glbenchmark (opengl es). https://gfxbench.com/result.jsp, 2022.

[14] multithreading - pin processor cpu isolation on windows - stack overflow. https://stackoverflow.com/questions/15324586/pin-processor-cpu-isolation-on-windows, 2022.

[15] Processor benchmarks - geekbench browser. https://browser.geekbench.com/processor-benchmarks/, 2022.

[16] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *Proc. USENIX NSDI*, 2018.

[17] Salahuddin Azad, Wei Song, and Dian Tjondronegoro. Bitrate modeling of scalable videos using quantization parameter, frame rate and spatial resolution. In *Proc. IEEE ICASSP*, pages 2334–2337, 2010.

[18] Richard Bellman and Robert Kalaba. On adaptive control processes. *IRE Transactions on Automatic Control*, 4(2):1–9, 1959.

[19] Ankita Bhutani and Preeti Wadhwani. Cloud gaming market share forecast 2025 — industry size report. https://www.gminsights.com/industry-analysis/cloud-gaming-market, 2020.

[20] Karl Bridge and Michael Satran. Multitasking - win32 apps — microsoft docs. https://docs.microsoft.com/en-us/windows/win32/procthread/multitasking, 2018.

[21] James Bruce, Marta Mrak, and Rajitha Weerakkody. Testing av1 and vvc - bbc r&d. https://www.bbc.co.uk/rd/blog/2019-05-av1-codec-streaming-processing-hevc-vvc, 2019.

[22] Alan Bryman and Duncan Cramer. *Quantitative data analysis with IBM SPSS 17, 18 & 19: A guide for social scientists*. Routledge, 2012.

[23] James Bulman and Peter Garraghan. A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines. In *Proc. USENIX HotCloud*, 2020.

[24] Ronald S. Bultje. The world's fastest vp9 decoder: ffvp9. https://blogs.gnome.org/rbultje/2014/02/22/the-worlds-fastest-vp9-decoder-ffvp9/, 2014.

[25] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Congestion control for web real-time communication. *IEEE/ACM Transactions on Networking*, 2017.

[26] Marc Carrascosa and Boris Bellalta. Cloud-gaming: Analysis of google stadia traffic. *arXiv:2009.09786*, 2020.

[27] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. T-gaming: A cost-efficient cloud gaming system at scale. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[28] Kuan-Ta Chen, Yu-Chun Chang, Hwai-Jung Hsu, De-Yu Chen, Chun-Ying Huang, and Cheng-Hsin Hsu. On the quality of service of cloud gaming systems. *IEEE Transactions on Multimedia*, 2013.

[29] Yushin Cho, William A Pearlman, and Amir Said. Low complexity resolution progressive image coding algorithm: progres (progressive resolution decompression). In *Proc. IEEE ICIP*, 2005.

[30] Tzu-Der Chuang, Pei-Kuei Tsung, Pin-Chih Lin, Lo-Mei Chang, Tsung-Chuan Ma, Yi-Hau Chen, and Liang-Gee Chen. A 59.5 mw scalable/multi-view video decoder chip for quad/3d full hdtv and video streaming applications. In *Proc. IEEE ISSCC*, pages 330–331, 2010.

[31] Harald Cramér. Mathematical methods of statistics, 1946. *Department of Mathematical SU*, 1946.

[32] Robert G. Gallager Dimitri P. Bertsekas. Section 3.3: The m/m/1 queuing system. In *Data Networks (2nd Edition)*, 1992.

[33] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *Proc. ACM SIGCOMM*, 2011.

[34] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *Proc. USENIX NSDI*, 2018.

[35] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. USENIX NSDI*, 2017.

[36] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. Abc: A simple explicit congestion controller for wireless networks. In *Proc. USENIX NSDI*, 2020.

[37] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proc. ACM SIGCOMM*. 2019.

[38] Jefferson Han and Brian Smith. Cu-seeme vr immersive desktop teleconferencing. In *Proc. ACM Multimedia*, pages 199–207, 1997.

[39] Refael Hassin and Moshe Haviv. *To queue or not to queue: Equilibrium behavior in queueing systems*, volume 59. Springer Science & Business Media, 2003.

[40] Petr Holub, Jiří Matela, Martin Pulec, and Martin Šrom. Ultragrid: low-latency high-quality video transmissions on commodity hardware. In *Proc. ACM Multimedia*, 2012.

[41] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. Gaminganywhere: an open cloud gaming system. In *Proc. ACM MMSys*, 2013.

[42] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proc. ACM SIGCOMM*, 2014.

[43] Zenja Ivkovic, Ian Stavness, Carl Gutwin, and Steven Sutcliffe. Quantifying and mitigating the negative effects of local latencies on aiming in 3d shooter games. In *Proc. ACM CHI*, pages 135–144, 2015.

[44] Van Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, 1988.

[45] JFC Kingman and MF Atiyah. The single server queue in heavy traffic. *Oper. Manage., Critical Perspect. Bus. Manage*, 2003.

[46] Marwan Krunz and Herman Hughes. A traffic for mpeg-coded vbr streams. In *Proc. ACM SIGMETRICS*, 1995.

[47] Ana Kuzmanic and Vlasta Zanchi. Hand shape classification using dtw and lcss as similarity measures for vision-based gesture recognition system. In *EUROCON 2007-The International Conference on" Computer as a Tool"*, pages 264–269. IEEE, 2007.

[48] Yun Gu Lee and Byung Cheol Song. An intra-frame rate control algorithm for ultralow delay h. 264/advanced video coding (avc). *IEEE Transactions on Circuits and Systems for Video Technology*, pages 747–752, 2009.

[49] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proc. ACM SIGCOMM*, 2020.

[50] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proc. ACM SIGCOMM*, 2019.

[51] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. Toward a practical perceptual video quality metric — netflix techblog. `https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652`, 2016.

[52] Xiaofei Liao, Li Lin, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. Liverender: A cloud gaming system based on compressed graphics streaming. *IEEE/ACM Transactions on Networking*, 2016.

[53] CC Lin, JI Guo, HC Chang, YC Yang, JW Chen, MC Tsai, and JS Wang. A 160kgate 4.5 kb skram h. 264 video decoder for hdtv applications. In *Proc. IEEE ISSCC*, pages 1596–1605, 2006.

[54] Candice Liu. Hardware decoding vs software decoding in 4k h264/h265 video. https://www.macxdvd.com/mac-video-converter-pro/hardware-decoding-4k-ultra-hd-video.htm, 2020.

[55] Andrea Lottarini, Alex Ramirez, Joel Coburn, Martha A Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler. vbench: Benchmarking video transcoding in the cloud. In *Proc. ASPLOS*, pages 797–809, 2018.

[56] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving Consistent Low Latency for Wireless Real Time Communications with the Shortest Control Loop. In *Proc. ACM SIGCOMM*, 2022.

[57] China Mobile and ZTE. Powered by sa: 5g mec-based cloud game innovation practice. GSMA 5G Case Studies (https://www.gsma.com/futurenetworks/wp-content/uploads/2020/03/Powered-by-SA-5G-MEC-Based-Cloud-Game-Innovation-Practice-.pdf), 2020.

[58] Omar Mossad, Khaled Diab, Ihab Amer, and Mohamed Hefeeda. Deepgame: Efficient video encoding for cloud gaming. In *Proc. ACM Multimedia*, 2021.

[59] Vit Niennattrakul and Chotirat Ann Ratanamahatana. On clustering multimedia time series data using k-means and dynamic time warping. In *2007 International Conference on Multimedia and Ubiquitous Engineering (MUE'07)*, pages 733–738. IEEE, 2007.

[60] Ilya Nikolaevskiy. Refactor framebuffer to store decoded frames history separately (i82be0eb3) · gerrit code review. https://webrtc-review.googlesource.com/c/src/+/116686, 2019.

[61] OPG609. List of 60fps games playable on ps5. https://www.reddit.com/r/PS5/comments/kiuh2t/list_of_60fps_games_playable_on_ps5/, 2020.

[62] Adrian Pennington. So you say you're planning a 16k live stream... - nab amplify. https://amplify.nabshow.com/articles/so-you-say-youre-planning-a-16k-live-stream/, 2022.

[63] Stefano Petrangeli, Viswanathan Swaminathan, Mohammad Hosseini, and Filip De Turck. An http/2-based adaptive streaming framework for 360 virtual reality videos. In *Proc. ACM Multimedia*, 2017.

[64] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. Improving mobile gaming performance through cooperative cpu-gpu thermal management. In *Proc. ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[65] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 1994.

[66] Elizabeth Ross, John Parente, Mike Jacobs, David Kuehn, John Baldwin, Corey Plett, Brock Mammen, and Liza Poggemeyer. typeperf — microsoft docs. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/typeperf, 2017.

[67] Saeed Shafiee Sabet, Steven Schmidt, Saman Zadtootaghaj, Babak Naderi, Carsten Griwodz, and Sebastian Möller. A latency compensation technique based on game characteristics to mitigate the influence of delay on cloud gaming quality of experience. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 15–25, 2020.

[68] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. IETF RFC 6298.

[69] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 2007.

[70] Arun Kumar Sharma. *Text book of correlations and regression*. Discovery Publishing House, 2005.

[71] Ivan Slivar, Lea Skorin-Kapov, and Mirko Suznjevic. Cloud gaming qoe models for deriving video encoding adaptation strategies. In *Proc. ACM MMSys*, 2016.

[72] Ivan Slivar, Mirko Suznjevic, and Lea Skorin-Kapov. The impact of video encoding parameters and game type on qoe for cloud gaming: A case study using the steam platform. In *Proc. IEEE International Conference on Quality of Multimedia Experience (QoMEX)*, 2015.

[73] James Stringer. Pushing it to the limit – parsec at 240 frames per second with approximately 4-8 milliseconds of ... — parsec. https://parsec.app/blog/parsec-game-streaming-total-latency-at-240-frames-per-second-c0818cc0daa5, 2022.

[74] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Supporting mobile vr in lte networks: How close are we? *Proc. ACM SIGMETRICS*, 2018.

[75] Tingfeng Wang, Zili Meng, Mingwei Xu, Rui Han, and Honghao Liu. Enabling high frame-rate uhd real-time communication with frame-skipping. In *Proc. ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2021.

[76] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004.

[77] Keith Winstein and Hari Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *Proc. USENIX ATC*, 2012.

[78] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proc. USENIX NSDI*, 2013.

[79] Jiyan Wu, Chau Yuen, Ngai-Man Cheung, Junliang Chen, and Chang Wen Chen. Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 2015.

[80] Gang Yi, Dan Yang, Abdelhak Bentaleb, Weihua Li, Yi Li, Kai Zheng, Jiangchuan Liu, Wei Tsang Ooi, and Yong Cui. The acm multimedia 2019 live video streaming grand challenge. In *Proc. ACM Multimedia*, pages 2622–2626, 2019.

[81] Saman Zadtootaghaj, Steven Schmidt, and Sebastian Möller. Modeling gaming qoe: Towards the impact of frame rate and bit rate on cloud gaming. In *Proc. IEEE International Conference on Quality of Multimedia Experience (QoMEX)*, 2018.

[82] Saman Zadtootaghaj, Steven Schmidt, Saeed Shafiee Sabet, Sebastian Möller, and Carsten Griwodz. Quality estimation models for gaming video streaming services using perceptual video quality dimensions. In *Proc. ACM Multimedia Systems Conference (MMSys)*, 2020.

[83] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proc. ACM SIGCOMM*, 2015.

[84] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. Onrl: improving mobile video telephony via online reinforcement learning. In *Proc. ACM MobiCom*, 2020.

[85] Wenxiao Zhang, Feng Qian, Bo Han, and Pan Hui. Deepvista: 16k panoramic cinema on your mobile device. In *Proceedings of the Web Conference*, pages 2232–2244, 2021.

[86] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 2019.

[87] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. Learning to coordinate video codec with transport protocol for mobile video telephony. In *Proc. ACM MobiCom*, 2019.

[88] Chao Zhou, Mengbai Xiao, and Yao Liu. Clustile: Toward minimizing bandwidth in 360-degree video streaming. In *Proc. IEEE INFOCOM*, 2018.

[89] Dajiang Zhou, Shihao Wang, Heming Sun, Jianbin Zhou, Jiayi Zhu, Yijin Zhao, Jinjia Zhou, Shuping Zhang, Shinji Kimura, Takeshi Yoshimura, et al. An 8k h. 265/hevc video decoder chip with a new system pipeline design. *IEEE Journal of Solid-State Circuits*, 52(1):113–126, 2017.

[90] Dajiang Zhou, Jinjia Zhou, Xun He, Jiayi Zhu, Ji Kong, Peilin Liu, and Satoshi Goto. A 530 mpixels/s 4096x2160@ 60fps h. 264/avc high profile video decoder chip. *IEEE Journal of Solid-State Circuits*, 46(4):777–788, 2011.

[91] Dajiang Zhou, Jinjia Zhou, Jiayi Zhu, Peilin Liu, and Satoshi Goto. A 2gpixel/s h. 264/avc hp/mvc video decoder chip for super hi-vision and 3dtv/ftv applications. In *Proc. IEEE International Solid-State Circuits Conference*, pages 224–226, 2012.

Appendices

A Potential Solutions and Concerns

In this section, we discuss why other potential solutions are insufficient to address the problem in this paper, and discuss other concerns of adapting the frame rate during runtime.

A.1 Potential Solutions

Discarding frames or adjusting resolutions. For most widely adopted codecs, dropping one frame or changing the resolution will make the following frames fail to recover the raw pixels of the block because they are differentially encoded by the motion vector to the previous one[2]. This is to utilize the redundant information between frames to reduce the bitrate of the stream. Since key frames do not rely on previous frames, they are usually much larger than other predictive frames (sometimes 10×) [46]. Therefore, given the same bottleneck bandwidth, sending a frame with 10× larger size will take approximately 10× time (tens to hundreds of milliseconds), which drastically increases the delay for the users. Moreover, frequently requesting key frames will degrade the goodput of the streaming and potentially increase the congestion in the network. Therefore, directly dropping delayed frames at the client or frequently changing the resolution will introduce stalls for the subsequent frames and degrade the users' experiences of high-quality RTC.

Adjusting the bit-rate. Without changing the resolution and frame rate, adjusting the bit rate has a very limited effect in reducing the decoding delay. Generally speaking, resolution, bit rate, and frame rate could be independently set. The display resolution describes the number of distinct pixels in each dimension that

[2]Recent advances on scalable video coding could partially break the inter-frame dependency, yet degrades video quality with the same bit-rate [69].

can be displayed, and the frame rate represents the number of pictures within one second of video. And the bit rate represents the amount of data used for storing the coded bit-stream. So the higher resolution we set, the more pixels a single picture will have, which could mean a higher definition of the video. And setting a higher frame rate means there will be more pictures per video second to make the video smoother. If we set a higher target bit-rate while keeping other parameters unchanged, the encoder can use more data to represent the pictures to achieve lower possible image distortion with a lower *quantization parameter* [17].

In this case, with the unchanged frame rate and resolution, the decoding procedure is also unaffected. For example, in H.264/AVC, a sequence of macroblocks can be composed of a slice, a picture, therefore, is a collection of one or more slices. Slices are completely independent of each other, and the macroblocks inside a video frame can be reconstructed in parallel. The video decoding has been parallelized using slice-level or block-level parallelism. The resolution will affect how many pixels there are in one frame, and the frame rate determines the tolerable decoding delay for each frame. The parallelized decoder is not significantly affected by the precision of each pixel. We further measure the decoding performance with different bitrates in production in Appendix B.4.

Preset the frame rate and resolution based on client types. An alternative to AFR is that the application checks whether the hardware could reliably decode the video at a certain resolution and frame rate at initialization. This, however, would lead to underutilization on the client side. The decoding capability of hardware is fluctuating over time due to various reasons. For example, we measure the distribution of decoding delay of each user session in Appendix B.5. One-fourth of users will have at least 1‰ time of a long decoding delay of >18ms, which could result in severe queuing delay, as we illustrated in Figure 6. In this case, if we set the resolution and frame rate based on this tail metric, users will have a much lower resolution and frame rate during most of the time. Therefore, we need to control the frame rate in the runtime to dynamically adapt to the network and decoder dynamics.

Allocating the application with dedicated resources. Another seemingly feasible solution is to bind the application to a certain CPU core or GPU core to avoid the potential fluctuations caused by scheduling. However, we do not have such privileged control on client devices. As a user space application, the controllability over the user's system is limited. Even if an expert user pins the application to a certain core, for commercial systems such as Windows, pinning does not indicate isolating the core for that application only [14]. The system can only ensure the pinned application to run on that core, but could also schedule other processes if still available. Moreover, since our application is not CPU-intensive most of the time, there would usually be idle resources on the same core where the user binds the application to. Therefore, there could still be the same issue of latency increases *at tail*.

CPU	Release date	Score	Portion
Intel® Core™ i5-4590	Q2 2014	868	1.66%
Intel® Core™ i5-7200U	Q4 2016	481	1.61%
Intel® Core™ i5-9400F	Q1 2019	1058	1.56%
Intel® Core™ i5-4460	Q2 2014	801	1.41%
Intel® Core™ i5-5200U	Q4 2014	573	1.38%

Table 3: Top 5 CPU models of clients in our cloud gaming service.

GPU	Release date	Score	Portion
Intel® UHD Graphics 630	Q3 2017	888	4.54%
Intel® HD Graphics 4600	Q2 2013	474	3.42%
Nvidia GeForce GTX 1050Ti	Q4 2016	5059	3.19%
Intel® HD Graphics 630	Q3 2016	825	2.77%
Nvidia GeForce GT730	Q2 2014	863	2.48%

Table 4: Top 5 GPU models of clients in our cloud gaming service.

A.2 Practical Concerns

Since the frame-rate needs to be adjusted at the server, a straightforward concern is whether the frame-rate adaption over the Internet is timely for the stringent delay requirement of high-quality RTC. The measurements in production have two following findings. On one hand, the round-trip network delay is short enough to enable timely feedback: the average round-trip network delay is around 20ms of our cloud gaming service (Appendix B.2). Measurements over other high-quality RTC services (e.g., Google Stadia) have similar results of less than 20ms [26, 57]. On the other hand, the degradation of decoding delay usually lasts for a long time, with a median duration of more than 100 milliseconds (Appendix B.5). Moreover, we also demonstrate that the increase in decoding delay and network delay is hardly correlated (Appendix B.6). Therefore, for high-quality RTC, when the decoder fluctuates, it is timely enough to control the frame rate over the Internet.

B Measurement over Dataset

In this section, we supplement the observations in the main text with measurements in production. The measurement settings follow the details in §5.2.

B.1 User Characteristics

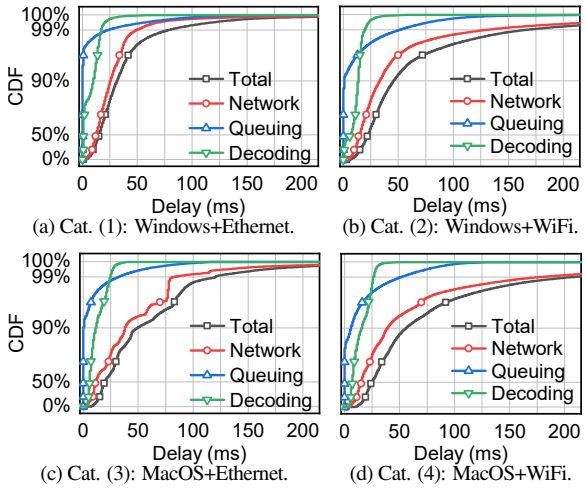
In addition to the distribution in §3.1, we present the top-5 models, with their release dates, benchmark scores, and portion in our users, of CPU and GPU in Table 3 and 4.

B.2 Delay Distributions

Compared to traditional RTC scenarios, the delay distribution for high-quality RTC has some unique features according to our measurements. We present the Cumulative distribution function (CDF) of component delays and the total delay to explore the delay patterns.

First, due to the edge deployments, the network delay in our cloud gaming service is quite small. According to Figure 20, the average round-trip network delay is approximately around 20ms. Even in this case, similar to traditional RTC services, the network delay is accounted for a large part of the total delay, the network delay line closely follows the total delay at the median for all four categories in Figure 20.

However, the tail delay of others component delays like



(a) Cat. (1): Windows+Ethernet. (b) Cat. (2): Windows+WiFi. (c) Cat. (3): MacOS+Ethernet. (d) Cat. (4): MacOS+WiFi.

Figure 20: Raw measurements of delays from production.

decoding delay and queuing delay are noticeable under cloud gaming scenarios. For the decoding delay, we can notice that the decoding delay for 1080p frames is 18ms at the 99th percentile. Note that the decoder of all sessions evaluated in this paper has been hardware-accelerated. Therefore, as analyzed in §3.1, the queuing delay is becoming noticeable at the tail. Referring to Figure 20, the 99th percentile of queuing delay can reach 50ms under categories (2) and (4), which could degrade users' experience for high-quality RTC services. We further present the root cause analysis below in Appendix B.3.

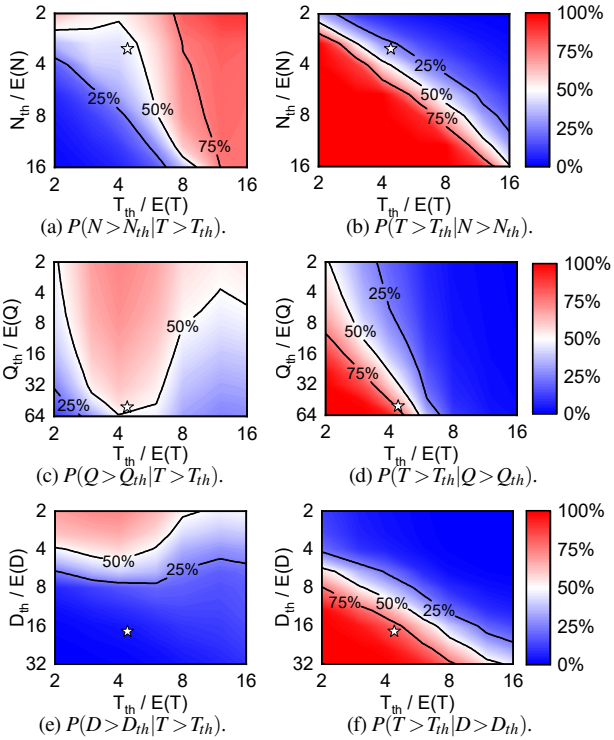
B.3 Root Cause Analysis

The total delay is mainly contributed by the network delay, decoding delay, and queuing delay §3. Therefore, we want to investigate how these three components contribute to the increase in total delay at the tail. For each frame, we denote T as total delay and C as component delay, where the component delay could be the network, decoding, or queuing delay.

To analyze the necessity and sufficiency of the component delay increasing to the total delay at the tail, we then calculate two conditional probabilities between the event of T longer than a certain threshold T_{th}, and the event of C longer than a certain threshold C_{th}:

- $P(C > C_{th}|T > T_{th})$. We want to account for how component delay increasing contributes to total delay under different delayed degrees T_{th}, and this conditional probability is subject to quantify it. If this conditional probability is close to one, there will be great confidence to blame the component delay for contributing C_{th} delay to the total delay to reaching T_{th}.
- $P(T > T_{th}|C > C_{th})$. As the sum of component delays, the total delay should increase when one of the component delays increases. This conditional probability is subject to illustrate this assumption and indicates the probability of total delay reaching the T_{th} under different component delay increasing degree C_{th}.

We calculate the conditional probabilities for three components for different C_{th} and T_{th}, and have the following observations.



(a) $P(N > N_{th}|T > T_{th})$. (b) $P(T > T_{th}|N > N_{th})$. (c) $P(Q > Q_{th}|T > T_{th})$. (d) $P(T > T_{th}|Q > Q_{th})$. (e) $P(D > D_{th}|T > T_{th})$. (f) $P(T > T_{th}|D > D_{th})$.

Figure 21: The heatmap of conditional probabilities for wired connections. The horizontal and vertical axes have been normalized by their average values. The star point's value is recorded in table 5 The down-left corner is 100% since the total delay should always be larger than the component delay.

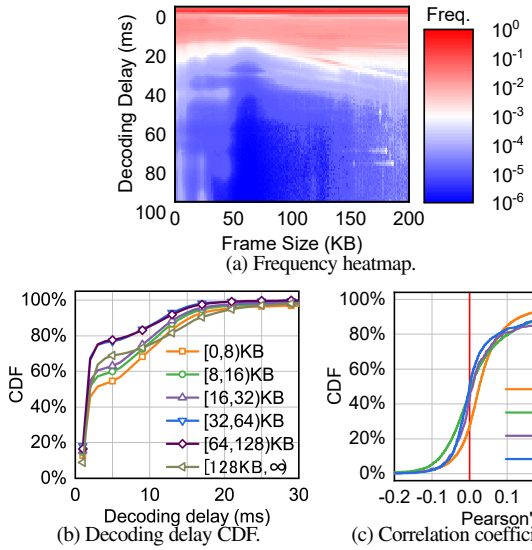
	Network	Queuing	Decoding
$P(C > C_{th}\|T > T_{th})$	44.7%	**56.6%**	4.0%
$P(T > T_{th}\|C > C_{th})$	29.8%	**69.5%**	84.2%

Table 5: Conditional probabilities with $T_{th} = 100ms$ and $C_{th} = 50ms$ for wired connections, which accounts for 82% of total users of our cloud gaming service.

Total delay increasing is a reflection of components delay increases. As the sum of the different types of components delay, It's obvious that no matter what kind of component delay is increasing, the total delay will also increase.

So to find out the sufficiency of total delay increasing, we calculate the conditional probability of $P(T > T_{th}|C > C_{th})$ in right-side of Figure 21. We can notice that for all the component delays, their delay increasing can also mean a higher probability of total delay increasing (75%ile line in the figure is shifting to the right with the component delay increasing). The down-left corner is 100%, because as the sum of all types of component delay, the total delay must be larger than any component delay.

Queuing delay is responsible for delay increases of >100ms. To figure out the necessity of total delay increasing, we calculate the conditional probability of $P(C > C_{th}|T > T_{th})$ in left-side of Figure 21. Our major finding is that with the different order of severity of total delay increasing (2-16× of $\mathbb{E}(T)$, the root cause of it is also changing. As we can see, when T_{th} is larger than $8\mathbb{E}(T)$, network delay has a high probability



(a) Frequency heatmap. (b) Decoding delay CDF. (c) Correlation coefficient CDF.

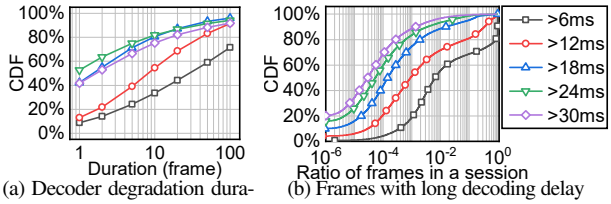
Figure 22: The correlation between the frame size and decoding delay for hardware decoders.

(shaded red) to be blamed. However, when T_{th} is from $3\mathbb{E}(T)$ to $8\mathbb{E}(T)$, queuing delay dominates the most increased events. It illustrates that the queuing delay is responsible for the increase of total delay by around 100ms. Specifically, we present the conditional probabilities for three components with $T_{th} = 100ms$ and $C_{th} = 50ms$ for wired connections in Table 5. As we can see, queuing delay has both high $P(C|T)$ and $P(T|C)$. Indicating that the total delay has a great possibility of reaching 100ms when queuing delay increases to 50ms. And for those video frames that total delay truly getting the 100ms, there will be great confidence to blame the queuing delay for contributing to the majority of delay increasing. So the queuing delay will be the root cause of the increase of total delay to 100ms.

B.4 Decoding Performance

In this section, we explain the reasons behind the ineffectiveness of controlling the service process for eliminating queuing time by adjusting the bit rate. The decoding time of decoders mainly depends on the resolution of the streaming. However, due to the dependency between frames, changing the resolution during the streaming will make the subsequent frames undecodeable and needs to request a new key frame for most codecs [29]. Yet, since the frame size of key frames is usually several times ofthose of other frames [46], frequently requesting key frames will impose additional overhead on the network and degrade the users' experiences.

Another straightforward solution is to try to accelerate the service process by reducing the bit rate while maintaining the same resolution. With the same resolution and frame-rate options, reducing the bit rate means lesser video data per video frame can carry. We are to investigate whether sending video frames with smaller data sizes is helpful for decoding acceleration. However, according to our measurements on the H.264 decoder, merely changing the bit rate does not significantly reduce the decoding time.



(a) Decoder degradation duration. (b) Frames with long decoding delay

Figure 23: Decoder degradation when filtered with different thresholds for decoding delay.

We measure the relationship between the frame size and decoding time of the dataset described in §5.2. We first present the heat map in Figure 22(a). With the variation of frame size, the distribution of decoding time does not significantly change, where the decoding time of most frame sizes intensively falls around several milliseconds, as shown in the red area at the top of the heat map. To eliminate the frame size variation under the same target bit rate, we split the frame size into different intervals and present the cumulative distribution function (CDF) in Figure 22(b). As the frame size become larger, the [128KB, ∞) the line does not locate in the rightest area (higher decoding delay). And other frame size interval's CDF lines stay together, indicating that the lowering frame size does not help for the decoding time acceleration.

Moreover, we split the dataset into four different categories (Table 1), to demonstrate that reducing frame size will not help decode acceleration under various platforms. We leverage the Pearson correlation coefficient to illustrate the independence, which value of zero can indicate that there is no association between the two variables [70]. Figure 22(c) shows that most of the Pearson's r value is located around zero, indicating the poor association between frame size and decoding delay. Therefore, controlling the service process of encoding bit-rate cannot effectively reduce the decoding time and alleviate the load of the decoder queue.

B.5 Decoder Degradation

Because the queue overhead will be introduced by the mismatch of the rate of two sides of the queue [39], if the decoding speed is not capable of processing the incoming default 60fps, it will be necessary for AFR to change to a lower target frame rate. However, since the client and server are located distant, the frame-rate adjustment request to the server side will need a control loop to take effect on the client side with the updated frame rate. So if the AFR control loop is shorter than the decoder degradation duration, the decoder will be capable of processing a higher incoming frame rate before the AFR requests take effect.

We measure the duration of the decoder degradation level over the traces introduced in §5.2. As we can see in Figure 23(a), for frames with a decoding time of more than 12ms, 50% of them last for more than 10 frames. Under 60fps streaming, considering the average of RTT is close to one frame interval of 16.7ms, and the 90%ile encoder response delay is less than three frames interval §6.4. In this case, lowering the frame rate will be helpful for alleviating the decoder queue even under the control loop delay of AFR. Therefore, AFR is capable of timely adjusting the frame rate to adapt to the decoder degradation. Moreover, the

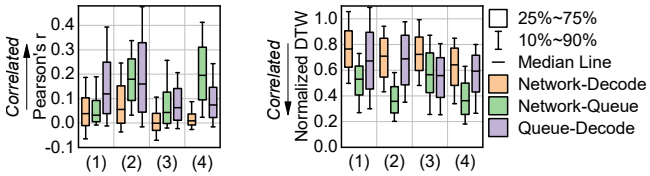
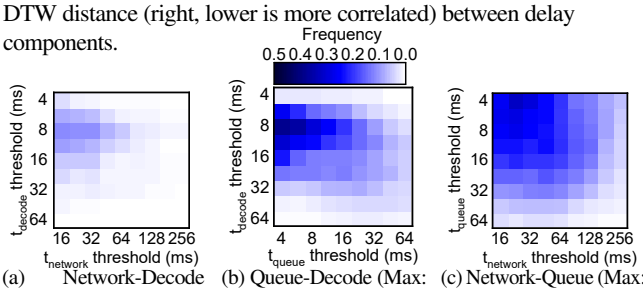


Figure 24: Pearson's r (left, higher is more correlated) and normalized DTW distance (right, lower is more correlated) between delay components.



(a) Network-Decode (Max: 0.12). (b) Queue-Decode (Max: 0.39). (c) Network-Queue (Max: 0.32).

Figure 25: Cramer's V between different delay components.

AFR can significantly help alleviate the queue overhead under those frames with a long period of decoder degradation and sustain queuing time for waiting for overhead queue elimination.

We further measure the ratio of frames with different decoding delays and present the results in Figure 23(b). Half of the user sessions suffer from a decoding delay of >12ms for at least 1‰ frames. This also indicates that the degradation of decoding delay is a general issue among all clients.

B.6 Component Correlation Analysis

The streaming pipeline will be affected by many components, like the networking, decoding, and queuing delays can both cause total delay increases to degenerate the user's experience Appendix B.3. In this paper, we propose AFR to reduce the tail queuing delay by matching the arrival rate of the decoder queue to the service rate (decoding speed). When decoding delay increases to disable decode frames timely, the AFR will send a frame-rate adjustment request from the client to the server. However, the request and subsequent frames need to be transported through the network. Therefore, a straightforward question is: does the increase of decoding delay affect the network delay to put an extra effect on the AFR control loop? We will figure out this by measuring the independence of those component delays.

We quantify the independence of different component delays with Pearson's r value [70], dynamic time warping (DTW) [18], and Cramer's v value [31]. In short, all these metrics demonstrate the poor association between networking and decoding delay, inclining that we could decouple the network and decoder issues and independently control them.

Regarding the Pearson correlation coefficient, the value of zero can indicate that there is no correlation between the two variables [70]. Figure 24 illustrates that for all four categories in Table 1, the Pearson's r value of networking and decoding are close to zero, indicating a poor correlation between them.

Moreover, the different component delays might be correlated with each other across frames. For example, the decoding delay

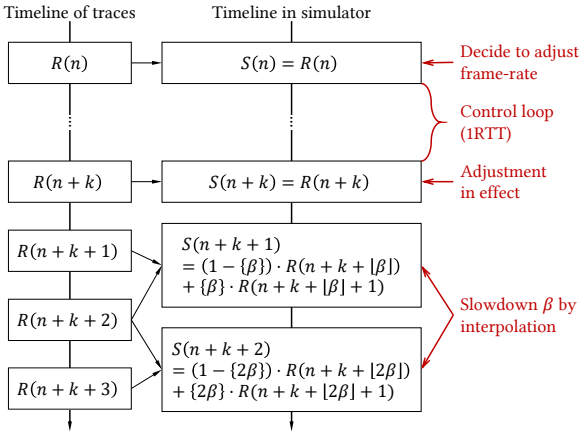


Figure 26: Illustration of frame-rate adjustment in our simulator.

could affect the subsequent queuing delay by its incapacity to decode video frames timely. To measure the correlation across frames, we leverage DTW to calculate the optimal match between two-time series [18]. The DTW algorithm is widely used in many scenarios like sign language recognition and time series clustering [47, 59]. The optimal match calculation under DTW is denoted by the match with minimal cost, where the cost is computed as the sum of absolute differences, for each matched pair of indices, between their values. Therefore, a larger DTW distance can be considered the mismatch between two series to a further extent. According to Figure 24, the normalized DTW distance of network delay to decoding delay under all four categories is large, showing the lack of correlation between them.

The strength of the relationship can also be assessed by Cramer's V value, which is a metric based on the χ^2-test but normalized for different data sizes. It indicates how strongly two categorical variables are associated [31]. A Cramer's V value of $\leqslant 0.1$ can be interpreted as hardly correlated [22]. According to our measurement in Figure 25, we can notice that all the Cramer's V values of networking and decoding delay are $\leqslant 0.2$, illustrating the weak association between networking and decoding state. Therefore, according to our measurements before, we can see the independence between networking and decoding delay.

C Simulator Implementation

In this section, we introduce the implementation of our simulator. Specifically, traces are recorded in the following format:

$$R(n) = \left[ts^{(n)}, \tau_{net}^{(n)}, \tau_{queue}^{(n)}, \tau_{decode}^{(n)}\right] \tag{9}$$

where $ts^{(n)}$ is the arrival timestamp of the n-th frame, τ_{net}, τ_{queue}, and τ_{decode} are the round-trip network delay, queuing delay, and decoding delay of that frame. The simulator reads the traces frame-by-frame at specific timestamps and measures the current frame rate based on the interarrival time as §4.2. The simulator then dequeues the head frame in the decoder queue when the decoder is available, where the decoding time of each frame is also read from the trace.

When the adaptive frame-rate decides to set the frame-rate

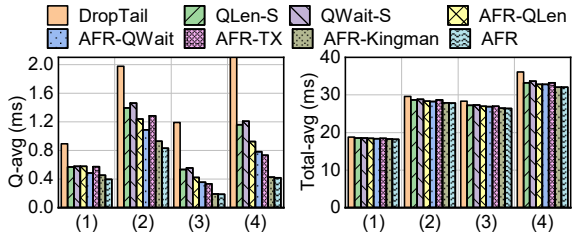


Figure 27: Average queuing delay (left) and total delay (right).

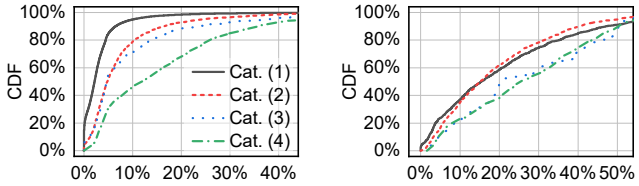


Figure 28: The number of wasted frames when skipping frames instead of adjusting the frame rate for AFR.

to f_{set}, the simulator first reads the current control loop by the round-trip network delay of the current frame $\tau_{net}^{(n)}$. The simulator then calculates the earliest frame $n+k$ that the new frame-rate f_{set} will take in effect:

$$k = \underset{k}{\operatorname{argmin}}\left(ts^{(n+k)} - ts^{(n)} \geqslant \tau_{net}^{(n)}\right) \tag{10}$$

After that, based on the measurement of the current frame-rate f_{cur}, the simulator calculates the slowdown factor $\beta = f_{cur}/f_{set}$, and reads the traces with a slowdowned speed. For example, as shown in Figure 26, When there are frames $R(n+k+1)$ to $R(n+k+3)$ in the original trace, the simulator reads the traces with indices $R(n+k+\beta), R(n+k+2\beta), \cdots$. When β is not integer, the simulator interpolates the traces with its neighbor frames ($S(n+k+1)$ and $S(n+k+2)$).

D Supplementary Experiments

D.1 Average Delay

We further measure the average queuing delay and total delay for four traces and present the results in Figure 27. As we can see, the reduction of tail delay of AFR does not sacrifice the average delay on all traces. In contrast, the average delay has also been slightly improved against baselines, due to the improvements at the tail.

D.2 Frame Costs of AFR with Skipping

Besides, as we discussed in §6.4, skipping frames without changing the frame rate from the content generator (e.g., gaming application) would waste the rendering resources of the server. For example, for high-quality RTC, rendering at 60fps would take approximately one time more GPU resources than rendering at 30fps. Therefore, we measure how many frames have been wasted (i.e., frame cost) if we merely skip the frames to approximate the target frame rate without adapting the content generator.

We present the results of all traces in Figure 28. For all traces, adjusting the frame rate could save 3% to 12% frame costs in all traces, saving considerable operating expenses for the service provider since GPU is one of the highest expenses. For stuttered sessions (following the definition in §6.2), the saved frame cost would be even higher.

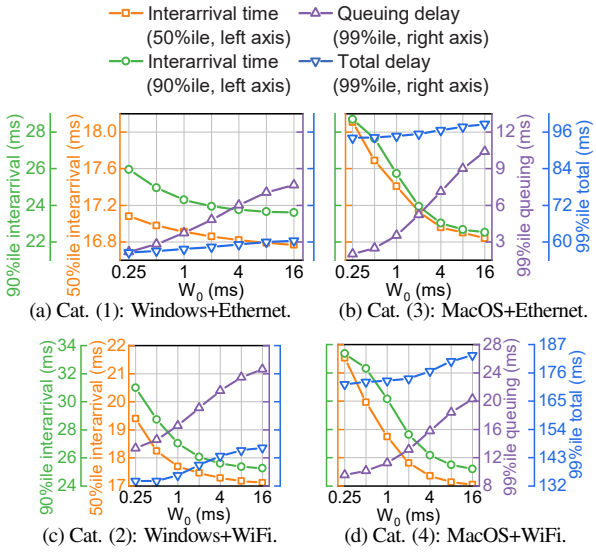


Figure 29: Sensitivity analysis on W_0 on different traces.

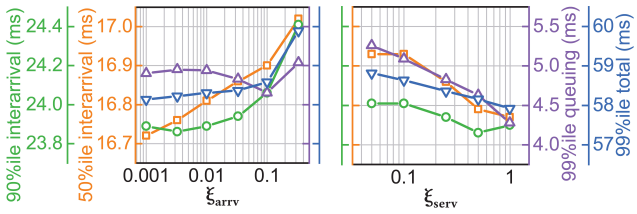


Figure 30: Performance of AFR with different settings of ξ_{arrv} and ξ_{serv}. Y-axes have been magnified compared to Figure 29.

D.3 Parameter Sensitivity

Long-term control target (W_0) We present the simulation results on the sensitivity of W_0 (in the stationary controller) on different traces in Figure 29. As we discussed in §5.2, a lower W_0 results in a more aggressive queue control yet leads to the degradation of frame rate. We vary W_0 from 0.25ms to 16ms and measure the interarrival time, queuing delay, and total delay. By adjusting W_0, operators could effectively balance the total delay and frame rate. Therefore, operators could adjust W_0 according to the preferences on total delay and frame rate for different users and games.

EWMA discounting factors (ξ_{arrv} and ξ_{serv}). We also vary the EWMA discounting factors (ξ_{arrv} for the arrival process and ξ_{serv} for the service process). Higher ξ indicates that the EWMA focuses on the recent values more to capture changes, while a lower value indicates more attention to the historical trends. As shown in Figure 30, the performance metrics (including the queuing delay, total delay, and frame rate) are relatively robust to these two parameters. By varying ξ_{arrv} and ξ_{serv} across several magnitudes, most metrics change marginally. For example, the 99%ile of queuing delay changes by 4× when varying W_0 (Figure 29) while only changes by less than 15% when varying ξ_{arrv} by three magnitudes (Figure 30). We also observe trends in varying ξ_{arrv} and ξ_{serv}. Lower ξ_{arrv} values will slightly

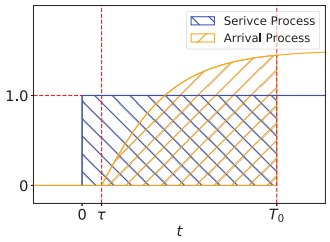


Figure 31: The system begins to control the queue after control-loop delay τ and stabilize the queue at T_0.

improve the performance of AFR, implying that the long-term behavior of arrival service is more critical. Higher ξ_{serv} also slightly improves the performance, indicating focusing on recent decoding time is helpful. This is because we have already filtered out outlier decoding time. Paying more attention to recent decoding time could make the AFR quickly adjust the frame rate.

E Convergence Analysis

Finally, we provide a detailed analysis of the convergence time during the state transitions of the stationary controller. As introduced in §4.2, let the expectation of queuing delay $E(\tau_{queue}) = W_0$, according to Eq. 1, we have:

$$\mu_a = \frac{\mu_s}{\rho} = \left(1 + \frac{c_a^2 + c_s^2}{2W_0}\mu_s\right)\mu_s \tag{11}$$

Then we can discuss the *convergence time* of the system. The convergence time here refers to the time at which the stationary controller converges to a stationary state when the service process changes, and the potential accumulated queue during the transition is drained up.

Specifically, without loss of generality, we discuss a simplified case shown in Figure 31: Both the arrival and service process have an average value of zero for $t<0$, and the service process changes from zero to one at $t=0$. The arrival rate will gradually respond to the change after a control loop of τ. We want to find the convergence time T_0 where

$$\int_0^{T_0} \mu_a \mathrm{d}t > \int_0^{T_0} \mu_s \mathrm{d}t \tag{12}$$

In this case, the queue accumulated during the response to the arrival rate will be cleared. We further illustrate the convergence in Figure 31. By substituting Eq. 11, we have:

$$\int_\tau^{T_0} \left(\mu_s + \frac{c_a^2 + c_s^2}{2W_0}\mu_s^2\right)\mathrm{d}t > \int_0^{T_0} 1\mathrm{d}t \tag{13}$$

From the measurement of EWMA in Eq. 5, we have

$$\hat{\mu_s} = 1 - (1-\xi_\mu)^{t-\tau} \quad (t>\tau) \tag{14}$$

Therefore, let $\gamma = 1-\xi_\mu$ to simplify the expression, we need to find the minimum T_0 such that:

$$\int_0^{T_0-\tau} \left((1-\gamma^t) + \frac{c_a^2 + c_s^2}{2W_0}(1-\gamma^t)^2\right)\mathrm{d}t > T_0 \tag{15}$$

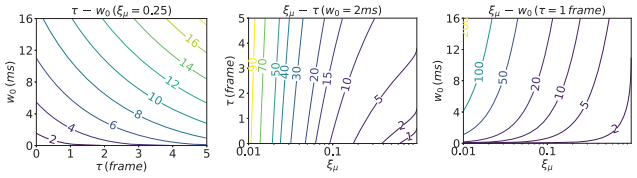


Figure 32: Contour plot of the convergence region of T_0 with different parameters.

By solving the integral in Eq. 15, finally we have

$$W_0 < \frac{c_a^2 + c_s^2}{2} \frac{(\gamma^{T_0-\tau}-1)(\gamma^{T_0-\tau}-3) + 2(T_0-\tau)\ln\gamma}{2(\gamma^{T_0-\tau}-1) + 2\tau\ln\gamma} \tag{16}$$

For example, when set $c_a^2 + c_s^2 = 2$, we vary the other parameters in Eq. 16 and present the minimum T_0 in Figure 32. In the most general settings of AFR ($\tau = 1$ since the average RTT is around 15ms, $\xi_\mu = 0.25$ as introduced in §5.2, $W_0 = 2$ms), the stationary controller can converge to the new stationary state within 2 frames. In other settings of the AFR parameters, the stationary controller could also converge and drain the queue within tens of frames, which is much less than the frame-rate adjustment interval of hundreds of frames as evaluated in §6.2.

LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed

Hao Li[1], Yihan Dang[1], Guangda Sun[1,2], Guyue Liu[3], Danfeng Shan[1], Peng Zhang[1]

[1]*Xi'an Jiaotong University* [2]*National University of Singapore* [3]*New York University Shanghai*

Abstract

NFV has entered into a new era that heterogeneous frameworks coexist. NFs built upon those frameworks are thus not interoperable, obstructing operators from getting the best of breed. Traditional interoperation solutions either incur large overhead, *e.g.*, virtualizing NFs into containers, or require huge code modification, *e.g.*, rewriting NFs with specific abstractions. We present LemonNFV, a novel NFV framework that can consolidate heterogeneous NFs without code modification. LemonNFV loads NFs into a single process down to the binary level, schedules them using an intercepted I/O, and isolates them with the help of a restricted memory allocator. Experiments show that LemonNFV can consolidate 5 complex NFs without modifying the native code while achieving comparable performance to the ideal and state-of-the-art pure consolidation approaches with only 0.7–4.3% overhead.

1 Introduction

The past decade has witnessed the flourish of Network Function Virtualization (NFV) research, with the goal of replacing hardware middleboxes with software network functions (NFs) running on commodity servers. Prior research efforts have led to a plethora of NFV frameworks focusing on various aspects, including performance optimization [30, 33, 35, 49], programming models [34, 41], resource management [51, 60], and more recently security [43, 52, 54]. Since there are no conventional and widely-adopted interfaces, these NFV frameworks are unsurprisingly implemented in *heterogeneous* ways, with different libraries (*e.g.*, DPDK [5], netmap [57]), languages (*e.g.*, C, C++, Rust), and abstractions (*e.g.*, Click element [37], BESS module [30]).

NFs built upon these heterogeneous NFV frameworks are not *interoperable*, which exposes two hard choices for users in actual deployment. The first choice is asking operators to learn, deploy, and maintain multiple frameworks to serve different purposes. The second choice is picking one framework and asking the developers to add extra functionalities by reinventing the wheel. The former choice dramatically increases the cost of operation, and the latter choice requires substantial engineering effort and is error-prone. This reality raises a timely and important question: *can heterogeneous NFs interoperate without modifying the code?*

To answer this question, the first natural candidate solution is using virtualization. Under this model, each NF runs in a virtual machine or a container on one core, and packets are steered across cores to chain multiple NFs. While this approach hides the heterogeneity of NFs under standardized virtualization interfaces, it incurs prohibitive performance overhead when chaining multiple NFs [33, 49]. The overhead stems from three main sources: virtualized components, cache misses, and context switches (details in §2.1). Despite the efforts of various lightweight virtualization techniques [39, 56, 68], these overheads cannot be fully eliminated. As a result, this virtualization approach fundamentally cannot achieve the line speed, one of the key requirements of NFV deployment.

The second candidate solution is using consolidation. This model implements NFs as software modules and runs them in the *same* process. NFs are chained through function calls and scheduled in a *run-to-completion (RTC)* mode, *i.e.*, once a packet is received by an NF, the NF continues processing it until finishes. By eliminating cache miss and context switch overheads, this approach ensures the line-rate processing and has been adopted by existing high-performance NFV frameworks such as Metron [35] and BESS [30]. While consolidation works well for NFs under the same framework, it seems *unlikely* to be applied to heterogeneous NFs without modifying the code due to three key challenges across loading, scheduling, and memory management.

- *How to launch heterogeneous NFs in one process?* Launching NFs written with different frameworks and languages into the same process could result in various conflicts (*e.g.*, dependencies, functions, variables). A potential solution that manually modifying the code to resolve each conflict could be tedious.
- *How to chain multiple NFs with separate control flows within a process?* Each NF has its own control flow

which often includes an infinite loop of packet processing. Chaining these separate control flows requires locating new entry points and correctly schedule them to process packets.

- *How to isolate memory of multiple NFs within a process?* NFs running in the same process share the same stack and heap, without memory isolation. This brings security concerns when chaining multiple NFs from different and possibly untrusted vendors. A previous solution that rewrites NFs with safe programming languages would incur large porting efforts.

In this work, we propose *LemonNFV*, an NFV framework that enables consolidating heterogeneous NFs without modifying the code. To address the above challenges, LemonNFV provides three abstractions for NFs: the compiled binary, the scheduling entry points, and the memory allocation interfaces. The key point of these abstractions is that they (1) are essential and sufficient for making NFs interoperable, and (2) can be easily adopted to NFs without much coding efforts by capturing the natural homogeneity that already exists in all NF implementations. By transforming NFs into such *LEast Modified network functiONs (LEMONs)*, LemonNFV then implements a set of utilities upon those abstractions that load, execute and manage NFs inside a process. Concretely, we make the following contributions in designing LemonNFV.

Loading via LEMON Binary (§4.1 and §4.4). We view each NF as a software module and leverage the standard binary format to load them. The binary hides the complexity of NFs written in different languages and/or with conflicting names. To further enable dynamic chaining and NF migration, we build a *LEMON loader* to specify memory layout and resolve dependency conflicts, both of which are not supported by existing loaders (*e.g.*, `ld.so`).

Chaining via Schedulable I/O (§4.2). We observe that packet I/O could be an ideal scheduling point where an NF's processing logic starts and ends. Based on this observation, we provide unified I/O interfaces as entry points for inter-NF scheduling. These interfaces can replace existing NFs' I/O interfaces, which are usually built on top of several common packet I/O libraries, such as DPDK, `libpcap` and netmap. To chain multiple NFs inside one process, we provide a *scheduler* to correctly switch between different NF control flows.

Isolation via Restricted Memory Interfaces (§4.3). We use a *restricted allocator* to set explicit *NF boundaries* by creating private stack and heap for each NF, which can replace the native memory allocation interfaces like `malloc` and its variants. To isolate different NFs in the process, we implement an *isolator* that leverages hardware-aided technique (Intel PKU) to realize efficient intra-process memory isolation [1].

We evaluate LemonNFV with real NFs and traffic (§6). The results show that LemonNFV can (1) consolidate 5 complex NFs without code modification – even they are implemented with different frameworks and/or languages; (2) realize comparable performance to the ideal and state-of-the-art consolidation approaches with only 0.7–4.3% overhead of isolation. *Ethics:* This work does not raise any ethical issues.

[1] Currently, our isolation model focuses on memory isolation only and does not support control flow integrity, state sharing and packet isolation as in related works [31, 52, 66]. See §4.3 and §7 for details.

2 Motivation

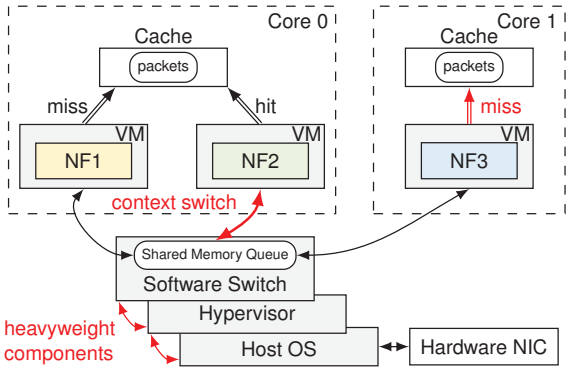
In this section we explain why neither virtualization (§2.1) nor existing consolidation techniques (§2.2) can address the need of heterogeneous NF interoperation.

2.1 Virtualization is Slow

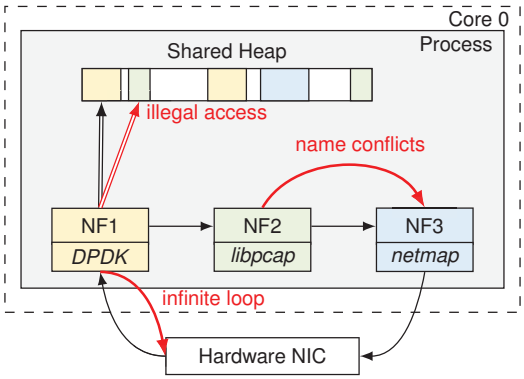
Figure 1a shows the three types of performance overhead incurred by virtualization approaches. *(V1) Heavyweight components:* Packets may need to go through all levels of virtualization components: host OS, VM hypervisor, guest OS and software switch. Each of the components brings a non-negligible overhead since they may run protocol stacks and perform queueing. *(V2) Cache misses:* If the NF instances are deployed on separate cores, passing packets across cores will be inevitable, in which case accessing each packet will become LLC or DRAM bounded. Moreover, passing packets across cores is often achieved by software switches, requiring frequent enqueuing and dequeuing when the NFs send and receive packets. *(V3) Context switches:* Scheduling will happen if there are multiple NF instances pinned to a single physical core, which would result in extra switching overhead.

Researchers have been leveraging the emerging lightweight virtualization techniques to improve the performance of virtualization NFV systems. By reducing full-fledged VMs into more lightweight environments, such as containers [21, 61, 63, 68], unikernels [39, 46, 69], and even processes [40, 42, 56, 76], the virtualization overhead, *i.e.*, *V1*, is greatly reduced or eliminated, yet still preserving *V2* and *V3*.

In fact, running NFs as separate instances makes it impossible to reduce *V2* and *V3* at the same time. Pinning instances to dedicated cores (*e.g.*, OpenNetVM [76], NFP [64]) eliminates the overhead of context switches (-*V3*), but forces packets to be delivered over shared memory and results in L1/L2 cache misses (+*V2*). On the other hand, compacting instances on a single core (*e.g.*, Quadrant [68], EdgeOS [56]) would reduce the cache misses on packets (-*V2*), while frequent context switches can lead to much more system calls and TLB misses (+*V3*). Therefore, we reach a conclusion that virtualization-based NFV frameworks can hardly meet line-rate (*i.e.*, 100/400Gbps) processing requirements because of its inherent overhead.



(a) A virtualization approach that runs NFs as separate instances. Red arrows indicate the major source of overhead, and grey boxes are components that can be optimized out.



(b) A consolidation approach that executes NFs in a single process. Black arrows are the ideal workflow while the red ones signify the obstacles when the NFs are heterogeneous.

Figure 1: The virtualization approach is slow, while the consolidation cannot handle the conflicts between heterogeneous NFs.

2.2 Consolidating Heterogeneous NFs is Hard

Consolidation, on the other hand, avoids all the extra overhead from the virtualization approaches by eliminating the boundaries between NFs [17, 30, 35, 52, 61]. Under this model, the SFC is deployed in one process, *i.e.*, no *V1*, and NF instances are executed in an RTC manner *i.e.*, no *V2* and *V3*, as shown in Figure 1b. However, it introduces even more challenges when trying to consolidate heterogeneous NFs.

Challenge 1: NFs cannot be loaded. NFs are independently developed with different frameworks and abstractions. As such, when putting them together in the same process, they may conflict with each other in terms of dependencies, functions and variables. For example, two NFs may define global data structures with the same name, while simply linking their source code would raise a multi-definition error. Things get more complex if NFs are written in different languages.

Naive solution: A naive solution for loading NFs into a single process includes three time-consuming tasks. First, operators have to check all symbols exposed by NFs to locate the conflicts. Second, they should manually resolve the conflicting symbols, which however is not always a feasible task, considering the conflicts that may happen between closed-source libraries. Third, they need to reconstruct other code with the resolved name, which usually requires deep understanding of the whole NF codebase. Despite the above tedious efforts, manual resolution can never work for the cross-language NFs or dynamic SFC updating without halt.

Challenge 2: NFs cannot be scheduled. The workflow of each NF is driven by *an infinite loop* of receiving and sending packets, *i.e.*, processing packets after receiving them, and receiving more after sending the processed ones. As a result, an NF will take up a core forever once it starts running, and the downstream NFs in the same process will never be scheduled.

Naive solution: To break the infinite loop inside each NF, one could extract the packet processing logic of each NF, and combine them together to form a synthesized SFC [17, 35, 36]. However, the packet processing logic is closely coupled with NF-specific packet and state abstractions, and combining them results in a large amount of code modification. For example, Snort [10] leverages its unique Data AcQuisition Library (DAQ) to receive packets and fill the metadata like timestamp, protocol annotation, *etc*. Such packet abstraction is incompatible with the packet abstraction under Click [37]. As a result, composing a synthesized SFC of Snort→Click requires to transform the packets as well as all metadata from the DAQ abstraction into the Click abstraction, and vise versa for Click→Snort.

Challenge 3: NFs may affect each other. Being in the same process, all NFs share the same stack and heap. In this way, operators are not able to restrict any memory operations of allocation, read and write, and thus lose the control over illegal accesses, *i.e.*, each NF can (unintentionally) modify others' data and make their states inconsistent. For example in Figure 1b, NF1 (yellow) can silently modify the data (red arrow) allocated by NF2 (green).

Naive solution: Consolidation frameworks often choose to trust the NFs under the same process since they come from the same vendor. However, trust cannot be granted when it comes to heterogeneous NFs across vendors. Using safe language, *e.g.*, Rust in NetBricks [52], to rewrite the NFs is a feasible option, but it's not practical given the large body of existing NFs and limited popularity of the safe language. Formal verification on NFs [72, 74, 75] on the other hand requires verification expertise and sometimes also forces NFs to use certain APIs [73].

3 LemonNFV Overview

The heterogeneous NF consolidation is challenging because NFs in the same process *share* the namespace, the control flow, and the memory. In this section, we introduce the key abstractions for breaking such sharing, and explain why they can be easily adopted without code modification (§3.1). We

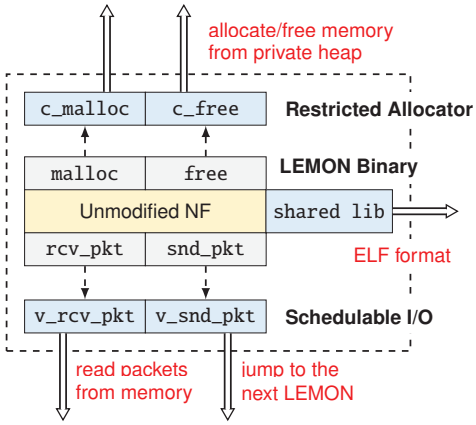


Figure 2: A LEMON with unified abstractions of binary, I/O and memory interfaces. Grey boxes are the common points existed in NFs, and the blue boxes modify/leverage their semantics for providing the unified behaviors (red annotations).

then overview the workflow of LemonNFV that implements a set of utilities upon those abstractions, including the loader, scheduler, isolator and the migration manager, to load, execute and manage LEMONs within the same process (§3.2). We finally discuss the ongoing challenges of LemonNFV (§3.3), which will be addressed in the next section.

3.1 Unified LEMON Abstractions

A LEMON carries an unmodified NF with its own namespace, control flow and memory, each of which could conflict with other LEMONs inside the process. The key challenge here is to decide the proper level to resolve the conflicts. Typical solutions either choose the lowest level for avoiding the code modification, *i.e.*, packaging NFs into OS-level containers, or operate at the highest level for ideal performance, *i.e.*, rewriting NFs' source code, neither of which can fully address our goal.

Instead, LemonNFV aims to resolve such conflicts with three middle-level abstractions: a binary that isolates the namespaces of each NF, a set of I/O interfaces that decouple the packet processing logic from infinite loop, and a set of memory interfaces that restrict the memory operations. These abstractions are high-level enough for resolving the underlying conflicts, while also low-level enough to hide the heterogeneity. In fact, there exists natural homogeneity in all NFs' implementation, and by leveraging it, LemonNFV can equip any NF with the proposed abstractions by a simple interception, *i.e.*, without code modification.

LEMON binary: wrapping the namespace. Simply putting all NFs' code together usually does not compile, because the independent-developed NFs might define variables, functions and dependencies with the same name but different semantics, which would cause name conflicts. Instead of manually wrapping an NF into an isolated namespace, *e.g.*, packing it to a C++ class with private members, we observe that the compiled binary naturally separates the namespaces of each software module, and the conflicts can be resolved through the symbol resolution process when loading the binary.

Natural homogeneity: ELF is the standard format for executables and shared libraries under Linux, which reveals a chance for LemonNFV to package each NF as a software module without modifying its native code but through a simple recompilation (right part in Figure 2).

Schedulable I/O: creating entry points. The packet processing logic of NFs are usually implemented as an infinite loop, which is not schedulable. Nevertheless, we observe that the I/O behaviors reveal the natural boundaries between NFs. To this end, we design a new set of I/O in substitution for the original, which creates explicit entry points of each NF for scheduling: for the packet receiving function, it could read packets from a shared memory region instead of the physical NIC; for packet sending, it could jump out of the infinite loop after pushing the packets back to specific queues on shared memory, according to the output port of the NF.

Natural homogeneity: NFs are built on *a handful of* I/O libraries like DPDK and `libpcap`, which means we can implement the schedulable I/O by only intercepting limited functions of these libraries (lower part in Figure 2).

Restricted allocator: separating memory domains. To realize the isolation requirement, one should create isolated memory regions for each of the NFs, even with the same process. While not all NFs have the compilation support from safe languages like Rust, the feasible solution is to create separate (instead of interleaved) memory domains for each NF, and protect those domains with bound guard [67] or privilege management [14, 53].

Natural homogeneity: All NFs rely on the native allocator, which only provides limited functions like `malloc`, `realloc` and `free`. That is, LemonNFV can override the native allocator with the restricted allocator, which enforces that the dynamic memory allocation in a LEMON takes place in its own heap (upper part in Figure 2).

3.2 LemonNFV Workflow

Having the above unified LEMON abstractions, LemonNFV implements several key components to build, execute and manage LEMONs. As shown in Figure 3, an SFC is a process (dashed rectangle) that runs two types of threads: hypervisor (red) and worker (gray). The hypervisor consists of two components: the LEMON loader that loads LEMONs and intercepts the original allocator and I/O functions, and the migration manager that migrates the LEMON to another worker, SFC, or server. The worker implements the trampolines, which process the packets from the hardware NIC, schedule a chain of LEMONs with virtualized I/O while ensuring their isolation. There is also a LemonNFV controller, relaying the user commands like LEMON loading to the hypervisors, which can be deployed on a remote server.

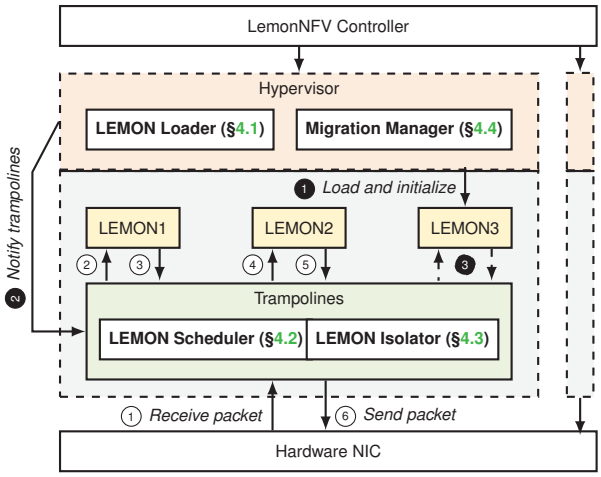


Figure 3: LemonNFV workflow, where a dashed rectangle represents a process (SFC) with two threads: a hypervisor (red) and a worker (grey). ①–⑥ illustrate how a packet being processed; and ❶–❸ are for runtime SFC changing.

Loading a LEMON. Each LEMON carries an unmodified NF, as well as its dependent libraries and configuration files. To this end, the NF developers are required to recompile the NFs into shared libraries, which only needs a few lines of modification in the `make` rules (See §6.2). This task can also be done by the operators if the target NF is open-source. Then, the operator should fetch all dependencies of the LEMON, pass their paths to the loader, and specify the total amount of memory needed. With these information, the LEMON loader would allocate the memory, load the code and variable segments, resolve the potential conflicts, and grant the corresponding privileges. The user can then consolidate an SFC by providing the path and interconnection of each LEMON in an interactive terminal.

Chaining and isolating LEMONs. Inside each worker, LemonNFV executes a chain of LEMONs in the RTC way. When receiving a (batch of) packet(s) from NIC (①), the LEMON scheduler in the trampolines will pass the packet to the first LEMON of the SFC, by calling its schedulable packet-receiving function (②). After the processing in LEMON1, the schedulable packet-sending function would transfer the control flow back to the trampolines (③), which would trigger the next LEMON, *i.e.*, LEMON2 (④). When it reaches the end of the chain (⑤), the trampolines send the packet(s) back to NIC (⑥). All of ①–⑥ are done within a single thread, thus will not produce any inter-core communication or thread context switch. When an SFC wants to scale out its performance, LemonNFV can duplicate the worker into more cores, and dispatch the flows using hardware NIC.

The trampolines also ensure the LEMON isolation through above chaining process. Specifically, the LEMON isolator would adjust the memory access privileges of each LEMON, *e.g.*, when executing LEMON2, the memory domains of LEMON1 and LEMON3 should be protected.

Managing LEMONs in runtime. Consider a simple management task that attaching LEMON3 to the end of current SFC. The hypervisor would first load and initialize LEMON3 (❶) using the LEMON loader. After that, it will notify the trampolines with the new LEMON, *i.e.*, the packet receiving function of LEMON3 (❷). Finally, the trampolines will execute LEMON3 next time a packet leaves LEMON2 (❸). Note that the hypervisor is in an individual thread, which means ❶ can be done asynchronously without halting SFC. ❷ is also lightweight, as the hypervisor only needs to notify the trampolines with the new entry point, which is a rare operation and would only halt the SFC for a negligible moment.

3.3 Ongoing Challenges

While making the LEMON interoperation a possible vision, LemonNFV is still faced with several practical challenges.

Isolated LEMON namespace. LEMON binary wraps the namespaces of each NF, but the name conflicts between multiple binaries still needs to be resolved. This is the responsibility of the LEMON loader, while the OS-default loader cannot suffice, because it tends to reuse the dependencies for all LEMONs.

Correct LEMON scheduling. Having the schedulable I/O, the LEMON scheduler in the trampolines needs to further address the following concerns for correctly scheduling the LEMONs. (1) how to efficiently switch LEMONs; (2) how to properly execute the logic other than the packet processing, *e.g.*, initialization; and (3) how to correctly handle the complex I/O behaviors like asynchronous Rx and Tx.

Efficient memory isolation. The restricted memory allocator guarantees the separate memory domains for each LEMON. However, it is still unclear how the LEMON isolator restricts the memory accesses to the legal areas. The key challenge here is to isolate those memory domains without compromising too much performance.

Flexible LEMON migration. NF migration is a critical requirement in NFV systems. Even with the help of the LEMON loader, it is still unclear for the migration manager to migrate the LEMONs to other workers, SFCs or even servers. The key challenge here is that the LEMON loader can only handle a LEMON as a whole, while the inner data structures (*i.e.*, NF states) of a LEMON might need to be *partially* migrated to another core or re-accessed in a different process for intra- or inter-server load balancing.

4 Detailed Design of LemonNFV

In this section, we discuss in details how LEMONs are loaded (§4.1), scheduled (§4.2), isolated (4.3) and migrated (§4.4) using the unified abstractions of LEMON.

4.1 Loading the LEMONs

Given an ELF binary, the loader is responsible to allocate the memory for the executable, copy the segments to the corresponding memory regions, resolve the external symbols, and finally call the constructors. While Linux has offered a mature toolchain for loading the ELF file at runtime, *i.e.*, `dl`-family functions, they are ill-suited for serving as the LEMON loader. In the following, we explain the specific requirements when loading a LEMON, and present our solution for the LEMON loader.

Dependency isolation. The OS loader, *i.e.*, `ld.so` with `dlopen`, attempts to reuse the libraries that have been already loaded for saving the memory and improving instruction cache affinity. However, this would cause dirty access of common libraries if multiple LEMONs depend on them.

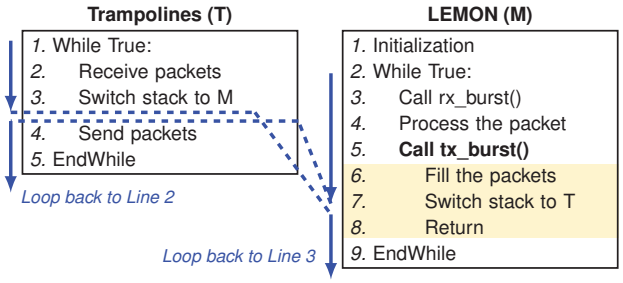
In LemonNFV, the LEMON loader views each NF and its dependencies as a sandbox, and will load the dependencies no matter whether other LEMONs have already loaded them. For example, each LEMON will load its own `libc`, such that they will not share the global variables like `optarg`. so that

Targeted symbol resolution. While loading, lots of functions in LEMONs should be intercepted to the customized versions, *e.g.*, the DPDK I/O to the schedulable I/O, the native `malloc` to the restricted `malloc`. The common solution to this task is `LD_PRELOAD`, which instructs the loader to first lookup the preload libraries when resolving every symbol of the executable. However, `LD_PRELOAD` redirects *all* symbols with the same name, while LEMONs and trampolines should use different versions. For example, trampolines call `rx_burst` to receive packets from NIC, which should not be resolved to the schedulable version as in LEMONs. Besides, LEMONs may depend on different versions of the libraries, and the intercepted functions with the same name also need vary to those semantics.

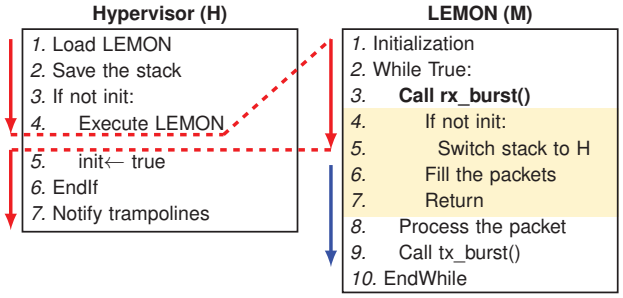
LemonNFV implements a symbol resolution mechanism tailored for loading LEMONs, which allows the trampolines and each LEMON to specify their own preload libraries, enabling different semantics for symbols of the same name.

Consistent loading address. The Linux loader cannot manually specify the loading address. Instead, the recent Linux kernels enable the random address loading, *e.g.*, ASLR [1], mostly due to the security reason. Such random loading would disable the ability of reloading a LEMON, because all pointers in the reloaded LEMON would become invalid. This feature is critical for fault recovery and LEMON migration.

To this end, the LemonNFV process reserves the same virtual address space, which is partitioned into fixed-size *slots*. the LEMON loader would load each LEMON to its *unique* slot, and allocate the fixed address regions for the private stack, heap and dependencies. As a result, all the pointers (expect packet pointers) in a LEMON snapshot would remain valid even being reloaded or migrated to another process.



(a) Scheduling between the trampolines and the LEMON.



(b) Initialization with the hypervisor.

Figure 4: Scheduling LEMONs with the schedulable I/O and private stacks. Solid arrows indicate the executing paths, and dashed lines are the function/stack transitions. The blue paths are executed by the working thread, and the red ones are from the hypervisor thread. Shadowed texts are the pseudocode of `tx_burst` and `rx_burst`.

4.2 Scheduling the LEMONs

A typical NF implementation consists of four stages: (1) the NF initializes its own data structures and hardware; then it starts an infinite loop which (2) receives packets using a function like `rx_burst(pkts)`; (3) processes the packets; and (4) sends the packets out using a function like `tx_burst(pkts)`. The LEMON I/O interfaces unify the way for LEMONs to fetch and send packets, *i.e.*, stage 2 and stage 4. Specifically, the LemonNFV trampolines are responsible to communicate with the hardware NIC. And the "NIC" in the LEMON is actually a memory region that stores the packets. As a result, `rx_burst` should fill `pkts` (the pointers of packets) with the packets from the trampolines; and `tx_burst` should fill `pkts` back to trampolines for the downstream LEMONs.

Having those basic I/O operations, how to schedule the LEMONs such that the packets can flow through them as if they were chained together will be our focus here.

Scheduling LEMONs with schedulable I/O. Each NF has its own control flow, from `main` function to the infinite loop of packet processing. The LEMON scheduler needs to cooperate with those control flows to properly jump into and out from the LEMON execution.

To this end, LemonNFV creates private stack for each LEMON, making its control flow separated from the trampolines and one another. Specifically, LemonNFV allocates a dedicated memory region for each LEMON as its stack, and

the trampolines maintain the corresponding stack pointers (*i.e.*, `SP` and `BP` registers). In this way, the LEMON scheduling can be simply implemented as saving the current states of registers (stack pointers and other callee-saved registers) and restoring the previously saved ones of the target LEMON. This process operates purely in the user space, thus incurs much less overhead than the context switch between processes or threads, which would trap into kernel and flush TLBs. We implement the above stack switch logic in the packet sending functions, because each time the NF is sending the current batch of packets out of the NIC, the control flow should move to the next NF in the SFC.

We use Figure 4a to illustrate how the scheduling works. Assume the SFC has only one LEMON, which has processed a batch of packets and is sending them out, *i.e.*, calling (the schedulable) `tx_burst` in Line 5 of M. After filling the packets, the execution stack is saved and switched to the trampolines (from Line 7 of M to Line 3 of T). The trampolines then send the packets to the NIC (Line 4 of T). In the next loop it receives a new batch of packets (Line 2 of T) and schedules the LEMON again (Line 3 of T), which will jump back to Line 7 of M. The LEMON will continue to receive and process the packets from the trampolines, *i.e.*, Line 8–9, 3–4 of M, until it is scheduled out, *i.e.*, `tx_burst` is called again. Any additional logic besides Line 4 (*e.g.*, profiling after sending packets) will also be executed at this moment.

Handling the logic other than packet processing. The hypervisor needs to take care of the LEMON initialization, which, like the packet processing logic, has no explicit boundary in the NF's code. To this end, we view the initialization as the logics from the first line of `main` function to the *very first time* the `rx_burst` is called, which means all preparations for packet processing have been done. We use Figure 4b to depict such process. The hypervisor thread (red path) loads the LEMON, saves its stack and executes the LEMON (Line 1–4 of H). After LEMON is initialized (Line 1 of M), it will eventually call (the virtualized) `rx_burst` (Line 3 of M). For the very first time it is called (*i.e.*, `init==0`), the LEMON should switch back to hypervisor's logic, and the hypervisor will notify the trampolines that the initialization is done. The trampolines will execute the LEMON from the saved stack (Line 6 of M) next time it is scheduled.

Except for the initialization, NFs may also include logic for event logging and runtime configuration. These routines are usually conducted in individual threads. See Appendix A for scheduling a LEMON with such threads.

Complex I/O behaviors. The above scheduling assumes the NFs call the packet I/O in a synchronized way, *i.e.*, receiving (Rx) and sending (Tx) once per batch, while NFs can also invoke less Rx and more Tx (*e.g.*, multicast), or more Rx and less Tx (*e.g.*, packet buffering). The current scheduling is based on Tx, thus can still handle the former case, and we extend the virtualized Rx I/O to deal with the latter. To be specific, we add a flag in virtualized Rx function to check whether it is called *for the first time in this batch*. If not, the Tx function is not called, which means this LEMON buffers or drops all packets, and blocks the downstream LEMONs. In such case, the trampolines should continue to receive the next batch instead of scheduling the next LEMON, ensuring correctness of NFs that buffer packets (*e.g.*, Reframer [28]).

4.3 Isolating the LEMONs

LemonNFV provides each LEMON with a separate memory domain via the custom allocators, so that the legal region a LEMON can access is bounded. However, achieving this is far from sufficient to isolate each LEMON from each other, because the illegal accesses are only *defined* but not *prevented*. We now discuss how LemonNFV checks illegal memory accesses efficiently. In the following, we first present the threat model of LEMON isolation, then introduce two *software fault isolation (SFI)* techniques to sandbox the memory accesses. After making our design choice, we present how LemonNFV realizes the LEMON isolation in runtime.

Threat model. LemonNFV isolates the SFCs from different tenants with processes. For each process, LemonNFV allocates two virtual functions (VFs), *i.e.*, NICs virtualized by SR-IOV [15], which provide almost the same performance compared to the physical NIC. Given such strict isolation between SFCs, we only need to take care the isolation between LEMONs along the SFC, *i.e.*, the intra-process isolation.

We assume each NF has its own packet processing logic, which are expected to be independent of the others. That is, even in the same process/thread, the data and states of NFs are strictly independent. This assumption disables most communications between NFs, and thus is weaker than NetBricks's threat model that supports state sharing [52]. However, we argue that this assumption is actually aligned with the case when chaining physical middleboxes, where the internal states are unavailable to external NFs, and packets are the only information that can be exchanged between NFs.

We assume the trampolines and hypervisor are written with care, while NFs are written in a negligent way that they might illegally modify the data, *e.g.*, overwrite the state in other NFs or hypervisor. Such bad operations can happen in either NF itself, *e.g.*, `*(bad_addr)=1`, or the libraries it depends, *e.g.*, `memset(good_addr,0,bad_size)`.

Two design options. The above threat model falls into the SFI area, where the most classical implementation is to check the bound of each memory write [38,67,71], *e.g.*, `*p=1` would be modified into `if(p>L&&p<H) *p=1` to ensure `p` is always within its own memory region `(L,H)`. Its performance is determined by (1) the number of the legal regions, since each instrumentation must check all these regions, and (2) the number of memory access statements, which equals to the number of instrumentation.

Recent studies advocate to leverage hardware-aided techniques to realize SFI [31,45,66]. To be specific, they partition the memory into domains, and a certain software module can only access its legal domains. When switching to another software module, this method has to change the legality of the domains. As such, its runtime overhead is mainly determined by the number of domains. Another defect of domain switching is that it restricts the number of domains due to the limitation of hardware.

Our design decision. Bounds checking would incur unacceptable runtime overhead for realizing LEMON isolation for the following reasons. First, the legal regions of a LEMON, *e.g.*, code, stack and heap, are separate, making each instrumentation quite costly. Second, the number of memory access statements could be large for NFs like encryption, which further lowers the performance. To make the performance penalty clear, we implement an extra compiling pass in LLVM that injects bound checking before every memory write at the IR level. We then prototype three typical NFs including NAT, firewall and IDS, and by chaining them into an SFC, we find that the isolated SFC is in average 24% slower than the original one. Such penalty is usually unacceptable for realizing the wire-speed processing.

On the contrary, we find domain switching is quite suitable for LEMON isolation. Recall the threat model that LEMONs do not share states between each other. That is, each LEMON can be packaged into a disjoint domain, such that the number of domain switching is actually the length of SFC, which is fixed and stable no matter how complex a LEMON is. For the limitation on number of domains, it should still be sufficient for LEMON isolation, since the length of SFC is usually small, *i.e.*, less than 10 for most real-world cases [4]. As a result, the domain switching approach is preferred to realize SFI between LEMONs.

Among the others like VMFUNC that require code modification [38,45], PKU has been viewed as the fastest domain switching approach and requires little modification on existing code [31,66]. PKU uses the spare four bits in each page table entry to partition memory into 16 domains, and specifies the access restrictions for each domain by a *pkey*. In doing so, the protection can be naturally guaranteed when accessing the page table, and thus incurs *zero extra cost* in runtime. More importantly, the operation of domain-switching, *i.e.*, writing the permissions to pkeys, purely works in userspace and only incurs negligible overhead, *i.e.*, less than 100 cycles.

Isolating LEMONs with PKU. Inside the process, LemonNFV specifies one pkey for each LEMON. In runtime, when a LEMON is scheduled by the trampolines, LemonNFV enables its pkey to grant the access rights to its own domain, *i.e.*, its own stack, heap and data segments, and disables the access to any other domain. Since the switching happens when scheduling LEMONs, the switch logic is embedded into the schedulable I/O for each LEMON.

Consider a simple example with two LEMONs, M1 and M2. The trampolines' domain occupies pkey0, and each LEMON (as well as its stack, heap and dependencies), is allocated with one pkey, *i.e.*, pkey1 and pkey2, respectively. The packet domain is always readable/writable to trampolines and LEMONs, and thus does not need a specific pkey to protect. At runtime, when trampolines are receiving or sending packets, all three pkeys are enabled, because trampolines have the full visibility to all domains. Before going into M1, trampolines would disable pkey0 and pkey2. After M1's processing, the schedulable I/O in M1 would enable pkey2 for M2's processing. After the processing of the last LEMON, *i.e.*, M2, its schedulable I/O will disable pkey2 and enable pkey0 to jump back to trampolines. Note that to reduce the domain switching, M1 would directly jump to M2 instead of jumping back to the trampolines. In general, LemonNFV would switch $N+1$ times for an SFC with N NFs.

4.4 Migrating the LEMONs

NF migration is essential when operators seek a balanced load and/or efficient resource utilization. Existing works realize this feature on top of a fully supervised infrastructure, *e.g.*, OS-level virtualization [12, 44], or through specific migration interfaces, *e.g.*, OpenNF [27]. Without above prerequisites, LemonNFV leverages the standard LEMON binary to empower users to migrate the LEMONs to other cores and machines in an efficient way [2].

Intra-process LEMON migration. In consolidation approach, it is important that packets should be evenly dispatched to the workers, otherwise the CPU resources are wasted [35]. Due to the dynamics in network (*e.g.*, traffic burst) and resources (*e.g.*, adding a core), such balance is hard to achieve if statically binding packet classes (*i.e.*, a set of flows) to cores. Instead, the NFV framework needs to migrate the packet classes *and* the corresponding NF states to a new core, to balance the working load.

LemonNFV addresses this challenge by relieving the close binding between cores and LEMONs, making the consistent migration possible for any state structures used in LEMONs. To be specific, LemonNFV creates a pool of LEMONs, each of which is dedicated for a packet class. Given the simple fact that both executable code and states are within the LEMON, "migrating the state of packet class P to core C" becomes "letting core C execute the LEMON corresponding to P".

Figure 5 shows a simple migration scenario, where the operator allocates two cores (C_1 and C_2) to handle three packet classes (P_1–P_3). In this case, LemonNFV will create three LEMONs (M_1–M_3) dedicated for P_1–P_3, and ensure that cores will always handle the packet classes with their corresponding LEMONs. Assume we want to migrate P_2 to C_2. The

[2]The proposed migration schemes are for common load balancing scenarios, while specific migration cases, *e.g.*, migrating an arbitrary flow, balancing a non-splittable flow, are not supported. See §7 for a discussion.

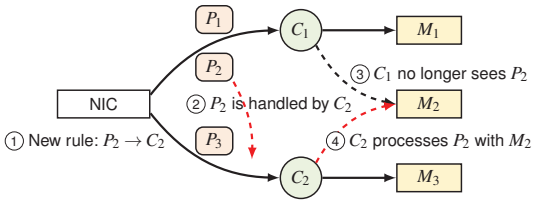


Figure 5: Intra-server LEMON migration. C_1 and C_2 are cores, P_1–P_3 are three packet classes, and M_1–M_3 are three identical LEMONs dedicated for P_1–P_3.

hypervisor first issues a new rule to the NIC (or modifies the global RSS table [16]), which directs the NIC to tag and send P_2 to the queue binding to C_2 (①). At this point, all packets belonging to P_2 will be handled by C_2 (②), and C_1 cannot see P_2's packets immediately (③). When C_2 receives the packets of P_2, it can safely process them with M_2 (④). Since the states of P_2 are always within M_2, there is no need to synchronize or copy states between cores. However, since each core has a receive queue, there is a minor risk that C_1 still holds a few packets of P_2 after the migration. For this case, we could create a receive queue for each packet class, and let different cores to handle the queue while migration, which would cost little performance [16].

②–④ are natural consequences of ①, meaning that the overhead of migration is essentially the time of rule installation, which only amounts to sub-milliseconds.

Inter-process LEMON migration. Besides the intra-server load balancing, the network-wide load balancing calls for migrating LEMONs across different LemonNFV servers (processes) in runtime. This is done by (1) allocating identical addresses for LEMONs across processes and (2) migrating the snapshots (memory dump) of LEMONs iteratively.

For the first, the LEMON loader has ensured the address consistency across different processes. One concern here is that can a single process provide enough address space, if considering different types of NFs and per-packet-class pool of LEMONs. In fact, modern 64-bit system can easily reserve 96TB of virtual memory (*e.g.*, 0x100000000000 - 0x700000000000) to support 32768 LEMON instances (6GB memory each), *i.e.*, 128 types of NFs with 256 packet classes, which are sufficient given limited number of popular NFs. Note that this restriction is for the unique LEMONs in *network-wide* servers, and the length of SFC in a single server is still bounded by the physical memory and PKU limitation.

Secondly, LemonNFV needs to realize a packet-lossless migration. In runtime migration, the snapshot is taken after the LEMON loosing references over a batch of packets, and is transmitted to its destination. The LEMON loader then loads it into the corresponding slot and modifies the SFC. Since all states are within the LEMON, the migration will not lose any state. However, the cross-machine transmission might be time-consuming due to the large snapshot, resulting in packet losses to the LEMON.

To this end, LemonNFV iteratively copies the snapshot [12]. Specifically, in phase -1, LemonNFV pre-copies the whole snapshot to the target, and uses dirty page tracking [8] to locate that a portion of memory d becomes dirty through this phase. Then in phase 0, LemonNFV only transmits the dirty bytes, and locates the new dirty memory d. When d is getting smaller, the transmission time also decreases, which further reduces d in the next phase. Iteratively, the dirty memory transmission can finish within a negligible time, which is the right timing to route the packets to the new machine. See Appendix B for a detailed implementation.

5 Implementation

We implement a prototype of LemonNFV with 5K lines of C code, including the unified abstractions, *i.e.*, the schedulable I/O and the restricted allocator, and the system components, *i.e.*, the LEMON loader, the LEMON scheduler and the LEMON isolator with PKU. We highlight several key implementations and enhancements in our prototype.

Schedulable I/O. The prototype implements a schedulable version for all I/O functions used by NFs in §6, which includes 28 libpcap [7] and 41 DPDK [5] functions. The user can easily intercept I/O interfaces not included in the prototype by adding functions to the preload libraries when involving new NFs.

Restricted allocator. As mentioned in §3.2, each LEMON should notify its memory budget to LemonNFV. This is to avoid the runtime fault like memory leak: if the customized memory allocator detects that a LEMON exceeds its memory limitation, LemonNFV can unload it before it drains all host memory. Besides, this also enables a simple but effective optimization, *i.e.*, memory pre-allocation, which can largely improve the memory performance in runtime for memory-intensive NFs. To be specific, when a LEMON requires a certain portion of memory, the customized memory allocator pre-allocates all memory in the heap, so that it does not need to make expensive syscalls like `mmap` in runtime.

Fault isolation. §4.3 mainly considers the memory isolation, while the *fault isolation* is also critical for consolidation approaches. Since all NFs are in the same process, a runtime fault, *e.g.*, divided by zero, of a single NF would fail the whole chain. This task is addressable with the help of the trampolines in LemonNFV. First, LemonNFV registers a set of signal handlers for dealing with the runtime fault like `SIGABRT`, `SIGSEGV` and `SIGILL`. Then, once those signals are captured, the trampoline can decide how to prevent the faulty LEMONs from impacting others. In the prototype we simply remove the faulty one and pass the packets to downstream LEMONs. Other policies like restoring a LEMON to its nearest checkpoint can also be implemented based on the above fault capture scheme.

6 Evaluation

In this section, we evaluate the practicality, performance and overhead of LemonNFV with real and synthesized NFs. We are particularly interested in the following questions.

1) Can LemonNFV consolidate heterogeneous NFs to obtain high performance without much coding effort? Experiments show that LemonNFV can consolidate real NFs from different frameworks as the virtualization approach, *i.e.*, *without modifying native code*, and achieve a high-performance SFC as the consolidation approach, *i.e.*, *incurring minor overhead between NFs* (§6.2).
2) Can LemonNFV outperform state-of-the-art NFV systems? Experiments show that LemonNFV is *1.9–2.4*× faster than a state-of-the-art virtualization approach, and incurs only *0.7–4.3%* overhead compared to a state-of-the-art consolidation approach (§6.3).

6.1 Experimental Setup

Testbed. Our testbed is an x86 machine (24×Intel Xeon 3Ghz, 256GB memory) equipped with a Mellanox CX-6 DX NIC (2-port 100Gbps). Hyperthreading and frequency boosting are disabled in all CPUs, and the host OS is Ubuntu 20.04 with Linux kernel 5.11. We use DPDK 21.05 as the packet I/O for LemonNFV with 32 as the batch size, and enable S-RSS in multi-core experiments.

We prepare two traces for testing: ISP trace from a large service provider that contains majorly TCP flows (8.6M packets, 3.9M active flows, 652 bytes in average) and ENT trace captured from an enterprise network which mostly consists of GTP (UDP) packets (11.2M packets, 462 bytes in average).

Another server with the same NIC replays the traces to the testbed at line rate, serving as the packet generator. The testbed is configured to forward all traffic back to the generator, no matter whether NFs drop them or not. We run each experiment under 100Gbps traffic and record the average value of 20 seconds. Each experiment is then repeated 10 times and the average result is reported.

Real NFs. We consider NFs built upon the fast userspace I/O (DPDK) and kernel I/O (`libpcap`). We involve three DPDK NFs: an IDS based on Rubik [41] that matches the reassembled payload with snort-like rules; a NAT based on FastClick [17] that is composed of many inherent elements like traffic classifier and ARP querier; and an ACL based on NetBricks [52]. We further involve two `libpcap` NFs: a connection tracker (CT) based on mOS [34] that tracks the status of TCP connections; and a DPI tool based on nDPI [9] that can identify 170+ L7 applications.

Synthesized NFs. We further choose several synthesized NFs which are feasible to be ported, such that we can fairly compare them under different frameworks. We use NFD [32] to produce four stateful NFs: network address port translator, heavy hitter detector, super spreader detector, and UDP flood mitigation. We further extract four NFs from OpenNetVM's repository [76], including payload scanning, stateless firewall, AES encryption and decryption, which are stateless but computing-intensive.

Table 1: The real NFs and the efforts for interoperation.

NF	Framework	Lang.	I/O	CN	CF	CH
IDS	Rubik [41]	C	DPDK	337	31K	2
NAT	FastClick [17]	C++	DPDK	94	331K	2
ACL	NetBricks [52]	Rust	DPDK	401	58K	8
CT	mOS [34]	C	`libpcap`	325	139K	4
DPI	nDPI [9]	C	`libpcap`	4498	121K	2

CN: *the LOC of the NF logic*, ***CF:*** *the LOC of the framework*
CH: *the LOC modified by LemonNFV for interoperation*

6.2 Comparing with the Ideals

When interoperating, *i.e.*, chaining, isolating and managing, heterogeneous NFs, we have two ultimate goals: without code modification *and* performance penalty. To this end, we use the real NFs to compare LemonNFV with two ideals, *i.e.*, a virtualization approach that does not modify a single line of code, and a pure consolidation approach that does not incur any performance penalty.

Efforts of interoperating NFs. As shown in Table 1, the real NFs are heterogeneous in many ways including the language, I/O, *etc*. We assume the virtualization approach emulates a full running environment, *e.g.*, with VM or Docker, and thus does not need to modify the real NFs for chaining them. On the other hand, the consolidation approach needs to extract or wrap the NF logic for interoperation.

Intuitively, it would only cost limited coding efforts, since the high-level NF logic is relatively simple and neat, as shown in CN in the table. However, the factual task would cost much more than that number. For example, to implement an NAT in FastClick only needs to write 94 LOC for a Click script, while to interoperate this NF with CT in mOS, one has to dig into the detailed implementation in FastClick *and* mOS, to ensure they have the same packet abstractions, are not conflicting in variable/function names, and do not incur dirty writes on global data structures, *etc*. Things get more complex when porting NFs with different languages, *e.g.*, rewriting a Rust NF into a C/C++ one. Generally, the effort of code reading and writing for consolidating heterogeneous NFs would approach to the numbers shown in CF.

LemonNFV does not modify a single line of native code (*e.g.*, C/C++, Rust) for consolidating these NFs. The only effort we made is to modify the compilation configurations of the NFs, *e.g.*, adding `-fPIC` and `-shared` in `Makefile`, to compile the NF as a shared library instead of an executable, which amounts to a handful of LOC. We also feed the command line arguments, *i.e.*, `argv`, to each LEMON with a configuration file, which also contains minor LOC. CH in Table 1 shows the total number of modified LOC.

Table 2: Performance comparison over real NFs.

	Per-Packet Latency (us)			Throughput (Gbps)		
	Ideal	Docker	LemonNFV	Ideal	Docker	LemonNFV
ISP	0.66	240 (+362×)	0.68 (+3.8%)	22.9	11.6 (-49%)	22.0 (-3.8%)
ENT	0.59	224 (+223×)	0.60 (+2.8%)	18.3	12.7 (-31%)	17.7 (-3.2%)

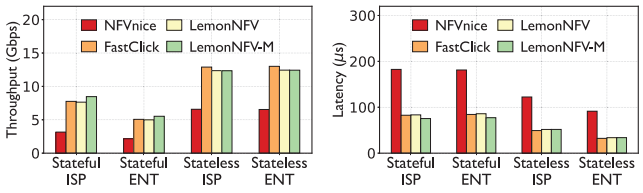
Performance comparison. We chain the real NFs to conduct a sequential SFC: IDS→NAT→ACL. We do not involve the two `libpcap` NFs here because the slow packet I/O would significantly enlarge the performance gap between virtualization and other approaches. As a reference, CT and DPI with LemonNFV are 5.6× and 3× faster than their `libpcap` versions respectively. We consider two performance metrics: the processing time of each packet, *i.e.*, the elapsed time after it entering the first NF and before it leaving the last NF, and the end-to-end throughput.

For virtualization approach, we use Docker containers to carry the NFs. Each container is equipped with one CPU core and two virtual NICs, and the SR-IOV [15] technique ensures the line-rate packet forwarding between Dockers without CPU interventions. Due to the large codebase of real NFs, we do not implement a pure consolidation approach. Instead, we estimate its performance as follows: for the per-packet latency, we simply accumulate the processing time of each single NF; for the throughput, we set up a basic DPDK I/O, and delay each packet for the accumulated processing latency. This should present the performance upper bound of consolidation since interference between NFs (*e.g.*, cache and memory contention) is removed.

Table 2 shows the performance comparison. It can be seen that the Docker approach adds more than 200× latency compared to the Ideal approach. Since those NFs are with high-speed I/O, the major overhead comes from the packet pool in the virtual NIC and DPDK, *i.e.*, *V2* mentioned in §2.1. On the other hand, LemonNFV eliminates such overhead and only incurs 2.8%–3.8% overhead, largely resulting from the PKU switch between NFs.

Since Docker uses three separate cores, we also scale Ideal and LemonNFV with three cores for throughput comparison. The results show that Docker is in average 40% slower than Ideal, while the overhead of LemonNFV is just 3.5%. The performance gap between Docker and LemonNFV is narrowed compared to the per-packet latency because the end-to-end measurement includes the I/O latency between the testbed and the packet generator (∼100us). Such gap would again enlarge with longer SFC.

Loading and migration. Under LemonNFV, loading a LEMON is essentially loading a shared library, which is fast and predictable. We measure the loading time of all 5 real NFs, and report that the max/average loading time is 28ms/7ms. Note that LEMONs are loaded by the hypervisor in an asynchronous way, so the factual overhead for the SFC can be neglected. As a comparison, booting a container usually takes hundreds of milliseconds [47].



(a) Throughput (b) Latency

Figure 6: Performance comparison with different SFCs and traces using 4 cores. LemonNFV-M indicates LemonNFV with memory pre-allocation enabled.

We also verify that our inter-process migration can be finished rapidly. We setup two processes on a single server, each of which carries a naive packet forwarding LEMON. The source process then loads IDS in Table 1, runs it for some time, and migrates it to the destination process. Results show that iterative migration copies over 400 dirty pages within 1.2s, and efficiently reduces the total downtime to only 6.9ms.

6.3 Comparing with Existing NFV Systems

We compare LemonNFV with state-of-the-art NFV frameworks. For obtaining higher performance, these frameworks often require developing NFs using specific interfaces. To this end, we port the synthensized NFs to these frameworks and compose two SFCs: Stateful that chains the four stateful NFs from NFD, and Stateless that chains the four OpenNetVM NFs. Performance is the focus of this comparison.

Comparing with NFVnice. We port the synthesized NFs into NFVnice [40], a state-of-the-art virtualization-based approach that deploys NFs inside processes and enables back-pressure between them to minimize wasted work. NFVnice sets a large packet queue (2^{15}) between each NF for higher throughput and less packet loss, which would in turn increase the latency. To this end, we measure the throughput with the default setting, and measure the latency by reducing the queue size to 32 (default batch size). This could represent the ideal performance a virtualization approach can achieve.

Figure 6 shows the comparison of NFVnice and LemonNFV. LemonNFV is at least 88% faster than NFVnice with 55% less latency. Note that besides the four worker cores, NFVnice employs two extra cores dedicated for packet I/O, which, however, has been heavily hindered by the context switches and cache misses when traversing SFCs.

Comparing with FastClick. FastClick [17] is an enhanced version of Click [37] with high-speed I/O, optimized compilation stages, and many useful elements. FastClick is the basis of many state-of-the-art consolidation approaches like Metron [35], RSS++ [16], PacketMill [25] and Reframer [28]. In FastClick, each NF is a C++ object, and chaining NFs is just calling a function of the object. This implementation eliminates all extra overhead between NFs, indicating the ideal performance of a consolidation approach.

Figure 6 shows that LemonNFV is only 1.5% slower than

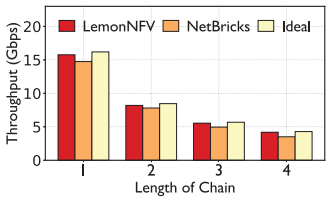


Figure 7: Throughput comparison with NetBricks.

FastClick for Stateful. This demonstrates the end-to-end overhead employed by LemonNFV, including the stack switch, the pkey set, *etc*. The overhead increases to 4.2% for Stateless, because this SFC is more lightweight, making the cost from LemonNFV more significant.

After enabling the pre-allocation, LemonNFV-M even outperforms FastClick by 9.2% in Stateful. This is because NFs in Stateful are memory-access-intensive, such that the pre-allocation would improve the performance significantly. For the similar reason, Stateless performs almost the same with and without this feature.

Comparing with NetBricks. The closest work to ours is NetBricks [52], which also runs SFC in a single thread and provides isolation among NFs. To compare the runtime overhead when hosting NFs, we implement a simple ACL NF using C in LemonNFV and Rust in NetBricks. This NF does not use any complex algorithms and data structures, *e.g.*, hash table, from the standard library, to minimize the performance difference from the language. We chain this NF for several times to measure the scalability, and further involve the pure consolidation, *i.e.*, chaining the C NF by function call, as the ideal approach.

Figure 7 shows that LemonNFV is on average 2.6% slower than Ideal, while NetBricks incurs 11.9% overhead. With a longer chain, the overhead of LemonNFV is stable (always ∼2.5%), but NetBricks incurs larger overhead (from 8.9% to 18.1%). This is because the overhead of NetBricks mainly comes from the NF logic itself, *i.e.*, checking array bounds with Rust (reportedly a 14% overhead for an LPM [52]), while the overhead of LemonNFV is irrelevant to the NF logic. As a result, though the increasing chain length also increases the number of domain switching, such overhead is still negligible compared to the workload of NF itself. Considering that the example NF is largely simplified, the factual gap between LemonNFV and NetBricks would be much significant in real cases.

Overhead analysis. We quantify the overhead employed by LemonNFV to better understand the performance illustrated above. For each LEMON switching, LemonNFV incurs the following overhead: (O1) switching the memory domain by writing the pkey registers, (O2) switching the private heap, and (O3) switching the private stack to the target domain.

For O1, each domain switching invokes one write to the pkey register, which averages to 82 cycles in both SFCs. O2 is stable and minor (*i.e.*, 9 cycles), because heap switching only changes the base pointer of the heap. For O3, stack switching needs to save the current context and restore the target. We measure such overhead for each NF switching in Stateful and Stateless and the result averages to only 31 cycles. Note that above overhead is for a whole batch, meaning that the per-packet overhead with default batch size (32) is only ∼4 cycles for each switching.

As a comparison, virtualization frameworks would be impeded by context switching, cache misses and TLB flushes. To the best of our knowledge, Quadrant [68] has the least overhead by pinning the NF instances on the chain to a single core, which results in a per-packet overhead of ∼110 cycles. In Stateless with 4 NFs, it will cost 530 more cycles than LemonNFV ($(110-4)\times 5$ switching). As the per-packet latency of Stateless is ∼4000 cycles in LemonNFV, Quadrant is thus ∼13% slower in terms of the isolation overhead.

7 Related Work and Limitations

We discuss efforts that relate to or inspire LemonNFV.

NF development. FastClick [17] and BESS [30] can flexibly wire their elements/modules up to the SFC (in an off-line manner). However, those modules must be programmed with specified interfaces, *e.g.*, being inherited from certain base classes. To reduce the overhead between NFs, NetBricks [52] proposes to use a new language, Rust, with a set of domain-specific abstractions for building NFs. However, it is costly to re-implement all existing NFs using such new language. NFD [32] can generate NF code from a high-level behavior model. While this facilitates the porting task, NFD does not make heterogeneous NFs interoperable, and operators are still bound to a specific interface. Nethuns [18] advocates a socket-independent programming abstraction for NFs, trying to unifiying the packet I/O, but in turn proposing a new I/O.

Moreover, with LemonNFV, one can directly launch a NF with out-dated I/O, *e.g.*, official NetBricks with DPDK 17.08 [11], or develop a new NF with the simple packet I/O, *e.g.*, `libpcap`, and get the newest and fast I/O for free.

NF optimizations. OpenBox [19] and SNF [36] optimize the SFC by eliminating the redundancy logics in NFs. Metron [35] goes a step forward by offloading part of the merged logic to programmable switches. PacketMill [25] and Morpheus [50] compiles optimal data structures and control flows for NFs. These optimizations require inner logics of NFs, and thus cannot be borrowed by LemonNFV that views each NF as an opaque box. We see it as an inherent trade-off between the reusability and deep optimizations.

NF execution models. Virtualization approaches are eager to improve the performance [22, 33, 40, 51, 76]. Exploiting the parallelism of NFs is viewed to be a promising direction [64, 77]. However, according to its own results, the parallel approach (NFP [64]) is 16.5% faster than the non-parallel one (OpenNetVM [76]), but 26% slower than the RTC approach (BESS [30]). The factual gap should be larger be-

cause BESS has reached the wire-speed in that results. Quadrant [68] packages NFs into containers and schedules them in a single core to avoid the cache misses, which, however, would cause the thread context switch.

FastClick [17], BESS [30], OpenBox [19] and Metron [35] are pure consolidation approaches, which achieve high performance without isolation. NetBricks [52] is the first consolidation approach that considers the isolation. By reimplementing NFs with a safe language (Rust), NetBricks ensures each instruction cannot access the data beyond its legal scope. However, such operation in Rust is too strict, lowering ∼20% throughput compared to native consolidation approaches [52].

Intra-process isolation. Hodor [31], ERIM [66], EPK [29] and CubicleOS [59] leverage Intel PKU to isolate software modules inside the process. LemonNFV is inspired by them and is the first for using PKU to isolate NFs in SFC. However, NFs can change pkey privileges themselves and thus break the control flow integrity or subvert the hypervisor. We see this as the nature of PKU and can leverage methods discussed in existing works to harden PKU-based isolation [20].

Besides memory and fault isolation, previous work has also proposed and enforced packet isolation, which prevents NFs from accessing packets that don't belong to them. Specifically, an NF may tamper packets outside its batch since the packet pool is shared by all NFs, contradicting the isolation among them. To solve this, NetBricks [52] leverages safe language to disable pointer arithmetic, and Quadrant [68] copies packets to cast packets on memory private to each NF. Similarly, LemonNFV can either limit permission on the exact batch using pkeys, or simply copy packets to address this problem.

NF migration. NFV frameworks need to migrate NFs and their states to balance the load across the cores or machines. Previous literatures achieve this feature by adding migration APIs [27, 55], copying states with identical state structures [35], or using centralized state tables [16], all of which require significant code modification to NFs. LemonNFV addresses this need by migrating the LEMON that packages the NF logic and code together. However, each LEMON corresponds to a certain packet class, which means LemonNFV cannot migrate a specific flow [27], or balance the load for a single large (*i.e.*, non-splittable) flow [16].

Limitations. The design of LEMON and LemonNFV does not meet every possible situation in deployment. (1) NFs that do not run as a standalone process (*e.g.*, NFs based on eBPF [3] or partially offloaded to hardware [35]) are not supported due to their fundamental deviation from a LEMON's execution model. (2) LemonNFV enables fast inter-server migration by disabling ASLR, which might be exploited by buffer overflow. Nevertheless, the operator can still migrate the whole LemonNFV process with ASLR enabled by leveraging checkpoint/restore methods (*e.g.*, `CRIU` [2]), when consolidating untrusted NFs.

LemonNFV requires recompiling NFs from the source code, which is not always available for off-the-shelf NFs. However, we believe it does not compromise much practicality of LemonNFV because the recompilation (1) still uses the standard compiler (*e.g.*, `gcc`), not raising concerns of security and inconsistency, and (2) only recompiles the main program, meaning that the original dependencies can be reused. These facts help the NF vendors to re-publish their NFs as LEMONs with a simple recompilation, and the users can directly plug them into LemonNFV.

We emphasize that LemonNFV does not aim to address *all* challenges in NFV. Instead, it tries to shed the light for the NFV world by enabling the ability of heterogeneous NF interoperation. On such basis, approaches like centralized state management [70], NF fault recovery [62], load balancing [16, 58], VNF placement and orchestration [24, 26], *etc*, could be complementary to LemonNFV.

8 Conclusion

We presented LemonNFV, a novel NFV framework that consolidates the heterogeneous NFs without code modification. We demonstrated the practicality of LemonNFV with 5 real NFs, and evaluated LemonNFV by comparing with state-of-the-arts. The results showed that LemonNFV outperforms the state-of-the-art virtualization approach by 1.9–2.4×, while only sacrifices 0.7–4.3% performance for isolation compared to the ideal consolidation framework.

Acknowledgements. We would like to thank the anonymous NSDI reviewers and our shepherd Dejan Kostic for their valuable feedback. Peng Zhang is the corresponding author. This work is partially supported by the National Key Research and Development Program of China (No. 2022YFB2901403) and the National Natural Science Foundation of China (No. 62172323 and No. 62272382).

References

[1] Address space layout randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[2] Checkpoint/restore in userspace. https://criu.org/.

[3] eBPF - Introduction, Tutorials & Community Resources. https://ebpf.io/.

[4] Service Function Chaining Use Cases In Data Centers. https://datatracker.ietf.org/doc/html/draft-ietf-sfc-dc-use-cases-06, 2017.

[5] DPDK. http://www.dpdk.org/, 2018.

[6] Libnids. http://libnids.sourceforge.net/, 2018.

[7] libpcap. http://www.tcpdump.org/, 2018.

[8] mm: Ability to monitor task memory changes (v3). https://lwn.net/Articles/546966/, 2018.

[9] nDPI. https://bit.ly/3ITdis5, 2018.

[10] Snort. https://www.snort.org/, 2018.

[11] NetSys/NetBricks. https://github.com/NetSys/NetBricks, 2019.

[12] The vMotion Process Under the Hood. https://blogs.vmware.com/vsphere/2019/07/the-vmotion-process-under-the-hood.html, 2019.

[13] Passive Real-time Asset Detection System. https://github.com/gamelinux/prads, 2020.

[14] Memory Protection Keys - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html, 2021.

[15] Single Root IO Virtualization (SR-IOV) - Mellanox. https://docs.mellanox.com/pages/viewpage.action?pageId=47033949, 2021.

[16] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. RSS++: Load and State-Aware Receive Side Scaling. In *ACM CoNEXT*, 2019.

[17] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *ACM/IEEE ANCS*, 2015.

[18] Nicola Bonelli, Fabio Del Vigna, Alessandra Fais, Giuseppe Lettieri, and Gregorio Procissi. Programming socket-independent network functions with nethuns. *SIGCOMM Comput. Commun. Rev.*, 52(2):35–48, 2022.

[19] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *ACM SIGCOMM*, 2016.

[20] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. Pku pitfalls: Attacks on pku-based memory isolation systems. In *USENIX Security*, 2020.

[21] R. Cziva and D. P. Pezaros. Container Network Functions: Bringing NFV to the Network Edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.

[22] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP*, 2009.

[23] Mohamed Esam Elsaid, Hazem M Abbas, and Christoph Meinel. Virtual machines pre-copy live migration cost modeling and prediction: a survey. *Distributed and Parallel Databases*, pages 1–34, 2021.

[24] Mehmet Ersue. Etsi nfv management and orchestration - an overview. https://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf, 2013.

[25] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-Core 100-Gbps Networking. In *ACM ASPLOS*, 2021.

[26] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *Technical Report arXiv:1305.0209, 2013*, 2013.

[27] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *ACM SIGCOMM*, 2014.

[28] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet order matters! improving application performance by deliberately delaying packets. In *USENIX NSDI*, 2022.

[29] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and efficient memory protection keys. In *USENIX ATC*, 2022.

[30] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.

[31] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*, 2019.

[32] Hongyi Huang, Wenfei Wu, Yongchao He, Bangwen Deng, Ying Zhang, Yongqiang Xiong, Guo Chen, Yong Cui, and Peng Cheng. NFD: Using Behavior Models to Develop Cross-Platform Network Functions. In *IEEE INFOCOM*, 2021.

[33] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *USENIX NSDI*, 2014.

[34] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *USENIX NSDI*, 2017.

[35] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *USENIX NSDI*, 2018.

[36] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr, and Dejan Kostić. SNF: synthesizing high performance NFV service chains. *PeerJ Computer Science*, 2:e98, 2016.

[37] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[38] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *ACM EuroSys*, 2017.

[39] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *ACM EuroSys*, 2021.

[40] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *ACM SIGCOMM*, 2017.

[41] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming Network Stack for Middleboxes with Rubik. In *USENIX NSDI*, 2021.

[42] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K.K Ramakrishnan, and Timothy Wood. Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions. In *ACM SIGCOMM*, 2018.

[43] Guyue Liu, Hugo Sadok, Anne Kohlbrenner, Bryan Parno, Vyas Sekar, and Justine Sherry. Don't yank my chain: Auditable NF service chaining. In *USENIX NSDI*, 2021.

[44] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. In *ACM HPDC*, 2011.

[45] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *ACM CCS*, 2015.

[46] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-In-Time summoning of unikernels. In *USENIX NSDI*, 2015.

[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *ACM SOSP*, 2017.

[48] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM*, 2020.

[49] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *USENIX NSDI*, 2014.

[50] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. Domain Specific Run Time Optimization for Software Data Planes. In *ACM ASPLOS*, 2022.

[51] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *ACM SOSP*, 2015.

[52] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *USENIX OSDI*, 2016.

[53] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX ATC*, 2019.

[54] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding network functions in the cloud. In *USNIEX NSDI*, 2018.

[55] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *USENIX NSDI*, 2013.

[56] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-Grained isolation for scalable, dynamic, multi-tenant edge clouds. In *USENIX ATC*, 2020.

[57] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[58] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *ACM APNet*, 2019.

[59] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *ACM ASPLOS*, 2021.

[60] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *USENIX NSDI*, 2012.

[61] Junxian Shen, Heng Yu, Zhilong Zheng, Chen Sun, Mingwei Xu, and Jilong Wang. Serpens: A high-performance serverless platform for nfv. In *IEEE/ACM IWQoS*, 2020.

[62] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-recovery for middleboxes. In *ACM SIGCOMM*, 2015.

[63] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. Snf: Serverless network functions. In *ACM SoCC*, 2020.

[64] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling Network Function Parallelism in NFV. In *ACM SIGCOMM*, 2017.

[65] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*, 2018.

[66] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK). In *USENIX Security*, 2019.

[67] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *ACM SOSP*, 1993.

[68] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable nf virtualization platform. In *ACM SoCC*, 2022.

[69] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as Processes. In *ACM SoCC*, 2018.

[70] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *USENIX NSDI*, 2018.

[71] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[72] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani, and Aditya Akella. Liveness verification of stateful network functions. In *USENIX NSDI*, 2020.

[73] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *ACM SOSP*, 2019.

[74] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified nat. In *ACM SIGCOMM*, 2017.

[75] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *USENIX NSDI*, 2020.

[76] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *ACM HotMiddlebox*, 2016.

[77] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *ACM SOSR*, 2017.

[78] Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. A closer look at NFV execution models. In *ACM APNet*, 2019.

Appendix A Multi-Threaded LEMON Scheduling

For a single-threaded LEMON, the hypervisor would initialize it, and the working thread will continue executing it, as shown in Figure 4b. However, a LEMON could create its own threads, which, except for the packet processing threads, could include threads for event logging or user input communication. In the following, we present how LemonNFV cooperates with the multi-threaded LEMONs.

Taking Figure 8 as an example, a LemonNFV process has a hypervisor thread (T_h, red path), and an RTC working thread

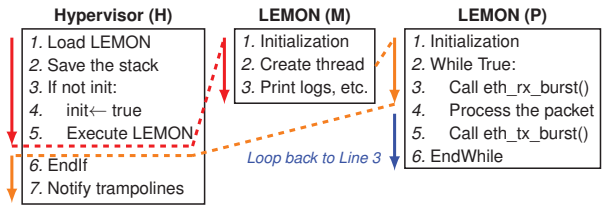


Figure 8: Cooperating with multi-threaded LEMONs. The red path is the (original) hypervisor thread, and the orange path is the newly created packet processing thread. After the LEMON is initialized, the red thread becomes its main thread, and the orange thread becomes the hypervisor thread.

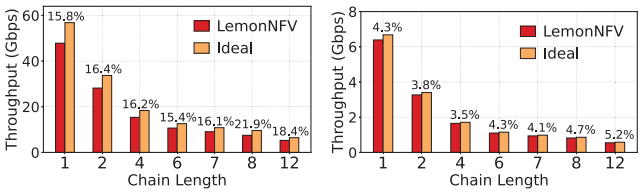
(T_r, blue path). Assume a new LEMON M1 is being loaded into the SFC, which creates a new thread for packet processing (T_p, orange path), and manages that thread in its main thread (T_m). The hypervisor would first execute the `main` function in T_h, that is, $T_m = T_h$ (Line 5 in H). After creating T_p in T_h (Line 2 in M), the execution in T_p would eventually call the virtualized I/O for the very first time, *i.e.*, the initialization part (Line 3 in P). Then the execution would be switched into the hypervisor stack (Line 6 in H). Note that since T_p is executing the hypervisor logic, this thread now actually plays the role of hypervisor thread. Next, T_p (the new hypervisor thread) would notify T_r that a LEMON is ready to be attached into the SFC (Line 7 in H), and T_r will switch the stack into the packet processing logics of this LEMON next time received the packets (Line 4 in P).

In sum, after loading a LEMON, T_h becomes the management thread of the new LEMON, T_p becomes the new hypervisor thread, and T_r is still the RTC working thread. Note that the LEMON can also use T_m as the packet processing thread, and create a new thread for management. In this case, T_m (T_h) will still be the hypervisor thread. The key here is that there is only one packet processing thread for each LEMON, so the virtualized I/O calls in that thread would eventually guide the hypervisor and trampolines to properly handle it.

Appendix B Dirty Pages Tracking in Inter-Process Migration

In the case of VM, the hypervisor can migrate the running VM to another host without halting it, namely live migration. The key is to capture the dirty pages by changing the accessing rights of the page table entries. For example, Xen implements a shadow page table to the original VM page table [44]. When the hypervisor decides to track memory modifications, the internal page table of the VM is transparently set to read only. That is, memory writes will not trigger a fault, but propagate to the shadow page table, recording the dirty pages during pre-copy stage in the hypervisor. vMotion in VMWare takes a similar approach [12].

Although each LEMON does not have a shadow or an isolated page table, its memory region has explicit boundary.



(a) Lightweight (b) Heavy

Figure 9: Performance comparison of differently-loaded NFs between their LemonNFV and vanilla DPDK version.

As a result, it is possible to adopt the above method by directly changing the accessing rights of the page table inside a LEMON. To be specific, when migrating a LEMON, LemonNFV first blocks the LEMON and changes the permission of the LEMON's memory to read only. Then, all the memory writes to the LEMON will trigger a `SIGSEGV` signal. A pre-registered signal handler would capture this signal, record the address to be written (which pollutes the page), and grant write permission to the corresponding page. Upon completion of the pre-copy stage (*i.e.*, phase -1 in §4.4), the LEMON is set to read only again, waiting for the next iteration.

Experiments on `libnids` [6], prads [13] and nDPI [9] with real traffic show that after a few iterations, the number of the modified pages eventually converge to 13, 7 and 4, respectively (4KB for each page), which are far lower than the stopping condition of iteration in Xen (<50 pages) and vMotion (<16MB) [23], meaning that the factual migration would only halt the LEMON for a neglected moment.

Appendix C Microbenchmark

In this section we present some additional microbenchmarks of LemonNFV.

As in §6.3, the overhead of LemonNFV is irrelevant to the NF logic, contrast to array bound checking. It's then worth discussing how this overhead would impact the performance, under various workload and SFC length.

We prepare a light NF (∼200 cycles per packet) and a heavy NF (∼2300 cycles per packet), packing them as LEMON as well as porting them to DPDK. Under LemonNFV, variable number of identical LEMONs are chained together, while the DPDK version repetitively invokes the packet processing function until chain length is met. Since the vanilla DPDK version does not introduce additional overhead, it is referred to as 'Ideal' later as a baseline.

Figure 9 presents the end-to-end throughput under variable chain length and NF workload. We have the following observations and analysis for the results. First, the performance gap between LemonNFV and Ideal is large when the NF is lightweight (17.1% on average across all chain length), but becomes much less significant when the NF is heavy (4.3% on average). This corresponds to our analysis in §6.3 that the overhead of LEMON switching is irrelevant from NF logic. Since real-world NFs are generally more complex and

time-consuming, LemonNFV would be more preferable than isolation methods based on array bound checking.

Second, given the fixed switching overhead, the performance gap should be stable when the chain length increases. This is also verified in Figure 9 when the chain length is under 7. However, the slowdown of LemonNFV grows when the chain length continues to increase. For example, the average slowdown of lightweight NF rises from 16.0% to 20.2% when the SFC is longer than 8. We believe that this is mainly due to cache contention of NFs co-locating on the same core [48,65]. Compared with virtualization based systems that often run NFs on separated cores, RTC scheduling will be more likely to drain cache and cause performance degradation [78]. We consider it as a feature of RTC and leave better profiling and optimizing cache performance as our future work.

Disaggregating Stateful Network Functions

Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai‡, Mario Baldi‡, Krishna Doddapaneni‡, Arun Selvarajan‡, Arunkumar Arumugam‡, Balakrishnan Raman‡, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, Srikanth Kandula
Microsoft and AMD Pensando‡

Abstract– For security, isolation, metering and other purposes, public clouds today implement complex network functions at every server. Today's implementations, in software or on FPGAs and ASICs that are attached to each host, are becoming increasingly complex, costly and bottlenecks to scalability. We present a different design that disaggregates network function processing off the host and into shared resource pools by making novel use of appliances which tightly integrate general-purpose ARM cores with high-speed stateful match processing ASICs. When work is skewed across VMs, such disaggregation can offer better reliability and performance over the state-of-art at a lower per-server cost. We describe our solutions to the consequent challenges and present results from a production deployment at a large public cloud.

1 Introduction

All major cloud providers implement stateful network functions at their servers. These network functions are essential for network virtualization (e.g., private address spaces [75, 88]), enhanced security (e.g., stateful firewalls [14, 15]), load balancing [56, 87], QoS [62, 68, 92] and cost metering [5, 9, 20].

The key challenges in implementing stateful network functions in a virtualized context are three-fold:

- First, the state that must be maintained and accessed at line-rate can be per flow (for stateful firewalls) or per endpoint (to virtualize IP addresses) and can exceed 100MB for many virtual machines. Programmable switches [24, 25] have small SRAMs and are hence appropriate only in niche cases such as to only support a small subset of all flows [83, 85] or in bare-metal settings where the cloud provider has no access to the servers [41]. The most widely-used NF implementations today combine software in host virtual switches [52, 57, 88] with FPGAs or smart NICs that are directly attached to the servers [6, 58].
- Next, attaching FPGAs and smart NICs to each server is wasteful because these cards must be provisioned to meet the peak anticipated usage at each server but the actual usage is far below the peak most of the time, and on most servers. Moreover, VMs that use networking features which are only supported by the latest FPGA or smartNIC cannot be deployed on older hardware; in turn, this can lead to a sizable waste of non-networking resources.
- Finally, the tail performance is limited today. For example, in the three largest public clouds, we will show that the number of new connections per second a VM can support is well below the NIC capacity and is likely bottlenecked on stateful NFs that are applied on each new connection. Customers who deploy middleboxes in VMs (such as the Palo Alto VM-series firewall [33]) to secure access to their other VMs are forced to deploy many more middleboxes to offset the performance limitations in the provider's NF processing [35].

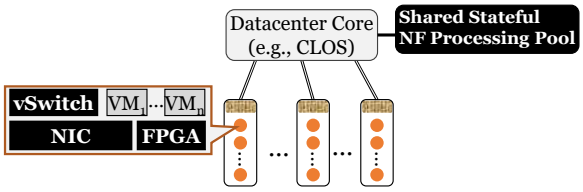


Figure 1: Today, stateful network functions are implemented on-host in virtual switches, NICs and FPGAs shown with a dark background. We propose to disaggregate stateful NFs, i.e., also process them in shared resource pools located elsewhere in the datacenter.

We propose to disaggregate the processing of stateful network functions into shared off-host resources pools as shown in Figure 1. Similar to how a customer can pick the CPU or memory for a VM, customers can also now pick a floating network interface (fNIC) which explicitly specifies NF requirements (e.g., # new flows per second, # concurrent flows) as well as the network capacity in Gbps. When deploying a VM, we allocate resources for the fNIC either on-host (that is, on the vswitch and the FPGA), or at an off-host shared resource pool or some combination at both locations.

We call out a few advantages from such a disaggregation

Stateful NF	State at each VM	Computation
Private address spaces [2, 10, 22]	A dictionary that maps customer's private addresses to the provider's physical addresses; one entry per remote endpoint that the VM speaks with.	Lookups, adds and deletes into the mapping dictionary
Stateful ACLs [32, 36]	Per ongoing flow that has passed the ACLs, a hashmap containing the flow's five tuple and the reverse five tuple	Lookups, adds and deletes into the per-flow hash table
Billing [5, 9, 20]	Total bytes, sliced by windows and per billable communicating entity such as a datacenter or a cloud service; also, bursts and peak rates	Multiple counters and sketches
Stateful NATs, load balancers	Per ongoing flow, the new *flow* to masquerade as.	Lookups, adds and deletes into the rewrite dictionary

Table 1: Some example stateful network functions that are implemented at cloud VMs and the associated state and computation. For more details, see [52, 57, 88].

of stateful NFs. First, we will show that for most of the time most of the VMs require much fewer NF processing capacity than the peak. We can thus reduce cost and power usage by equipping servers with less capable FPGAs or smart NICs and handling all of the spillover load at the shared resource pools. Next, we can deploy VMs which require novel NF processing on any server in the datacenter, including on older hardware, as opposed to restricting to just servers that have the latest FPGAs or smart NICs. As noted above, doing so reduces the fragmentation of non-networking resources. Third, we show how to increase the tail performance for VMs well beyond what is achievable from using the single FPGA or smartNIC that is attached to the host; for example, the number of new connections-per-second a VM can accept may only be limited by its NIC capacity. Doing so reduces cost and eases the deployment of middlebox VMs.

There has been much work on resource disaggregation. Disaggregating stateful NFs is similar in some ways to prior works that disaggregate resources such as memory, storage or GPU [45, 63, 74, 91] but there are a few key differences. One challenge is with regards to implementing a high-performance shared NF resource pool. The pool must simultaneously support large state and high-speed packet processing (e.g., 100s of GBs of states at multi Tbps packet processing rates). Doing so requires coherent access over a large memory at a high-speed. Programmable switches [24, 25] can process at multi Tbps but only have about 1GB of SRAM per switch. We have implemented an appliance which can be thought of as a bag-of-NICs wherein each NIC contains match-processing-unit ASICs that are programmable in P4 as well as a large coherently-accessible memory. Each appliance has 12 NIC cards, each card has a power draw of 75W, 16GBs usable for NF state and can process duplex packets at 100Gbps.

Another novel challenge from disaggregating stateful network functions is that fault tolerance shifts from a fate-sharing mode to a single point of failure. That is, when an FPGA or a smart NIC fails, only the VMs on the corresponding host fail but when an appliance (in the shared NF resource pool) fails the impact is felt by any VM whose fNIC happens to be allocated on that appliance. Naïvely replicating the state of network functions is hard because both primary-backup style replication [44, 51] and Paxos-like protocols [46, 48, 50, 81] queue requests while the state is being replicated. In the case of stateful NFs, requests can be any packet that changes state and so holding requests at speeds of hundreds of Gbps will require a very large packet buffer. We show how to replicate state *in-line* by ping-ponging packets between the replicas (pairs of programmable NICs) effectively buffering the state-changing requests on the network wires.

To the best of our knowledge, we are not aware of any prior work that disaggregates stateful NFs such as connection tracking firewalls or uses programmable bag-of-NICs appliances or supports in-line state replication. Some works offload specific stateful NFs to top-of-the-rack switches [41, 83, 85, 95] but do not support a rich class of NFs and the memory limit on Tofinos restricts them to only speedup a small subset of flows. Andromeda [52] deploys dedicated software middleboxes to process NFs but does not support stateful NFs citing concerns such as "state loss during upgrade or failure" and "transferring state when offloading". We discuss other related work in §7. To sum up, our key contributions are:

- We build a case to disaggregate stateful NFs by studying the functions and telemetry at a large cloud provider (§2).
- We present Sirius which disaggregates a rich class of stateful network functions onto pools of P4 programmable NIC cards. We show how to replicate state inline between pairs of nearby cards such that individual card failure does not adversely impact ongoing connections (§3.2). We discuss multiple disaggregation design-points including those that split or migrate the load of a VM across different NF processors (§3.3) and show a programmable NIC implementation that achieves better performance-over-cost than state-of-the-art (§4).
- We report results from a production deployment which, in part, show that when VMs offload onto Sirius, their stateful NF processing capacity improves by about 10×.

2 Background and Motivation

2.1 Stateful Network Functions

Table 1 lists some stateful network functions that are supported by public cloud providers. As the table notes, some NFs must maintain per-flow state whereas others keep state at coarser granularity. Counters and sketches are used to measure network usage for billing and diagnostics [73]. Customers

can also configure (add to) the stateful NFs on their VMs. As exemplars, we discuss two kinds of stateful NFs that are widely used in production but less well known academically.

First, virtual network peering [8, 12, 23] allows VMs in different virtual networks to communicate. Doing so requires all VMs in participating vnets to know how to map a virtual address belonging to any vnet into the corresponding physical address. This mapping must be kept up-to-date whenever VMs are deployed or migrated. Today's vnet peering implementations cache the map in a stateful layer at VMs [8, 12, 23].

Next, private links [7, 11, 29] let VMs communicate with PaaS services that have public IPs on a more direct path. For example, when a VM in AWS reads from an EBS volume which has a public IP, naïvely, such traffic must go via the cloud egress gateway similar to traffic to any public IP and then turn back towards the cloud store. Private links are more efficient and secure by having such traffic go directly to the cloud storage; a stateful layer at each VM encapsulates the outgoing traffic based on the VMs vnet id and the private IP address of the PaaS service and, in the reverse direction, a stateful layer at the PaaS service remembers which virtual and physical addresses a flow comes from when decapsulating (so-called *stateful decap*) and uses that state to encapsulate packets so that they go back to the appropriate VM.

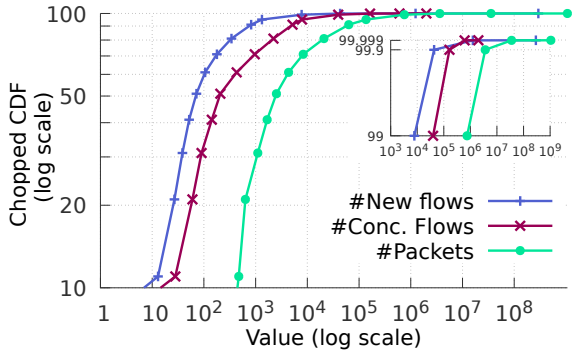
To sum, associated with each VM in the public cloud, providers implement numerous stateful network functions. Among the NFs considered in this paper, the connection-tracking firewalls, NATs and load balancers are the most intensive – they all require per-connection state. The total state to maintain per VM is often large since there can be hundreds of distinct *rules* to apply: one per private link, stateful ACL or vnet peer. NF actions on new connections are often implemented in software due to complexity and ease-of-programmability [52, 57, 88] whereas the per-packet actions are implemented in FPGAs or ASICs [6, 58].

2.2 NF workload at a public cloud

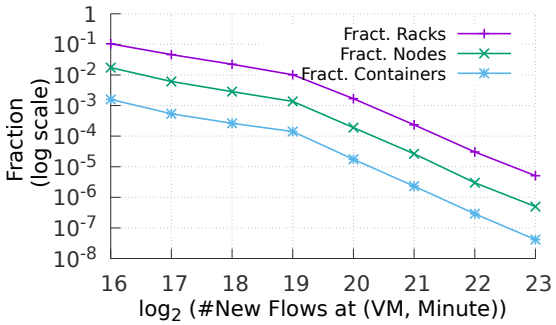
We characterize the usage of network functions at Azure. For each VM and each minute, we obtain the number of newly established flows, the number of active flows and the byte and packet counts. Our results here summarize metrics from a three month period. A typical minute has reports from $O(10^8)$ VMs and $O(10^7)$ nodes. Our key findings are as follows.

Skew in load for NFs: Figure 2a shows that the load for network functions is skewed; we measure load in terms of the number of newly arriving flows which must be verified to be policy compliant, the number of concurrently active flows for which state has to be maintained and the number of packets being exchanged. We see that the median load is multiple orders of magnitude smaller than the peak load. The inset zooms in on values further on the tail.

When the load is skewed, provisioning every host for the peak (e.g., by adding FPGAs or smart NICs), can be costly and most of the NF processing capability remains unused. We aim to provision hosts for the average load and handle the excess load using a disaggregated, logically shared, pool.



(a) Cumulative distribution function (CDF) of the NF Load at VMs; axes are in log scale and points on the CDF are unique (VM, minute) tuples.



(b) VMs that have high NF load are spread among many nodes and racks.

Figure 2: Characterizing the workload for stateful NFs at a large public cloud; data collected across over 10^8 VMs in a three month period.

Metric	Containers	Nodes	Racks
σ / μ	14.23	5.00	0.67
99th / μ	13.54	10.49	2.52

Table 2: Coefficient-of-variation (=stdev σ/ avg. μ) of the number of newly arriving flows per minute at each VM compared to the same metric when rolled up into the nodes or racks that contain the VMs.

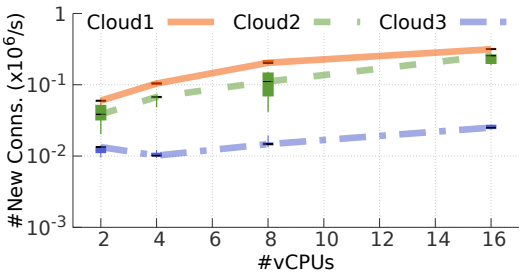
Containers with high NF load are spread throughout the network: Figure 2b zooms in on VMs and timewindows (minutes) which report high NF load; the x axes is a logarithmic bin, that is $x = 18$ denotes that the numbers of new flows in a (VM, minute) was in the range of $[2^{17.5}, 2^{18.5})$. The bottom-most line on the figure reports the fractions of distinct containers which exhibit high NF load. If the high-load containers were concentrated into a few nodes and racks, then the fraction of nodes and racks which show high load will be no larger than the fraction of containers. However, the figure shows that highly-loaded containers are spread across many more nodes and racks. The case for other NF load metrics is similar. About 10% of the racks have at least one container which reported over 50000 new connections in a minute.

Variation in NF load: At the granularity of individual VMs, we observe sizable temporal variations in NF load of up to

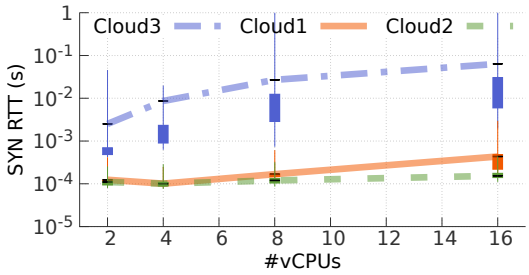
one order of magnitude larger than the median (see Figure 15). However, the variability appears uncorrelated spatially. That is, the sum of the load of all the VMs in a server or a rack has smaller coefficient-of-variation (see Table 2). Shared pools of NF processing capability thus can be provisioned with smaller peak to average ratios and will be more cost-efficient.

2.3 Characterizing NF Performance

To measure the state-of-art stateful NF performance of public clouds today, we deploy pairs of VMs of different sizes, add a stateful ACL to the *client* VM and initiate TCP connections in an open loop to the other *server VM*. We progressively increase the connection initiation rate and measure the maximum rate that was achieved and the connection establishment latency. Our tool is multi-threaded and asynchronous. We appropriately change various configuration variables and achieve better results than listed in public datasheets [13, 17–19, 34]. Figure 3a shows the maximum number of new connections per second (CPS) achieved by VMs of different sizes on the three largest public clouds today. All VMs are identically configured Ubuntu Linux instances. The variation is over experiment runs likely due to performance interference on the VMs or on the network path; we repeat each point at least ten times. The figure shows that increasing the VM size tends to increase the CPS perhaps because the per-VM networking limits improve [3, 21, 28]. However, the highest CPS across all public clouds and experimented VMs is 0.3M. Figure 3b shows the latency between sending a SYN and receiving a SYN-ACK. In the latency plot, we only use trials where most of the connections succeed to avoid latency cliffs. The figures show that processing the stateful NFs which are deployed in public clouds today represents a sizable bottleneck– there is a sizable latency when establishing new connections and VMs are limited in the number of new connections per second that they can sustain.



(a) Maximum Connections Per Second achieved.



(b) Latency between sending a SYN and getting a SYN-ACK.

Figure 3: Benchmarking connection establishment rate and latency at different VM sizes on three public clouds. The lines connect the average value, whiskers go from 1st to 99th and boxes go from 25th to 75th percentiles.

3 Disaggregating NF processing in Sirius

Sirius offers a new API to offload network function processing into pools of appliances that contain custom programmable cards. Each VM or container can specify a floating NIC (fNIC) with requirements on the following dimensions that relate to processing network functions:

- The number of new flows per second (CPS)
- The maximal number of concurrent flows
- Network capacity
- Feature, capability selection and ruleset size

Values for some of these dimensions are already in cloud provider and NVA vendor datasheets [3, 13, 17–19, 21, 28, 34]; Sirius also allows off-the-shelf fNIC sizes that users can pick from (e.g., a *small* fNIC) to help with configuration.

We extend our cloud VM allocator [65, 93] to provision fNICs using either resources on the smartNICs that are attached to servers or one or more Sirius appliances. Disaggregation lets VMs be placed on servers that may not locally satisfy fNIC requirements. Allocating fNICs to cards is an instance of the multi-dimensional bin packing problem [61, 86]. A better packing will map more fNICs onto fewer cards. We considered different heuristics and found that heuristic choice improves efficiency only when the fraction of large fNICs (whose resource needs are a substantial fraction of the card capacity) is high. In the rest of this section, we discuss the disaggregated datapath, *inline* state replication and methods to split or move an fNIC's load between multiple cards. Our design is modular and can work with different implementations of the shared processing pool; in §4, we discuss our P4 programmable cards which in our tests can serve over 3M new connections per second and 16M concurrent connections while processing a rich set of stateful NFs.

3.1 Connectivity and availability

The NF processing pool in Sirius is a collection of appliances. In our prototype, an appliance is a pair of 3U servers with six programmable cards each in PCIe slots. Each card has two 100Gbps QSFP+ connectors and 32GB DRAM.

Reliability, efficiency and flexibility were our key considerations when deciding how to connect Sirius appliances within a datacenter. Adding an appliance to each rack may lead to under-utilization (and fragmentation) since not every rack may have enough demand for NF processing. We also aim for

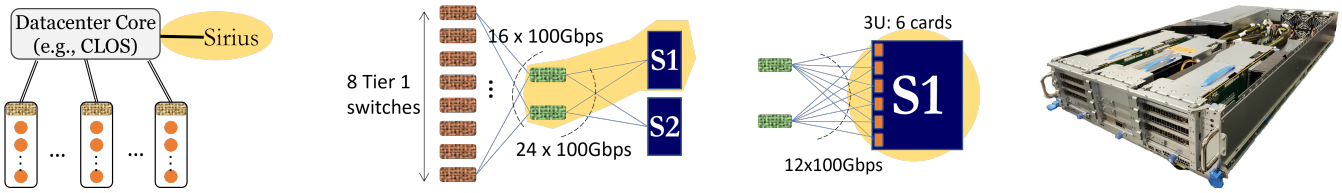


Figure 4: Connectivity diagram. Please read the figures left to right with each subsequent figure fleshing out the portion that is highlighted in the preceding figure. A Sirius appliance can be thought off as a pair of servers labeled S1 and S2 respectively that are connected as shown. Sirius assigns offloaded floating NICs of VMs to the programmable cards in the servers. We deploy more appliances to keep up with the total NF load.

a high level of reliability since NF processing is crucial for security and reachability in public clouds, e.g., for ACLs and private address spaces. Our goal is to ensure that the failure of any one server, card, link or switch must not degrade NF processing capability by a substantial amount. We also aim to independently scale out the NF processing capability as demand increases, e.g., by adding more appliances. Finally, we want VM placement to not be constrained by the availability or lack thereof of NF processing capability; that is, VMs located anywhere in a datacenter should be able to, when needed, use NF processing from the Sirius pool.

We choose to connect Sirius appliances as shown in Figure 4. Our minimum deployment unit is one appliance beneath two top-of-rack (ToR) equivalent switches that connect to the rest of the datacenter network similarly to other ToR switches. Such connectivity also ensures that any VM in any of the racks connected underneath the same CLOS will have equivalent access to Sirius's appliances thus realizing a large shared NF processing pool. In our experience with Sirius, the primary bottleneck is the NF processing capability and the state on the cards. That is, the number of new flows arriving per second which must be validated and the number of concurrent connections for whom state must be maintained (e.g., in a stateful load balancer). Our measurements show that the added latency incurred by traffic passing through an appliance is small relatively because traffic between randomly placed VMs in the public cloud almost always bounces off a switch in the CLOS tier; in particular, the increase is negligible for north-south traffic (which enters or leaves the datacenter). Finally, the connectivity diagram in Figure 4 preserves access to the NF processing capability under the following conditions.

1. At most one of the two green switches in front of a Sirius server fails.
2. At most one of the two links that connect a given card to the switches fails.
3. At most half of the links that connect the green switches to the red Tier-1 switches fail.
4. At most half of the red switches fail.

The last two conditions above ensure that other racks in the CLOS will have at least one valid path to the green switches. By preserving access to the bottleneck resource, NF processing remains unimpeded and Sirius will still be able to support high rates of new flows per second and concurrent flows.

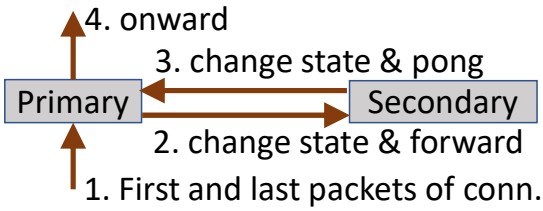


Figure 5: In-line replication of connection state in Sirius by ping-pong'ing packets that change state to both the primary and secondary cards.

3.2 In-line Connection State Replication

To avoid individual card failures from affecting ongoing connections, we duplicate connection state across two programmable cards. A key novel aspect here is that we do so without buffering packets. Due to the very high packet rates that these cards handle, holding packets in the primary card until state is established on the secondary card, as is done typically to replicate state [46, 48, 50, 81], will require very large buffers. We discuss a method that replicates state without any additional buffering by ping-pong'ing the packets of each connection that change state. As shown in Figure 5, for example, SYNs of a TCP connection which will establish state on the primary card are also forwarded to the secondary card. The secondary card also establishes state for this connection in its local memory and forwards the packet back to the primary. The primary card then transmits the packet to the destination in the usual way. Both cards independently delete the state of connections which remain idle beyond an age out threshold. Besides avoiding additional buffering, such *inline* state replication requires only a small code change to send and process ping-pong messages since the code to check rules and update state can be reused.

Each card pair (primary and secondary) exchanges heartbeats and fails over independently. That is, if the primary misses several heartbeats, the secondary card will receive all of the traffic on fNICs that were assigned to the pair. To achieve such failover, both cards announce BGP routes for the fNICs' virtual IPs; the primary card announces a shorter AS path than the secondary.[1] At a slower timescale, a different software controller provisions new replicas (e.g., pairs a newly promoted primary with a new secondary card) and schedules bulk state replication (which we describe below).

[1] We discuss corner cases in failover in §A.1.

The controller also reduces allocatable appliance capacity if necessary based on the number of cards that are operational. Two control loops at different timescales are commonly used to react to faults [67, 80]; to our knowledge, we are not aware of its use in replicating the state of network functions.

We observe that most of the connection state turns over quickly. For example, a usage stream that has 4M new connections per second and 32M concurrent connections has the average connection lasting 8s. Thus, waiting a bit will allow us to move much less state, which belongs to the long-lived connections, with the trade-off being a small increase in the period for which state is present on only one card.

The goal of our bulk synchronization is to replicate checkpointed state from one card to another quickly. There are multiple ways to implement checkpoints; we append an epoch value to each record in the state and atomically increment the value of the current epoch to take a checkpoint since then all records with a smaller epoch value will belong to the checkpoint.[2] To copy a checkpoint between paired cards, the ARM cores on the cards move state in batches over a reliable transport. We tradeoff the overhead of copying checkpoints with an increase in the period wherein only one copy of the state exists in the following additional ways: we prioritize moving the state of long-lived connections since other connections may close before needing to be moved and we pace the copy messages so that resource contention (on the memory bus and network) does not adversely affect normal activity.

To sum, replicating connection state between a pair of cards has the following costs and benefits. On the costs, storing each record at two cards halves the total available state that a Sirius applicance can maintain. The NF capacity, say in terms of connections per second that can be handled by an appliance, also halves for the same reason. The connection setup latency increases due to the ping-pong. Also, bulk synchronization, when triggered, uses memory and network bandwidth. On the benefits, the failure of a single card only impacts ongoing connections for the period before traffic failovers onto the secondary card. In-flight connections, that is, connections whose state is not yet present on both cards may only have to retransmit some SYNs (and FINs). To see why, observe that at any of the four steps for a new connection in Figure 5, the failure of either cards at best requires a retransmission.[3] Finally, planned card failures can be handled without any impact as so: (X1) promote the secondary and pick a third card to be the new secondary, (X2) take a checkpoint and (X3) initiate bulk synchronization. Upon completion of the bulk synchronization, the old primary card can be taken offline.[4]

[2] We use a small circular counter to track epoch values.

[3] We use a poison bit on the record written to the primary card which will be deleted only after the packet pongs back from the secondary to handle failures that may happen after step 2 in Figure 5.

[4] As a proof sketch, note that any new connection that reaches the new primary (old secondary) after X1 will reach the new secondary via the ping-pong method. Furthermore, all state at the time of the checkpoint, X2, will have been reliably copied to the new secondary.

3.3 Dividing NF load appropriately

So far, we have shown that the state for NFs can be maintained in a disaggregated resource pool with high availability. Here, we discuss different design points which divide the NF load between smart NICs that are directly attached to servers and one or more cards in the disaggregated Sirius pool.

3.3.1 Pin fNIC locally or to one card pair

Here, the load of each fNIC is assigned either to the on-server smart NIC or to a pair of cards as discussed in §3.2.

To realize pinning to a card pair, the outgoing packets of an fNIC are encapped in an NVGRE tunnel and sent to the chosen primary card in the Sirius pool which applies NFs on the packets and forwards them on to the destination. Traffic in the reverse direction takes an analogous path, first reaching the appliance/card which applies NFs and then forwarded to the VM if appropriate. We implement the encap and decap logic at the smart NICs on the servers.

3.3.2 Disaggregation Cost/ Benefit Analysis

The above design point already leads to substantial cost savings from disaggregation because one appliance can handle the NF load of over 24000 VMs on average. We compute this number as follows. In §6, we will show that each card used by Sirius can process over 16M new connections per second (CPS) with an extensive set of NFs. There are 12 cards per appliance. We assume that each VM has an average CPS load of $4K$ which is $400\times$ the current median load per Figure 2a and we halve the NF capacity to replicate state as discussed in §3.2. Hence, the cost for the additional switches, cables and the appliance in Figure 4 amortize well. Moreover, regarding peak load and temporal variations, note that these 24000 VMs may be distributed over hundreds of racks and, as we saw in Table 2, the total load over many rack has much lower variability. Thus, Sirius can meet SLOs with much smaller surplus capacity in its disaggregated pools.

3.3.3 Split the load of an fNIC across multiple cards

With the previous design point, the maximum size of an fNIC is limited by the capacity of one card in the Sirius pool. Moreover, as we will show, packing VMs into appliances is less efficient when the size of the balls (i.e., the fNIC size of a VM) becomes close to the size of the bins (i.e., NF capacity in one card). Sirius appliances can also be implemented using diverse hardware and different NFs may be better suited to different hardware. To this end, we aim to split the load of an fNIC across multiple cards or appliances. That is, different portions of the traffic entering or leaving one VM can receive their NF processing at different cards.

Consider splitting the load using a hash function–$\mathsf{hash}(\text{local IP}, \text{remote IP}) \bmod n$, in the encapper, to pick

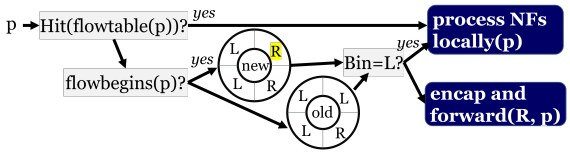


Figure 6: When spilling over NF load, as discussed in §3.3.4, we hash packets into bins and assign the bins to either local (L) or remote (R) NF processing. The figure shows two assignments new and old where the proportion of load that is processed locally is 50% and 75% respectively. When bins are re-assigned, the figure shows how we reduce the state that must be moved by using both bin assignments for a short duration.

from among *n* different cards. Such a *symmetric hash* ensures that the traffic of a flow in both directions will be processed at the same location which is required by some NFs (e.g., NATs [55]). While this requirement can be met in other ways, symmetric hashing requires no additional state at the encapper and decappers and we use hash functions that are easily implementable in NICs. Next, some NFs require groups of flows to be analyzed at one location. For example, a usage meter or a DDoS detector may want to count all bytes from a VM that leave the datacenter. Our experience is that most such NFs have mergeable actions [39], for example, to compute the total byte count, we can add up the partial sums from different processors. Many sketches (such as hyperloglog [54], count-min [49]) are mergeable with small reduction in accuracy [39, 40]. Finally, when an fNICs traffic is split across multiple NF processors, the ruleset corresponding to the fNIC must be installed at all of the corresponding processors; in practice, doing so adds overhead but is tenable because the total state at the NF processors is dominated by the per-flow state, counters and sketches rather than ruleset size; similarly processing the ruleset dominates the computation at the NF processor over the one-off installation of the ruleset.

3.3.4 Use Sirius as a load spillover

Thus far, all our load allocations have been static. That is, the whole or a portion of an fNICs traffic was allocated statically to the server's smart NIC or to a Sirius pool. An alternative is to move NF load that cannot be processed locally *dynamically* into a Sirius pool. For example, we may start processing all of the NF load locally on the server's smart NIC and when the total load nears smart NIC capacity, shed the excess load into the Sirius pool. Doing so will allow cloud providers to offer burstable SLOs on NF processing.[5] One Sirius appliance can scale to even more VMs compared to the pinned design-point above because only the excess the load of fNICs will be steered to the appliance.

Naïvely supporting such dynamism would require *moving* the state of NFs. For example, if a portion of the traffic that was to be processed on the smart NIC must now spill over into a Sirius pool then the corresponding state of all NFs must move. Intuitively, doing so is complex and our key contribution is to do so efficiently and correctly. First, our design aims to reduce the amount of state that must move to the extent possible. We hash packet headers, partition the resulting hash value into a fixed number of buckets (say 32), and assign different buckets to be processed for NFs at different locations. To move load, we change the bucket assignments; that is, to move 25% of the load from the smart NIC to a Sirius pool, we would reassign a quarter of the buckets that were being processed at the former location to the latter. Instead of moving all of the NF state that corresponds to a moving bucket we move *lazily* as shown in Figure 6. Effectively, newly created state (e.g., state for new connections) immediately reflects the current bucket assignment but for the previously established state, we delay movement by a short period (τ). Connections with duration below τ will not move and we observe that long-lived connections comprise a small fraction of all connections. The trade-off here is that we can rebalance load less frequently (once per τ). Our second idea is that many kinds of state can be re-created at the new NF processing location by just processing packets. For example, stateful ACLs insert the five tuples into a dictionary. The necessary information to create such state – the five tuple – is present in every packet of a flow and so, instead of moving state, we mark and steer packets to their new NF processing location. For state that cannot be recreated in this way, we craft new packets that include the packet header of the original flow and the state and transmit these packet to the new NF processing location. When the new location acknowledges creating the requisite state, the previous processing location deletes its state and load steering will exclusively use the new bucket assignment.

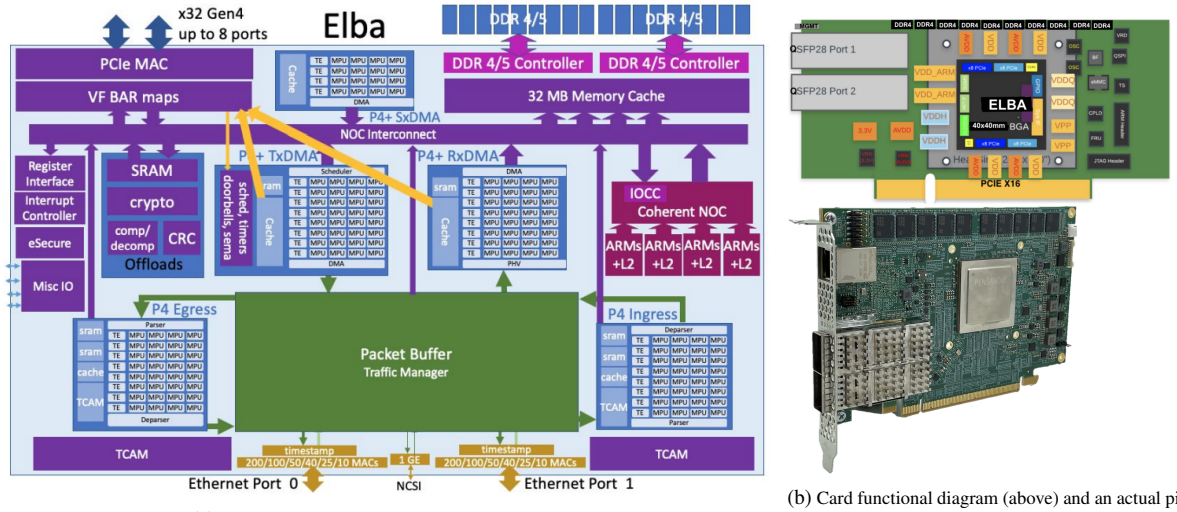
4 Efficient and high-rate NF processing

Thus far, we have discussed how to disaggregate the processing of stateful network functions in public clouds by using cards that (1) support inline replication of state (§3.2), (2) support various disaggregation design points including load splits and state movement (§3.3), and (3) implement a rich set of network functions (Table 1 and §2.1). Any implementation that satisfies these requirements can be used in this design including, for example, software-only or switch-only implementations. Here, we discuss our implementation which uses a specific kind of programmable NIC and compares favorably on functionality, performance and cost.

To process stateful network functions efficiently and at a high rate, we use the P4 programmable card shown in Figure 7 which has two 100 (or 200)GbE QSPF+ connectors, multiple pipelines that are programmable in P4, coherent shared memory, ARM cores for the more complex data plane processing and specialized logic for encryption and compression.

Relative to FPGA-based smart NICs [58], conjoining match-process-units (MPUs) that are programmable in P4

[5] Burstable allocations are already available for CPU and memory. They allow short-duration bursts or price differently the average and peak usage.



(a) Architecture diagram of our programmable ASIC.

(b) Card functional diagram (above) and an actual picture (below).

Figure 7: Hardware used to enable efficient and high-rate NF processing.

Metric	Sirius DSC-200	Other Packet Processors Tofino	Tofino2
Bandwidth (Tbps)	0.4	6.5	12.8
Memory	32GB	0.48GB	0.8GB
# Match Action Pipelines	5	4	4
# Stages/ pipeline	4	12	20
Packet Buffer	up to mem.	22MB	64MB
Integrated general-purpose cores	16	0	0

Table 3: Salient differences between packet processing hardware; Sirius uses the DSC-200 card [4] to support stateful network functions at a high scale and performance.

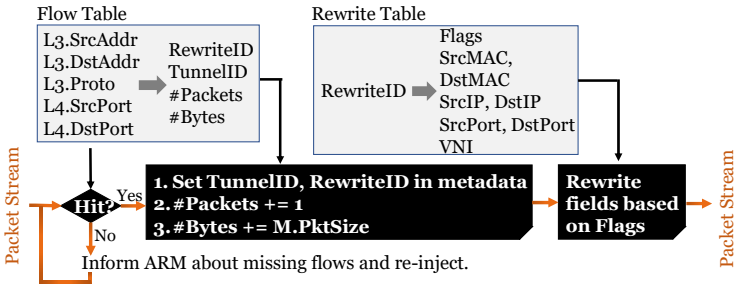


Figure 8: Stateful Load Balancer with NAT as implemented by Sirius.

with general purpose ARM cores gives us better programmability and performance at a lower power cost. Intuitively, power usage decreases because unlike FPGAs which expose general *gate-level* programmability, our card only exposes programmability in P4 that is needed to process protocols and stateful NFs efficiently. We use the ARM cores to handle programs that may be challenging to implement in P4 [72] such as reliably exchanging state migration messages (see §3.2).

Packets flow through one or more ingress and egress P4 pipelines and go through ARM cores only if needed. Each pipeline operates at 400Gbps (over 50M packets per second) thus ensuring line rate on both interfaces. We parse a packet once and populate a packet header vector (PHV) which is used by later stages. Each pipeline has local SRAM and TCAM to store high bandwidth tables and can also access the shared DRAM through a coherent shared memory which hides memory latency. A table engine at the beginning of each stage protects against stalls by processing multiple PHVs, issuing high latency reads in advance (e.g., to the DRAM), and moving to an MPU the next PHV for which all data is available. Each stage has multiple match-process-units (MPUs) which never stall and have dedicated write paths to the stage data buffer wherein writes are merged at a bit level to allow multiple MPUs to update different fields of a PHV. The MPUs implement a novel domain-specific instruction set architecture with an emphasis on bit field manipulations and fast header updates. We also use wide instructions (e.g., 64bit wide) which lets us use richer encoding and fewer instructions.

Coupling ARM cores and MPUs: Our card connects the P4 pipelines via a high speed network-on-chip (NOC) to a full system-on-chip (SOC) subsystem with multicore ARM A-72 CPUs. P4 programming determines which portions of packet data, headers, or metadata should be delivered to the DRAM and ARM on a per-application, per-packet basis. To support chained operations which may combine a P4 control operation with non-P4 operations, such as encryption or data integrity checksum verification, we attach a chaining buffer directly to the NOC to support high-bandwidth multi-hop chaining.

Illustrative Example: Using the case of a stateful load-balancer, we call out key aspects of how our hardware implementation improves upon the state of the art. Figure 8 shows a functional view of our stateful load balancer implementation. The relevant state (table shown with light background) is stored in DDR memory on the card. The logic boxes (shown in dark background) are implemented in the MPU pipelines and the exception path (for a new flow which does not have a

hit in the flow table) is handled by ARM cores. When load balancers are implemented without per-flow state (e.g., using a stateless hash function), any change to the pool of targets will disrupt ongoing flows; for example, a failure in one of the targets will cause the hash function to change from mod-n to mod-$n-1$ and all flows whose targets change will be disrupted [83]. Recognizing this issue, several large enterprises deploy stateful load balancers which remember per flow the target that the flow was assigned to [56, 83, 85, 87]. Prior work that proposes to accelerate stateful load balancers is limited by on-switch memory, for example, Sailfish [85] uses the Tofino chipset to support a few thousand stateful connections per switch, while the other flows are processed in software and receive no benefits. In our tests, one card can support over 16M concurrent connections and 3M new connections per second. We note a few aspects that help us achieve such performance:

- Although the flow table (in grey on the left in Figure 8) has one entry per ongoing connection, the rewrite table uses indirection and can be significantly smaller in size.
- Our table datastructures allow for more expressive rewrites including changes to the MAC addresses. Thus, we can use a single rewrite table for multiple NFs beyond load balancing (e.g., NVGRE encap [30]).
- We allow partitioning the MPU programs (shown in dark in Figure 8) among multiple pipelines so as to leverage data proximity.
- We divide the table ownership between ARM cores and MPUs to avoid coordinating multiple writers.
- When a new flow arrives for load balance, an ARM core installs entries in the flow and rewrite tables and reinjects the first packet of that flow into the MPU pipelines.

Comparing with recent works [42, 47, 79, 94], two of the P4 pipelines in the DSC (the Ingress and Egress pipelines at the bottom of Figure 7a) resemble reconfigurable match tables (RMT) [42] except that the DSC also has pipeline-local SRAM and not just stage-local SRAM. However, unlike RMT, all of the DSC pipelines can access shared DRAM through coherent caches. The DMA pipelines ({Tx-, Rx-, Sx-}DMA in Figure 7a) are novel and are triggered by timers and doorbells from a programmable scheduler. PANIC [79] addresses chaining offloads and is similar to the DSC which also uses specialized offloads (for crypto, compression and others, see Offloads in Figure 7a). However, while the DSC chains offloads, offloads are not central to the use of DSC in Sirius. FlexCore [94] discusses runtime re-programmability of switches; they add and remove P4 functions on an SN3000 [31] switch with minimal disruption to ongoing activity. We do not discuss re-programmability of the DSC cards in this paper. dRMT [47] pools all of the per-stage memory into shared memory that is accessible to any stage and uses a run-to-completion model wherein a packet is fully handled at one processor (and not in a sequence of match-action stages as in RMT). Our card offers larger shared DRAM instead. While it is unclear how dRMT's scheduler, which calculates a static schedule at compile time to guarantee deterministic throughput and latency, generalizes to the case of stateful NFs, the DSC supports stateful NFs more simply by dividing the work between P4 pipelines and ARM cores.

5 Implementation

We have implemented several stateful network functions (including those in Table 1) on the programmable NIC shown in Figure 7 from AMD Pensando. We have also added new code to the smartNICs attached to the hosts in Azure to steer traffic to and from the disaggregated Sirius pool. The resulting system, alongside software controllers to provision and monitor the fNICs, is in public preview at Azure [1] .

6 Evaluation

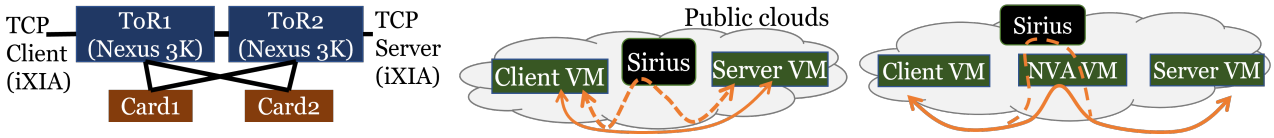
First, in a lab setting using full line-rate traffic generators, we show results for how the programmable NICs used in Sirius handle stateful network functions. We also evaluate key failure scenarios. Next, we report results from Azure wherein fNICs of virtual machines and network virtual appliances are offloaded to Sirius.

6.1 Methodology

Figure 9 shows our three experimental setups. On the left, in a lab, we use a traffic generator that sends and receives packets at hundreds of Gbps. We also mimic failures of the ToR switches, links, and cards to evaluate our state replication.

The other two setups use Sirius's production deployment in Azure. Figure 9b measures the performance between virtual machines (VMs) and Figure 9c measures the performance of network virtual appliances (NVAs) [13, 18, 33] when deployed on VMs. Here, we compare the default method that the public cloud uses to process stateful NFs versus offloading those NFs onto Sirius. In Figure 9b, we offload the floating NICs of both the VMs onto Sirius. In Figure 9c, we offload the floating NICs of the middlebox VM onto Sirius.

Stateful NFs: For the setup in Figure 9a, each card enforces a large prioritized set of stateful ACLs. As shown in Table 4, each ACL rule is a conjunction of predicates on sets or ranges of source and destination addresses, ports and protocol. Rules apply in priority order and may either accept or deny a connection. Some rules are specific to individual VMs whereas others apply to all VMs in a vnet or subscription. Recall from §2.1 that stateful firewalls maintain per-flow state of all ongoing connections so as to admit traffic in the reverse direction. As discussed in §3.3, the cards also encap and decap the packets to intercede on traffic transparently. For the VM-to-VM setup in Figure 9b, VMs in the public cloud already run many stateful NFs by default, for example, to virtualize



(a) Testbed (b) VM to VM: Default vs. offloading to Sirius (c) NVAs with fNICs offloaded to Sirius

Figure 9: Evaluation setups. On the left is a testbed where we use iXIA breakingpoint [27] to generate traffic of up to 100Gbps. The other two figures depict our experiment setups in Azure. We deploy VMs of different sizes and compare the performance when the Azure-specific stateful NFs are offloaded onto Sirius. On the right we measure the performance of network virtual appliances (such as the Palo Alto VM-series firewall) when using floating NICs that offload onto Sirius.

Granularity of ACL set	#Rules	Total #Conjuncts (prefixes or ranges)			
		Src IP	Dest. IP	Src port	Dest port
fNIC level	202	5102	5102	1021	1021
Subnet level	26	1168	1168	141	141
Subscription	8	394	394	57	57

Table 4: The ACLs for a stateful firewall deployed on the cards in Figure 9a; note: some ACLs are unique per floating NIC whereas others are common across all fNICs in a subnet or an entire subscription.

Resource type	Resource Capacity					
#Cores	2	4	8	16	32	64
Mem. (GB)	8	16	32	64	128	256
NIC Capacity (Gbps)	1	2	4	8	16	30

Table 5: The capacity of various resources for the SKUs used in Figure 9.

their network [2, 10, 22] or to estimate traffic bills [5, 9, 20]. In addition, we insert 1000 prioritized stateful ACLs on the client VM; these ACLs are similar to those in the testbed experiment. For the middlebox experiment in Figure 9c, we configure each middlebox with the reference load specified by the middlebox vendor.

VMs: We evaluate popularly-used Linux Ubuntu SKUs at various public clouds as shown in Table 5. We choose VMs with varying numbers of cores, from 2 to 64 vcpus; the other resources vary roughly proportionally as shown.

Traffic: In Figure 9a, we generate UDP and TCP flows of different sizes at different rates in an open loop using a synthetic traffic generator [27]. This appliance must be physically connected to switches and so, in the public cloud experiments, we use VM-based traffic generators. The Linux network stack cannot generate small TCP connections at high rates, e.g., fewer than 50K zero-byte flows per core [82]. We use the TREX tool instead which, using DPDK, can generate TCP-like connections at much higher rates [37].

6.2 Processing Stateful NFs in Sirius

To sum, the experiment here will show that when supporting a rich set of stateful ACLs (Table 4) the programmable NIC used by Sirius can support up to 3M new TCP connections per second (Figure 10b) and over 50M UDP packets-per-second (Figure 10a). The latency to ping-pong state messages between a card pair is less than 40μs (Figure 10c).

In more detail, Figure 10 shows the thruput and latency when the two cards in Figure 9a are set up as a state replicating pair; that is, all state changes for SYNs and FINs are ping-ponged between the cards as discussed in §3.2. Each datapoint is a several minute experiment and the errorbars show the range of measured values.

Since some stateful NFs are evaluated on every packet, we first measure the maximum number of packets per second (PPS) that our card can support by having the traffic generator send the smallest possible UDP packets at the highest possible rate.[6] Figure 10a shows that our card supports over 50M packets per second.[7] Typical packets are larger, e.g., many are MTU-sized, and so our card can process complex stateful NFs at line-rate with a fair amount of headroom.

The end-to-end latency through the card for 64B and 1500B packets is 2.36μs and 3.14μs respectively.

We also vary the number of concurrent flows which increases the state on the card and can make NF processing more challenging. Figure 10a shows only a modest decrease in PPS up to 64M concurrent flows; state for these many flows uses up most of the 32GB of DRAM on each card.

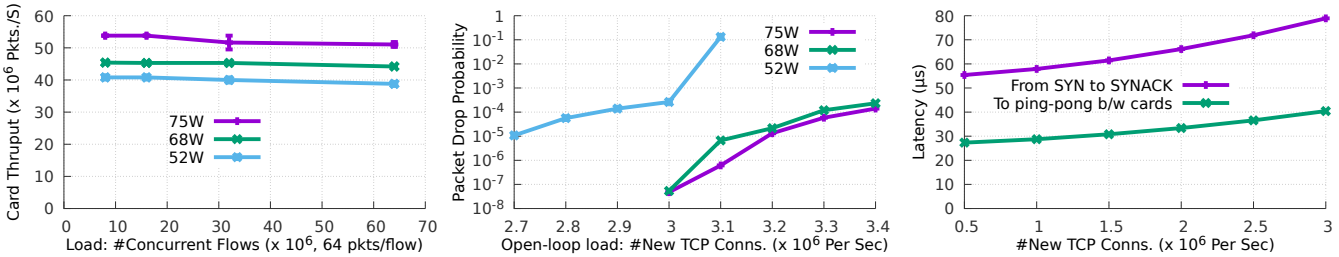
Finally, Figure 10a shows results for low power states of our card wherein we decrease the frequency of the MPU pipelines from their baseline value of 1.5GHz. Observe that we achieve 33% lower power draw with only a 25% drop in PPS. Thus, dynamic power cycling appears viable.

Next, when new connections arrive (or old connections finish) the state maintained in a stateful NF processor must change. To measure the maximum number of new connections per second (CPS) that one card pair can support, we have the traffic generator issue TCP connections in an open-loop with no payload.[8] Figure 10b shows the packet drop probability (y axes is in log scale) near our desired operating point of 3M CPS. Lower values to the right are better. Since SYN and FIN packets ping-pong between the cards, each card effectively processes twice as many state changes. The remaining packets of the connection, however, only go through the primary card. Also, recall from §4 that only the ARM cores change state and SYNs are reinjected into the MPUs after the ARM cores apply the ruleset. Figure 10b shows that while the 68W power state has very little effect on CPS, the lowest power state (52W) reduces the CPS to about 2.5M. We are not yet sure why and

[6] Each packet is 118B due to VxLAN tunneling with an interframe gap of 12B and the ethernet preamble of 8B [38]. Thus, on a 100Gbps link, the generator issues roughly 90M packets/s.

[7] Per previous calculation, this amounts to 60Gbps.

[8] 6 packets per connection: SYN, SYN-ACK, ACK, FIN, FIN-ACK, ACK



(a) The packets-per-second through our card when maintaining different numbers of concurrent flows.

(b) The fraction of packets dropped when our card is subject to open loop load, a line for each power state.

(c) Latency from sending a SYN to receiving a SYNACK and to ping-pong the SYN between cards.

Figure 10: Card measurements in the lab setup shown in Figure 9a: when applying per connection the thousands of stateful ACLs shown in Table 4, the figures show the thruput in PPS, packet drop probability and latency at different card power levels.

are looking into this issue.

For the CPS test above, Figure 10c measures the latency between sending a TCP SYN and receiving a SYN-ACK and the latency portion that is attributed to ping-pong between cards. The latency is flat at lower CPS load but grows super-linearly at higher demands likely due to queuing at the ARM cores or at reinjection. We note that RPCs can achieve a smaller latency [70, 84] by reusing connections and the latency here is better than that measured at the three clouds in Figure 3b.

6.3 Stateful NFs under faults

For the setup in Figure 9a, we have the traffic generator issue small TCP flows open-loop at the rate of 3M per second. The flows are spread over 16 floating NICs evenly allocated to the two Sirius cards. We examine the impact of three changes:

(a) planned switchover from Card1 to Card2,
(b) links between ToR1 and both cards go down and
(c) Card1 goes down.

For each scenario, we conduct three different experiments each lasting 60s and report average values. Each experiment comprises roughly 180M TCP connections and 1.08B packets. Table 6 shows that none of the flows *broke* as in there were no RSTs or connection time-outs in all three scenarios.

During planned switchover of load, as discussed in §3.2, Card2 advertises itself as the new destination for all of the floating NICs that were mapped to Card1. During the ensuing route reconvergence, the ToR switches drop 0.00316% of the packets and there are no drops at either of the cards. A naïve switchover would cause RSTs on half of the ongoing connections (all conns with state on Card1).

In scenario (b), where ToR1's links to both cards are down, the net available network capacity in/out of the cards halves but the CPS remains unaffected because, as noted in §3.1, Sirius retains large network capacity to the cards even when half of the connecting links fail. Table 6 shows that recovery here is slower and there are more drops because more routes must reconverge. The cards also see transient drops while their paths move over to ToR2.

Change	#Flow breaks	% of pkts dropped All	% of pkts dropped At Cards	Recovery Latency
(a) Planned switchover	0	0.00316%	0	1.89ms
(b) ToR1's links to both cards are down	0	0.00929%	0.0000227%	5.75ms
(c) Card1's links to both ToRs are down	0	0.00835%	0.0000201%	5.01ms

Table 6: Testing state replication under different fault scenarios in Figure 9a with 3M new TCP flows/s. Note, recovery in milliseconds and the extremely small fraction of packets that were dropped most of which are due to route reconvergence at the ToR switches. The drops at the Sirius cards are all packets that cannot be transmitted because the link to the next hop is down.

Scenario (c) mimics the failure of Card1. Here, the ToRs detect Card1 as being down and route all fNIC traffic on the backup BGP route to Card2. Contemporaneously, Card2 recognizes the failing peer, promotes itself to be the primary, and notifies the Sirius controller asking for a new secondary card. If card state was not replicated, half of all ongoing connections will receive RSTs in this scenario. Table 6 shows that no connections break. Instead, only a few packets are dropped most of which are at the ToRs. A few flows retransmit SYNs and FINs[9] which may have been lost without completing the pingpong in Figure 5 but none of the connections timeout.

6.4 VM-to-VM: Offloading fNICs to Sirius

Figure 11 shows the maximum connections per second achieved between pairs of VMs for the scenario in Figure 9b. Recall from §6.1 that we use TREX, a DPDK based generator on the VMs to create small TCP connections as many and as quickly as possible. These VMs have the default stateful NFs from public cloud and we add roughly 1000 randomly generated stateful ACLs (see §6.1). The figure shows that most of the VM SKUs, when onboarded on to Sirius, are only limited by the NIC capacity. That is, our tool makes as many connections as possible given the capacity limit of the NIC.[10] The figure shows that with Sirius, one VM pair can achieve

[9] 0.0062% and 0.0126% of the TCP flows respectively

[10] 6 packets per TCP connection each of which is 118 bytes after VxLan encapsulation which translates to 176.5K CPS per Gbps of NIC capacity. NIC capacities of the VMs are as shown in Table 5.

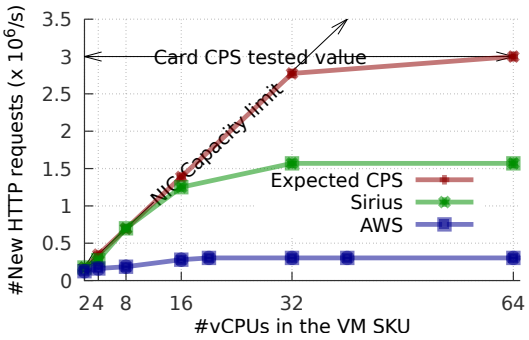


Figure 11: When the floating NIC of a VM is mapped onto the Sirius pool, showing the maximum CPS achieved between pairs of VMs.

over 1.5M connections per second. This value is below the maximum value per card – 3M CPS from Figure 10b– due to inefficiencies we believe in the code that steers fNIC traffic in the Azure smartNIC [58]. The figure also shows the CPS achieved using c5n series instances at EC2 using the same tool, the same configuration and the same guest OS. The lower CPS could be because EC2 employs different stateful NFs at each VM, uses a different NF processing system [6], applies explicit rate limits or some combination of all of the above. Comparing also with Figure 3a, we show that using Sirius a VM can achieve roughly 5× to 10× higher CPS. Further, recall from §3.3 that Sirius can split the load of a VM between multiple cards and so even higher CPS may be achievable.

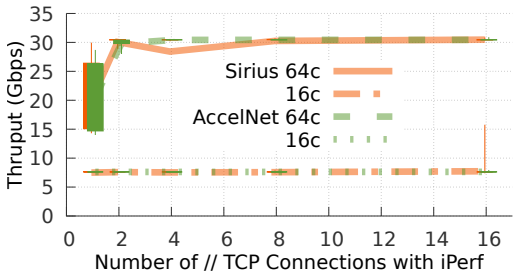
6.5 Measuring the Sirius datapath in Azure

To compare the datapath offered by the Sirius fNICs with the default datapath in Azure, we randomly and repeatedly deploy VMs and measure the latency and thruput on the two datapaths. Each VM in this experiment is equipped with three virtual NICs, one of which is used as the management interface and the other two are configured to use Sirius or AccelNet (the default in Azure) [58] respectively. The results shown are over millions of packets and tens of unique VM pairs.

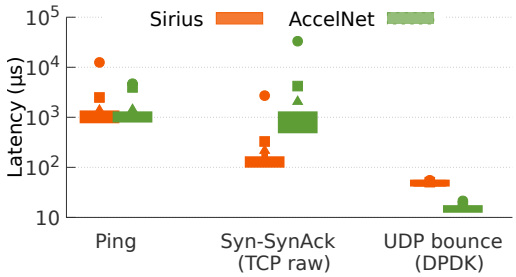
Figure 12a shows that the thruput achieved is nearly identical; with a small number of TCP flows, iPerf [26] can reach the NIC capacity on both of the datapaths.[11]

Figure 12b shows the latency for three kinds of applications. On the left are applications such as ping, tcping and hping3 which use the traditional in-kernel network stack. Such apps do not see any change in their RTT when using Sirius. Notice that with Sirius the datapath between a VM pair traverses up to two programmable NICs corresponding to the VMs' floating NICs. However, any increase in the physical length of the network path appears to be masked by the latency added by the guest kernel network stacks. In the middle of Figure 12b is the latency for the custom tool that we used in §2.3 which

[11] As noted in Table 5, the NIC capacity limits for the 16 and 64 core VMs that we used here are 8Gbps and 30Gbps respectively.



(a) Thruput achieved by iPerf with different numbers of TCP flows. The candlesticks show quartiles (bars) and the min and max values (whiskers) over many pairs of VMs and minutes.



(b) RTT using ping, TCP connections on raw sockets and a DPDK app that bounces UDP packets. In the candlesticks, the bars correspond to quartiles and the whiskers are the 10th and 90th percentiles. The triangle, square and circle above each bar show the 99th, 99.9th, and 99.99th percentile values respectively.

Figure 12: Comparing the datapath of AccelNet [58] with the disaggregated path through Sirius in Azure.

establishes TCP connections on raw sockets. As the figure shows, for such apps Sirius offers a better RTT than AccelNet because although Sirius may have a longer physical path, AccelNet takes much longer to process the stateful NFs for each new TCP connection. A third set of applications, on the right in Figure 12b, achieve very small latency by bypassing both the kernel network stacks (using an optimized DPDK app that we built) as well as the cloud's stateful NFs (by using UDP packets). As the figure shows, the typical latency for such apps is 15μs and 50μs respectively on the AccelNet [58] and Sirius datapaths. Note also the values on the tail. We conclude that any additional latency due to Sirius will only be visible to a small subset of applications and that for the vast majority of TCP-like traffic Sirius represents a clear improvement.

6.6 Offloading fNICs of middlebox NVAs

For the experiment setup shown in Figure 9c, Figure 13 shows the CPS achieved by traffic through different middlebox VMs. We generate results for Sirius using 32 core VMs as clients and servers of the traffic and offload the floating NIC of the middlebox VM onto Sirius. For all of the public clouds, we pick the best possible CPS numbers from datasheets released by the middlebox vendors [13, 17–19, 34]. The figure shows that using Sirius substantially improves the achievable throughput because the stateful network functions that cloud providers apply by default on the middlebox VM are often

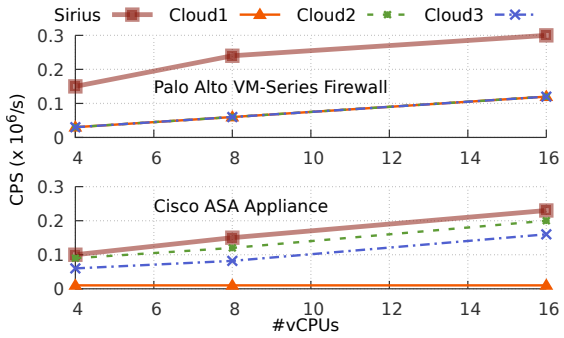


Figure 13: The CPS (# new connections per second) at which traffic can be sent through three different middlebox network virtual appliances on different public clouds and when onboarded onto Sirius.

the limiting factor to middlebox performance.

7 Related Work

The key focus of Sirius is to disaggregate stateful network functions onto pools of programmable NICs which tightly integrate P4-programmable MPUs and general purpose ARM cores with a large coherent memory system. With Sirius, we show how to replicate connection state inline so that individual card failure does not adversely impact ongoing connections (§3.2), discuss multiple design-points which split or migrate the load of a VM across different NF processors (§3.3) and offer an implementation that achieves better performance-over-cost than state-of-the-art (§4).

We are unaware of any prior characterization of the NF load at a large public cloud (§2.2). However, the case for disaggregation based on NF load being skewed across VMs and decorrelated (that is, having smaller variance when considered in aggregates such as at rack-level) is similar to the cases made to disaggregate other resources [59,66,74,78,89,91,96].

The state-of-the-art in processing stateful network functions is either in vswitch software or on programmable FPGAs that are directly connected to the host [6, 57, 58]. Andromeda [52] processes NFs at dedicated software middleboxes but explicitly states that they do not support stateful functions listing concerns such as 'state loss during upgrade or failure', 'transferring state when offloading' and 'ensuring that flows are 'sticky' to the hoverboard that has the correct state' [52]. We address some of these challenges in §3 and to the best of our knowledge are the first to disaggregate the rich class of stateful NFs listed in Table 1 and §2.1.

Offloading stateful network functions is non-trivial since a large amount of memory to maintain state must be accessible at high speeds. SRAMs support switch linerates but are expensive and so we use a bag of NICs architecture with memory coherence. Some prior works offload specific stateful NFs into programmable hardware [41, 83, 85]; however, they use switches and can only offload only a small subset of all flows, e.g., top-k by rate [41, 83, 85]. To compensate, TEA [73] pairs Tofinos with memory on remote servers and uses RPCs to access the remote state. When state is remote, it is challenging to achieve high performance and reliability. Also, Tofinos lack integrated general-purpose cores which forces TEA to build, in P4, a new RPC and a new reliable transport. With Sirius, each card has much larger memory. We replicate state between NICs on nearby servers in one pool and our general-purpose ARM cores simplify the logic. We believe that pairing cards which have tightly-integrated MPUs and ARM cores facilitates richer forms of disaggregation.

Another alternative is to use custom FPGAs with large memory (e.g., Xilinx and Altera). We are unaware of any works that match the performance and power draw of our cards using FPGAs. We believe that (1) P4 programmable MPUs are fundamentally more efficient than FPGAs [43, 76, 77], and (2) carefully dividing work between MPUs and ARM cores is key for high performance.

We discuss other related work in §C.

8 Conclusion

Stateful network functions are a key cog in today's public cloud architectures. We disaggregate their processing into a shared pool. Doing so avoids paying for smart-NICs at each server that are provisioned to support peak load, reduces constraints in VM placement and increases performance on the tail. Moreover, we attach this shared pool to the datacenter network at the layer off which most packets bounce off in the CLOS and so the latency and bandwidth overhead from packets taking a detour to the shared pool is small. We show a novel and simple solution that replicates connection state between pairs of cards without buffering packets while replicating state. We use NICs that have large memory, P4-programmable match-action pipelines and integrated general-purpose ARM cores. Our results from deployment at Azure show that network usage at VMs can reach NIC capacity even when complex stateful NFs are executed on each new connection and every packet.

Acknowledgements: We heartily appreciate the efforts of several team members whose work was crucial for the Sirius project including Aditya Baskar, Vivek Bhanu, Prachi Pravin Bhavsar, Weixi Chen, Nikita Dabir, Manasi Deval, Sumit Dhoble, Steve Espinosa, Osman Ertugay, Daniel Firestone, Shashank Gupta, Arun Jeedigunta, Sarat Kamisetti, Sam Kim, Guohan Lu, Ilias Marinos, Omar Mbarki, Kaixiang Miao, Kevin Pacella, Tommaso Pimpo, Vikas Prabhakar, Pirabhu Raman, Rohit Kumar Sharma, Yusef Skinner, Gabriel Silva, Prince Sunny, Hayden Udelson, Lihua Yuan, Zhenhua Yao, Xinyan Zan, Yuanyuan Zhou and Qi Zhang. We also thank the anonymous reviewers and our shepherd Aurojit Panda for feedback on the paper.

References

[1] Accelerated Connections and NVAs (Preview). https://bit.ly/424BkuF.

[2] Amazon EC2 instance IP addressing. https://go.aws/3qRcooQ.

[3] Amazon EC2 instance Network Bandwidth. https://go.aws/3mKS1ef.

[4] AMD DSC-200 Datasheet. https://bit.ly/3BrC0in.

[5] AWS: Data Transfer Costs for Common Architectures. https://go.aws/3cg5J3O.

[6] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[7] AWS PrivateLink. https://go.aws/3KG9xIs.

[8] AWS: VPC Peering. https://go.aws/3QcZxHI.

[9] Azure: Bandwidth Pricing. https://bit.ly/3Cou81Z.

[10] Azure: Private IP Addresses. https://bit.ly/3BqoSJ7.

[11] Azure PrivateLink. https://bit.ly/3CRXjKV.

[12] Azure: Virtual Network Peering. https://bit.ly/2AT5czqG.

[13] Cisco Adaptive Security Virtual Appliance (ASAv) Data Sheet. https://bit.ly/3UldTJq.

[14] Connection tracking: State and examples. https://bit.ly/3wrNdfP.

[15] Conntrack Tales – One thousand and one flows. https://bit.ly/3dL3eHf.

[16] Floodlight Controller. http://goo.gl/kzmC7.

[17] FortiGate-VM on Amazon Web Services. https://bit.ly/3Bp5qwb.

[18] FortiGate-VM on Google Cloud. https://bit.ly/3Sfz6CD.

[19] FortiGate-VM on Microsoft Azure. https://bit.ly/3BoXN93.

[20] Google Cloud: Bandwidth Pricing. https://bit.ly/3Cw83i9.

[21] Google cloud engine: Network bandwidth. https://bit.ly/3ywTVld.

[22] Google Cloud Platform: VPC Network Overview. https://bit.ly/3LpH4XP.

[23] Google Cloud: VPC Network Peering. https://bit.ly/3RsxiWG.

[24] Intel Tofino. https://intel.ly/3wxWT8w.

[25] Intel Tofino 2. https://intel.ly/3QTeD6F.

[26] iPerf. http://dast.nlanr.net/Projects/Iperf.

[27] IXIA BreakingPoint. https://bit.ly/3DqfOqd.

[28] Linux and Windows networking performance enhancements | Accelerated Networking. https://bit.ly/3kh7x05.

[29] New Private Service Connect simplifies secure access to services. https://bit.ly/3RyLMnN.

[30] NVGRE: Network Virtualization Using Generic Routing Encapsulation. https://www.rfc-editor.org/rfc/rfc7637.

[31] NVIDIA Spectrum SN3000 Open Ethernet Switches. https://bit.ly/3JLG9C1.

[32] OpenVSwitch: Conntrack Tutorial. https://bit.ly/3LsYtPd.

[33] Palo Alto Networks VM-Series Firewall. https://docs.paloaltonetworks.com/vm-series.

[34] Palo Alto Networks VM-Series Firewall Performance Datasheet. https://bit.ly/3f7vrJa.

[35] Palo alto networks vm-series performance & capacity. https://bit.ly/3BE8W7t.

[36] RedHat: IPTables and Connection Tracking. https://red.ht/3DAgucH.

[37] TREX: Realistic Traffic Generator. https://trex-tgn.cisco.com/.

[38] IEEE Standard for Ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, 2018.

[39] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable Summaries. *TODS*, 2013.

[40] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *NSDI*, 2012.

[41] Manikandan Arumugam et al. Bluebird: High-performance SDN for Bare-metal Cloud Services. In *NSDI*, 2022.

[42] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[43] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference. *ACM Trans. Reconfigurable Technol. Syst.*, 2018.

[44] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *Dependable Computing for Critical Applications 3*. Springer, 1993.

[45] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[46] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 2002.

[47] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslass, Ariel Orda, and Tom Edsall. dRMT: Disaggregated Programmable Switching. In *SIGCOMM*, 2017.

[48] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *SOSP*, 2009.

[49] G Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[50] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *SOSP*, 2015.

[51] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[52] Michael Dalton et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*, 2018.

[53] Tobias Distler. Byzantine fault-tolerant state-machine replication from a systems perspective. *ACM Comput. Surv.*, 2021.

[54] Marianne Durand and Philippe Flajolet. Loglog Counting of Large Cardinalities. In *ESA*, 2003.

[55] K. Egevang and P. Francis. The IP Network Address Translator (NAT). In *RFC 1631*, 1994.

[56] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI*, 2016.

[57] Daniel Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI*, 2017.

[58] Daniel Firestone et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[59] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[60] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[61] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[62] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *SIGCOMM*, 2020.

[63] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, 2017.

[64] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *International conference on reliable software technologies*. Springer, 1996.

[65] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *OSDI*, 2020.

[66] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, 2013.

[67] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[68] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *NSDI*, 2013.

[69] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, 2017.

[70] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *NSDI*, 2019.

[71] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *NSDI*, 2019.

[72] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling Fault-Tolerant Stateful In-Switch Applications. In *SIGCOMM*, 2021.

[73] Daehyeok Kim, Yibo Zhu, Zaoxing Liu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *SIGCOMM*, 2020.

[74] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *EuroSys*, 2016.

[75] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[76] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, 2006.

[77] Ian Kuon and Jonathan Rose. *Quantifying and exploring the gap between FPGAs and ASICs*. Springer Science & Business Media, 2010.

[78] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ASPLOS*, 2020.

[79] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. Panic: A high-performance programmable nic for multi-tenant networks. In *OSDI*, 2020.

[80] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[81] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *OSDI*, 2016.

[82] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[83] Rui Miao et al. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM*, 2017.

[84] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[85] Tian Pan et al. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *SIGCOMM*, 2021.

[86] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. In *ESA*, 2011.

[87] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[88] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vSwitch. In *NSDI*, 2015.

[89] Pramod Subba Rao and George Porter. Is memory disaggregation feasible? A case study with Spark SQL. In *ANCS*, 2016.

[90] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.

[91] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[92] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for Multi-Tenant cloud storage. In *OSDI*, 2012.

[93] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[94] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *NSDI*, 2022.

[95] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *NSDI*, 2022.

[96] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.*, 2021.

A Discussion

A.1 Complications in failover

It is possible for the ToR switches (shown in green in Figure 4), which probe the cards for liveness, to reach different liveness estimates for a card pair. That is, one of the switches can conclude a card is down while the other switch concludes otherwise. Similarly, even though we use multiple heartbeats and there are multiple network paths between a card pair, it is possible that so many consecutive heartbeats are lost allowing one of the cards in a card pair to conclude that the peer is down even though the peer is alive. A *split-brain* happens when different parts of the network assume that different cards (in a card pair) are responsible for an fNIC. Recall that the primary card announces a BGP route with a smaller AS path which helps resolve some of these complications. In addition, we notify all card role changes to a logically centralized Sirius controller which helps to ensure that *split-brain* cases, were they to happen, do not persist for very long.

A.2 fNIC abstraction guarantees

We statically partition each card's *capacity*, on the dimensions listed in §3, among the fNICs that are mapped to a card. The capacity values that we use for apportioning (e.g., the last row of Table 7) are slightly smaller than the maximum values that we see in experiments using iXIA traffic generators and a rich variety of network functions in subsection 6.2 and so we do not anticipate performance interference issues at the card.

A.3 Encryption, Traffic QoS

Some aspects such as end-to-end encryption and traffic prioritization can require on-host support. With Sirius, we are exploring how to divide such functions between the disaggregated pool and the smart NIC that is directly attached to the host. For example, the disaggregated pool can perform the more stateful processing such as exchanging keys or determining which queue to assign a flow to so as to meet its priority class or bandwidth limit while the on-host FPGA performs tasks that require less state such as marking or rate limiting queues [90].

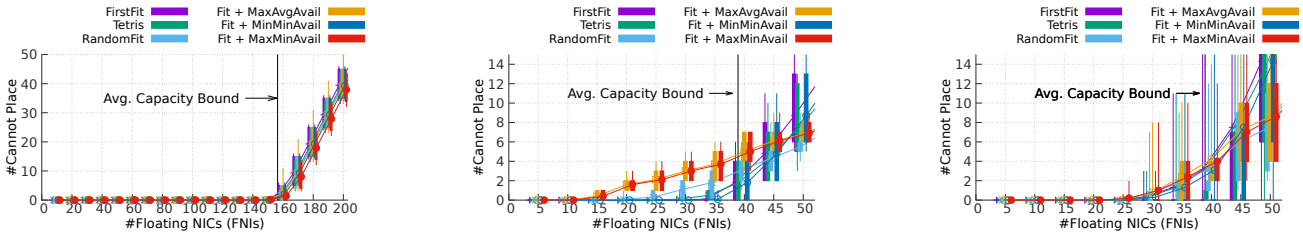
B Additional Results

B.1 Packing floating NICs into Sirius cards

Table 7 shows the sizes of the floating NICs that Sirius offers and the card capacity. A Sirius appliance has 12 cards. We evaluate several vector bin packing heuristics [61, 86] to pack fNICs onto Sirius cards. Our results in Figure 14 show that:

- The optimal choice of a packing heuristic, in terms of packing efficiency, depends on the size distribution of the fNICs and whether or not we split load of an fNIC across multiple cards.
- In some cases, such as when small fNICs dominate the workload, any heuristic can achieve the average capacity bound[12] which assumes that there are no card boundaries and that all resources are in one large pool.
- Splitting the load of an fNIC across cards substantially improves efficiency but also increases state that must be maintained on cards since rulesets belonging to split fNICs must now be deployed on multiple cards. The per flow and per endpoint state substantially dominates the ruleset size however. Nevertheless, we aim to split only fNICs that have large resource needs.

[12] $\text{bound} = \min_{\text{resource r}} \frac{\text{total capacity on r}}{\text{avg. FNI load on r}}$



(a) Packing efficiency when the fNICs listed in Table 7 arrive as per the production distribution from §2.2.

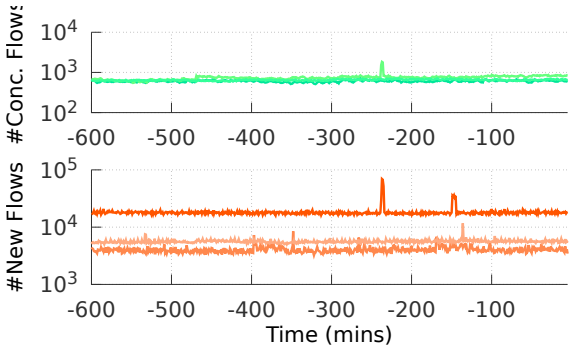
(b) Comparing the packing efficiency when the fNICs listed in Table 7 arrive with an equal probability.

(c) The case of Figure 14b except with the load from each fNIC split evenly across two cards.

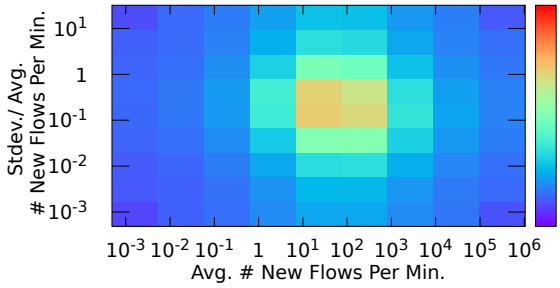
Figure 14: Comparing different vector bin packing strategies when mapping floating NICs (granularity of resource allocation for network functions supported by Sirius as listed in Table 7) to a Sirius appliance with 12 cards.

Term	Resource Sizing		
	#New Flows (Millions Per Sec.)	# Concurrent Flows (M)	Throughput (Gbps)
FNI_XXS	0.05	1	5.0
FNI_XS	0.10	1	10.0
FNI_S	0.25	2	12.5
FNI_M	1.00	2	12.5
FNI_L	2.00	4	12.5
FNI_XL	3.00	16	50.0
Sirius card	3.00	16	200.0

Table 7: The resource sizes, along multiple dimensions, that Sirius associates with different kinds of floating NICs. Cloud customers can choose which floating NIC to associate with their VM.



(a) Temporal changes in NF load; showing the total load at three randomly chosen nodes; note: y axes is in log scale.



(b) Clustering containers based on their average NF load and the coefficient of variability (=stdev/ avg) of their NF load.

Figure 15: Additional characterization results from the dataset in §2.2.

B.2 Variation in the load for NFs at Azure

We analyze load variability across all containers in Azure using the same dataset described in §2.2.

Figure 15a shows the temporal variation in the cumulative NF load. The figure shows many short-lived spikes, some of which are larger than 4×; note y axes is in log-scale. We also see some innate variability in the steady-state load across these nodes.

Figure 15b is a 2D matrix where each entry represents the number of containers that have the corresponding (x, y) value. The x axes is the average numbers of new flows in each minute. The y axes is the coefficient of variation in the same metric (=stdev./ avg.). Both axes are in log-scale. As well, the number of containers which is shown as a heat plot on the right is also in log scale. The figure shows that most of the containers have between 10 to 1000 new flows per minute and the coefficient of variation is typically between 0.1 and 1. While the variability is high for many containers, containers with more average load appear to only have slightly higher variability and there are no unexpected patterns.

C Additional Related Work

State replication for fault tolerance has received much attention; see [53, 64] for a review. Our method in §3.2 is different from the primary-backup style replication [44, 51] wherein the primary processes requests, forwards state changes to the backup and emits responses after receiving an acknowledgement from the backup. Our method is also different from Paxos-like protocols [46, 48, 50, 81] where replicas first agree on an order in which to process requests and then process the requests. The key difference is that both alternatives hold requests while replication is underway. In the case of stateful NFs, requests are packets that change state and so holding requests at speeds of hundreds of Gbps will require a very large packet buffer. To our knowledge, we are unaware of any prior work that ping-pong's packets between replicas which effectively holds the requests on the network wire. Redplane [72] replicates the state between programmable switches and a server-based remote state store but must cope in P4 with route changes that happen between the switches and the server-based store. Our method is simpler and more performant.

For the case of moving state on-the-fly between multiple NF processors, OpenNF [60] is perhaps the first to describe in detail the multiple issues that arise. Their solution however buffers all the packets that arrive while the state is being moved at an SDN controller (e.g., Floodlight [16]) which becomes a scaling bottleneck. OpenNF [60] reports results for O(100) flows (e.g., "a loss-free move involving state for 500 flows takes only 215ms") whereas each fNIC in Sirius can have many millions of ongoing flows. Similar to RedPlane [72], StatelessNF [69] and [71] store relevant NF state in an external state store (e.g., in a RAMCloud [69]). As noted above, relative to Sirius (which stores state in a nearby secondary card), we believe that storing state in an external state store has higher intrinsic overheads.

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

Minchen Yu[†] Tingjia Cao[‡*] Wei Wang[†] Ruichuan Chen[§]
[†]Hong Kong University of Science and Technology
[‡]University of Wisconsin-Madison [§]Nokia Bell Labs

Abstract

Serverless applications are typically composed of function workflows in which multiple short-lived functions are triggered to exchange data in response to events or state changes. Current serverless platforms coordinate and trigger functions by following high-level invocation dependencies but are oblivious to the underlying data exchanges between functions. This design is neither efficient nor easy to use in orchestrating complex workflows – developers often have to manage complex function interactions by themselves, with customized implementation and unsatisfactory performance.

In this paper, we argue that function orchestration should follow a *data-centric approach*. In our design, the platform provides a *data bucket abstraction* to hold the intermediate data generated by functions. Developers can use a rich set of data trigger primitives to control when and how the output of each function should be passed to the next functions in a workflow. By making data consumption explicit and allowing it to trigger functions and drive the workflow, complex function interactions can be easily and efficiently supported. We present `Pheromone` – a scalable, low-latency serverless platform following this data-centric design. Compared to well-established commercial and open-source platforms, `Pheromone` cuts the latencies of function interactions and data exchanges by orders of magnitude, scales to large workflows, and enables easy implementation of complex applications.

1 Introduction

Serverless computing, with its Function-as-a-Service incarnation, is becoming increasingly popular in the cloud. It allows developers to write highly scalable, event-driven applications as a set of short-running functions. Developers simply specify the events that trigger the activation of these functions, and let the serverless platform handle resource provisioning, autoscaling, logging, fault-tolerance, etc. Serverless computing is also economically appealing as it has zero idling cost: developers are only charged when their functions are running.

Many applications have recently been migrated to the serverless cloud [28, 35, 39, 43, 48, 59, 68, 75, 78]. These applications typically consist of multiple interactive functions with diverse function-invocation and data-exchange patterns. For example, a serverless-based batch analytics application may trigger hundreds of parallel functions for all-to-all data communication in a shuffle phase [49, 59, 79]; a stream processing application may repeatedly trigger certain functions to process dynamic data received in a recent time window. Ideally, a serverless platform should provide an *expressive* and *easy-to-use* function orchestration to support various function-invocation and data-exchange patterns. The orchestration should also be made *efficient*, enabling low-latency invocation and fast data exchange between functions.

However, function orchestration in current serverless platforms is neither efficient nor easy to use. It typically models a serverless application as a *workflow* that connects functions according to their invocation dependency [4, 11, 21, 24, 34, 51, 53, 66]. It specifies the order of function invocations but is *oblivious to when and how data are exchanged between functions*. Without such knowledge, the serverless platform assumes that the output of a function is entirely and immediately consumed by the next function(s), which is not the case in many applications such as the aforementioned "shuffle" operation in batch analytics and the processing of dynamically accumulated data in stream analytics. To work around these limitations, developers have to manage complex function interactions and data exchanges by themselves, using various approaches such as a message broker or a shared storage, either synchronously or asynchronously [6, 10, 24, 31, 49, 66]. As no single approach is found optimal in all scenarios, developers may need to write complex logic to dynamically select the most efficient approach at runtime (see §2.2). Current serverless platforms also incur function interaction latencies of tens of milliseconds, which may be unacceptable to latency-sensitive applications [46], particularly since this overhead accumulates as the function chain builds up.

In this paper, we argue that function orchestration should follow the flow of data rather than the function-level invocation dependency, thus a *data-centric approach*. Our key idea is to make data consumption explicit, and let it trigger functions and drive the workflow. In our design, the serverless platform exposes a *data bucket abstraction* that holds the intermediate output of functions in a logical object store. The data bucket provides a rich set of data trigger primitives that developers can use to specify *when* and *how* the intermediate

*This work was partially done while the author was at HKUST.

data are passed to the intended function(s) and trigger their execution. With such a fine control of data flow, developers can express sophisticated function invocations and data exchanges, simply by configuring data triggers through a unified interface. Knowing how intermediate data will be consumed also enables the serverless platform to schedule the intended downstream functions close to the input, thus ensuring fast data exchange and low-latency function invocation.

Following this design approach, we develop Pheromone[1], a scalable serverless platform with low-latency data-centric function orchestration. Pheromone proposes three key designs to deliver high performance. **First**, it uses a two-tier distributed scheduling hierarchy to locally execute a function workflow whenever possible. Each worker node runs a local scheduler, which keeps track of the execution status of a workflow via its data buckets and schedules next functions of the workflow onto local function executors. In case that all local executors are busy, the scheduler forwards the request to a global coordinator which then routes it to another worker node with available resources. **Second**, Pheromone trades the durability of intermediate data, which are typically short-lived and immutable, for fast data exchange. Functions exchange data within a node through a zero-copy shared-memory object store; they can also pass data to a remote function through direct data transfer. **Third**, Pheromone uses sharded global coordinators, each handling a disjoint set of workflows. With such a shared-nothing design, local schedulers only synchronize workflows' execution status with the corresponding global coordinators, which themselves require no synchronization, thus ensuring high scalability for distributed scheduling.

We evaluate Pheromone against well-established commercial and open-source serverless platforms, including AWS Lambda with Step Functions, Azure Durable Functions, Cloudburst [66], and KNIX [24]. Evaluation results show that Pheromone improves the function invocation latency by up to 10× and 450× over Cloudburst (best open-source baseline) and AWS Step Functions (best commercial baseline), respectively. Pheromone scales well to large workflows and incurs only millisecond-scale orchestration overhead when running 1k chained functions and 4k parallel functions, whereas the overhead is at least a few seconds in other platforms. Pheromone has negligible data-exchange overhead (e.g., tens of μs), thanks to its zero-copy data exchange. It can also handle failed functions through efficient re-execution. Case studies of two serverless applications, i.e., Yahoo! stream processing [40] and MapReduce sort, further demonstrate that Pheromone can easily express complex function interaction patterns (*rich expressiveness*), require no specific implementation to handle data exchange between functions (*high usability*), and efficiently support both latency-sensitive and data-intensive applications (*wide applicability*).

[1] Pheromone is a chemical signal produced and released into the environment by an animal that triggers a social response of others of its species. We use it as a metaphor for our data-centric function orchestration approach.

2 Background and Motivation

We first introduce serverless computing and discuss the limitations of function orchestration in current serverless platforms.

2.1 Serverless Computing

Serverless computing, with its popular incarnation being Function-as-a-Service (FaaS), has recently emerged as a popular cloud computing paradigm that supports highly-scalable, event-driven applications [8, 18, 22]. Serverless computing allows developers to write short-lived, stateless functions that can be triggered by events. The interactions between functions are simply specified as *workflows*, and the serverless platform manages resource provisioning, function orchestration, autoscaling, logging, and fault tolerance for these workflows. This paradigm appeals to many developers as it allows them to concentrate on the application logic without having to manage server resources [47, 63] – hence the name serverless computing. In addition to the high scalability and operational simplicity, serverless computing adopts a "pay-as-you-go" billing model: developers are billed only when their functions are invoked, and the function run-time is metered at a fine granularity, e.g., 1 ms in major serverless platforms [8, 18]. Altogether, these benefits have increasingly driven a large number of traditional "serverful" applications to be migrated to the serverless platforms, including batch analytics [39, 48, 59, 79], video processing [35, 43], stream processing [28], machine learning [75, 78], microservices [46], etc.

2.2 Limitations of Current Platforms

Current serverless platforms take a *function-oriented* approach to orchestrating and activating the functions of a serverless workflow: each function is treated as a single and standalone unit, and the interactions of functions are separately expressed within a workflow. This workflow connects individual functions according to their invocation dependencies, such that each function can be triggered upon the completion of one or multiple upstream functions. For example, many platforms model a serverless workflow as a directed acyclic graph (DAG) [4, 11, 21, 24, 34, 51, 53, 66], in which the nodes represent functions and the edges indicate the invocation dependencies between functions. The DAG can be specified using general programming languages [4, 21], or domain-specific languages such as Amazon States Language [11, 24]. However, this approach has several limitations with regard to expressiveness, usability, and applicability.

Limited expressiveness. Although the current function-oriented orchestration supports the workflows of simple invocation patterns, it becomes inconvenient or incapable of expressing more sophisticated function interactions, as summarized in Table 1. This is because the current function orchestration assumes that data flow in the same way as how

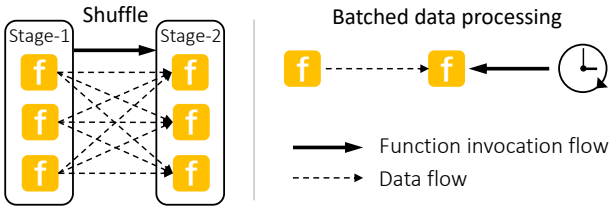


Figure 1: The shuffle operation (left) in data analytics and the batched data processing in a stream (right).

functions are invoked in a workflow, and that a function passes its entire output to others by directly invoking them for immediate processing. These assumptions do not hold for many applications, hence developers resort to create workarounds.

For example, the "shuffle" operation in a data analytics job involves a fine-grained, all-to-all data exchange between the functions of two stages (e.g., "map" and "reduce" stages). As shown in Fig. 1 (left), the output data of a function in stage-1 are shuffled and selectively redistributed to multiple functions in stage-2 based on the output keys. However, the way to invoke functions is not the same as how the output data flow: only after stage-1 completes can the workflow invoke all the stage-2 functions in parallel. In current serverless platforms, developers must manually implement such a complex data shuffle invocation via external storage [49, 59], which is neither flexible nor efficient.

Another example is a batched stream analytics job which periodically invokes a function to process the data continuously received during a time window [40, 73], as shown in Fig. 1 (right). A serverless workflow cannot effectively express this invocation pattern as the function is not immediately triggered when the data arrive, and thus developers have to rely on other cloud services (e.g., AWS Kinesis [7]) to batch the data for periodic function invocations [28–30]. Note that, even with the latest stateful workflow (e.g., Azure Durable Functions [17]), an addressable function needs to keep running to receive data. As we will show in §6.5, deploying a long-running function not only incurs extra resource provisioning cost but results in an unsatisfactory performance.

Limited usability. Current serverless platforms provide various options for data exchange between functions. Functions can exchange data either synchronously or asynchronously via a message broker or a shared storage [6,10,24,31,49,66]. They can also process data from various sources, such as nested function calls, message queues, or other cloud services [23].

The lack of a single best approach to exchange data between functions significantly complicates the development and deployment of serverless applications, as developers must find their own ways to efficiently pass data across functions [53] which can be dynamic and non-trivial; thus, reducing the usability of serverless platforms. To illustrate this problem, we compare four data-passing approaches in AWS Lambda: a) calling a function directly (Lambda), b) using AWS Step Functions (ASF) to execute a two-function workflow[2], c) allowing functions to access an in-memory Redis store for fast data exchange (ASF+Redis), and d) configuring AWS S3 to invoke a function upon data creation (S3) [32]. Fig. 2 compares the latencies of these four approaches under various data volumes. Lambda is efficient for transferring small data; ASF+Redis is efficient for transferring large data; the maximum data volume supported by each approach varies considerably, and only the S3 approach can support virtually unlimited (but slow) data exchange. Thus, there is no single approach that prevails across all scenarios, and developers must carefully profile the data patterns of their applications and the serverless platforms to optimize the performance of data exchange between interacting functions.

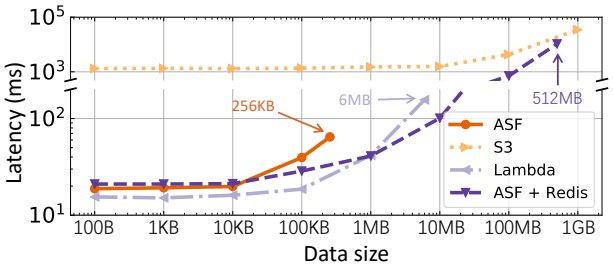


Figure 2: The interaction latency of two AWS Lambda functions under various data sizes using four approaches.

To make matters worse, the data volume exchanged between functions depends on the workload, which may be irregular or unpredictable. Thus, there may be no best *fixed* approach to exchanging data between interacting functions, and developers have to write complex logic to select the best approach at runtime. Developers also need to consider the interaction cost. Previous work has highlighted the tricky trade-off between I/O performance and cost when using different storage to share intermediate data [49, 59], which further exacerbates the usability issue. Altogether, these common practices bring a truly *non-serverless* experience to developers as they still have to deal with server and platform characteristics.

Limited applicability. Existing serverless applications are typically not latency-sensitive. This is because current serverless platforms usually have a function interaction delay of multiple or tens of milliseconds (§6.2), and such delays accumulate as more functions are chained together in an application workflow. For example, in AWS Step Functions, each function interaction causes a delay of more than 20 ms, and the total platform-incurred delay for a 6-function chain is over 100 ms, which may not be acceptable in many latency-sensitive applications [46]. In addition, as current serverless platforms cannot efficiently support the sharing of varying-sized data between functions (as described earlier), they are ill-suited for data-intensive applications [8, 24, 43, 46, 59, 66]. Altogether, the above characteristics substantially limit the applicability of current serverless platforms.

[2]We use the ASF Express Workflows in our experiments as it delivers higher performance than the ASF Standard Workflows [14].

3 Data-Centric Function Orchestration

In this section, we address the aforementioned limitations of the function orchestration practice in current serverless platforms, with a novel data-centric approach. We will describe how this approach can be applied to develop a new serverless platform later in §4.

3.1 Key Insight

As discussed in §2.2, the current function orchestration practice only specifies the high-level invocation dependencies between functions, and thus has little fine-grained control over how these functions exchange data. In particular, the current practice assumes the tight coupling between function flows and data flows. Therefore, when a function returns its result, the workflow has no knowledge about how the result should be consumed (e.g., in full or part, directly or conditionally, immediately or later). To address these limitations, *an effective serverless platform must allow fine-grained data exchange between the functions of a workflow, while simultaneously providing a unified and efficient approach for function invocation and data exchange*.

Following this insight, we propose a new *data-centric approach* to function orchestration. We note that intermediate data (i.e., results returned by functions) are typically short-lived and immutable [49, 67]: after they are generated, they wait to be consumed and then become obsolete.[3] We therefore make data consumption explicit and enable it to trigger the target functions. Developers can thus specify when and how intermediate data should be passed to the target functions and trigger their activation, which can then drive the execution of an entire workflow. As intermediate data are not updated once they are generated [49, 67], using them to trigger functions results in no consistency issues.

The data-centric function orchestration addresses the limitations of the current practice via three key advances. First, it breaks the tight coupling between function flows and data flows, as data do not have to follow the exact order of function invocations. It also enables a flexible and fine-grained control over data consumption, and therefore can express a rich set of workflow patterns (i.e., *rich expressiveness*). Second, the data-centric function orchestration provides a unified programming interface for both function invocations and data exchange, obviating the need for developers to implement complex logic via a big mix of external services to optimize data exchange (i.e., *high usability*). Third, knowing when and how the intermediate data will be consumed provides opportunities for the serverless platform scheduler to optimize the locality of functions and relevant data, and thus latency-sensitive and data-intensive applications can be supported efficiently (i.e., *wide applicability*).

[3] For data that need durability, they can be persisted to a durable storage.

Figure 3: An overview of triggering functions in data-centric orchestration. Source functions send intermediate data to the associated bucket, which can be configured to automatically trigger target functions.

3.2 Data Bucket and Trigger Primitives

Data bucket. To facilitate the data-centric function orchestration, we design a *data bucket* abstraction and a list of *trigger primitives*. Fig. 3 gives an overview of how functions are triggered. A serverless application creates one or multiple data buckets that hold the intermediate data. Developers can configure each bucket with triggers that specify when and how the data should invoke the target functions and be consumed by them. When executing a workflow, the source functions directly send their results to the specified buckets. Each bucket checks if the configured triggering condition is satisfied (e.g., the required data are complete and ready to be consumed). If so, the bucket triggers the target functions automatically and passes the required data to them. This process takes place across all buckets, which collectively drive the execution of an entire workflow.

We design various trigger primitives for buckets to specify how functions are triggered. The interaction patterns between functions can be broadly classified into three categories:

Direct trigger primitive (i.e., `Immediate`) allows one or more functions to directly consume data in the associated buckets. This primitive has no specified condition, and triggers the target functions immediately once the data are ready to be consumed. This primitive can easily support sequential execution or invoke multiple functions in parallel (fan-out).

Conditional trigger primitives trigger the target function(s) when the developer-specified conditions are met.

- `ByBatchSize`: It triggers the function(s) when the associated bucket has accumulated a certain number of data objects. It can be used to enable the batched stream processing [29, 30] in a way similar to Spark Streaming.
- `ByTime`: It sets up a timer and triggers the function(s) when the timer expires. All the accumulated data objects are then passed to the function(s) as input. It can be used to implement routine tasks [40, 73].
- `ByName`: It triggers the function(s) when the bucket receives a data object of a specified name. It can be used to enable conditional invocations by choice [12].
- `BySet`: It triggers functions when a specified set of data objects are all complete and ready to be consumed. It can be used to enable the assembling invocation (fan-in).
- `Redundant`: It specifies n objects to be stored in a bucket and triggers the function(s) when any k of them are

Table 1: Expressiveness comparison between the function-oriented workflow primitives in AWS Step Functions (ASF) and the data-centric trigger primitives in Pheromone.

Invocation Patterns	ASF	Pheromone
Sequential Execution	Task	Immediate
Conditional Invocation	Choice	ByName
Assembling Invocation	Parallel	BySet
Dynamic Parallel	Map	DynamicJoin
Batched Data Processing	-	ByBatchSize ByTime
k-out-of-*n*	-	Redundant
MapReduce	-	DynamicGroup

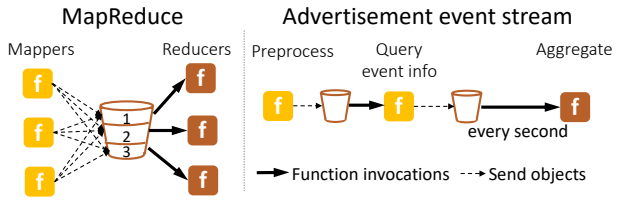


Figure 4: Usage examples of two primitives: DynamicGroup for data shuffling in MapReduce (left), and ByTime for periodic data aggregation in the event stream processing (right).

available and ready to be consumed. It can be used to execute redundant requests and perform late binding for straggler mitigation and improved reliability [50, 60, 69].

Dynamic trigger primitives, unlike the previous two categories with statically-configured triggers, allow data exchange patterns to be configured at runtime.

- DynamicJoin: It triggers the assembling functions when a set of data objects are ready, which can be dynamically configured at runtime. It enables the dynamic parallel execution like 'Map' in AWS Step Functions [13].
- DynamicGroup: It allows a bucket to divide its data objects into multiple groups, each of which can be consumed by a set of functions. The data grouping is dynamically performed based on the objects' metadata (e.g., the name of an object). Once a group of data objects are ready, they trigger the associated set of functions.

Dynamic trigger primitives are critical to implement some widely-used computing frameworks, e.g., MapReduce (which is hard to support in current serverless platforms as it requires triggering parallel functions at every stage and optimizing the fine-grained, all-to-all data exchange between them [48, 49, 59], see §2.2). Here, our DynamicGroup primitive provides an easy solution to these issues. As shown in Fig. 4 (left), when a map function sends intermediate data objects to the associated bucket, it also specifies to which data group each object belongs (i.e., by specifying their associated keys). Once the map functions are all completed, the bucket triggers the reduce functions, each consuming a group of objects.

We have developed a new serverless platform, Pheromone, which implements the aforementioned data bucket abstraction and trigger primitives. The design of Pheromone will be detailed in §4. Table 1 lists all the supported trigger primitives in current Pheromone platform. Compared to AWS Step Functions (ASF), Pheromone supports more sophisticated invocation patterns and provides richer expressiveness for complex workflows. We note that Azure Durable Functions [15] can also achieve rich expressiveness for complex workflows (§6.1). Yet, it fails to achieve the other two desired properties, i.e., high usability and wide applicability (§6.5).

```
struct BucketKey {
  string bucket_; // bucket name
  string key_; // key name
  string session_; // unique session id per request
};

abstract class Trigger {
  // Check whether to trigger functions for a new object.
  vector<TriggerAction> action_for_new_object(
      BucketKey bucket_key);

  // Notify the information of a source function.
  void notify_source_func(string function_name,
      string session, vector<string> function_args);

  // Check whether to re-execute source functions.
  vector<TriggerAction> action_for_rerun(string session);
};
```

Figure 5: Three main methods of the trigger interface.

Abstract interface. Pheromone's trigger primitives are not only limited to those listed in Table 1. Specifically, we provide an abstract interface for developers to implement customized trigger primitives for their applications, if needed. Fig. 5 shows the three main methods of the trigger interface. The first method, action_for_new_object, is provided to specify how the trigger's associated target functions should be invoked. This method can be called when a new data object arrives: it checks the current data status and returns a list of functions to invoke, if any. The method can also be called periodically in a configurable time period through periodical checking (e.g., ByTime primitive). The other two methods, notify_source_func and action_for_rerun, are provided to implement the fault handling logic which re-executes the trigger's associated source functions in case of failures. In particular, through notify_source_func, a trigger can obtain the information of a source function once the function starts, including the function name, session, and arguments; Pheromone also performs the periodic re-execution checks by calling action_for_rerun, which returns a list of time-out functions, such that Pheromone can then re-execute them. The detailed fault tolerance mechanism will be described in §4.4. We give an example of implementing a customized ByBatchSize trigger primitive via the abstract interface in our technical report [74].

3.3 Programming Interface

Our Pheromone serverless platform currently accepts functions written in C++, with capabilities to support more lan-

Table 2: The APIs of user library which developers use to operate on intermediate data objects and drive the workflow execution.

Class	API	Description
EpheObject	void* **get_value**()	Get a pointer to the value of an object.
	void **set_value**(value, size)	Set the value of an object.
UserLibrary	EpheObject* **create_object**(bucket, key)	Create an object by specifying its bucket and key name.
	EpheObject* **create_object**(function)	Create an object by specifying its target function.
	EpheObject* **create_object**()	Create an object.
	void **send_object**(object, output=false)	Send an object to its bucket, and set the output flag if it needs to persist.
	EpheObject* **get_object**(bucket, key)	Get an object by specifying its bucket and key name.

```
int handle(UserLibraryInterface* library,\
  int arg_size, char** arg_values);
```

Figure 6: Function interface.

```
1 app_name = 'event-stream-processing'
2 bucket_name = 'by_time_bucket'
3 trigger_name = 'by_time_trigger'
4 prim_meta = {'function':'aggregate', 'time_window':1000}
5 re_exec_rules = ([('query_event_info', EVERY_OBJ)], 100)
6 client.create_bucket(app_name, bucket_name)
7 client.add_trigger(app_name, bucket_name, trigger_name, \
8         BY_TIME, prim_meta, hints=re_exec_rules)
```

Figure 7: Configuring a bucket trigger to periodically invoke a function in a stream processing workflow.

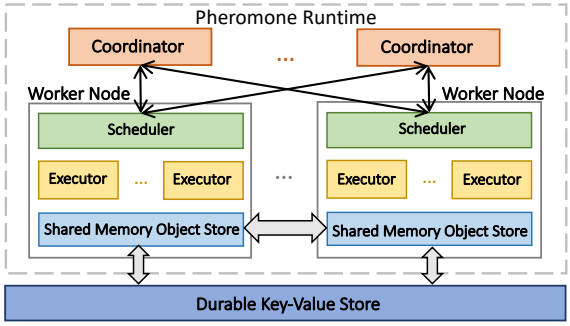


Figure 8: An architecture overview of Pheromone.

guages (see §7). Pheromone also provides a Python client through which developers can program function interactions.

Function interface. Following the common practice, developers implement their functions through the handle() interface (see Fig. 6), which is similar to the C++ main function except that it takes a user library as the first argument. The user library provides a set of APIs (see Table 2) that allow developers to operate on intermediate data objects. These APIs enable developers to create intermediate data objects (EpheObject), set their values, and send them to the buckets. A data object can also be persisted to a durable storage by setting the output flag when calling send_object(). When a bucket receives objects and decides to trigger next function(s), it automatically packages relevant objects as the function arguments (see Fig. 6). A function can also access other objects via the get_object() API.

Bucket trigger configuration. Developers specify how the intermediate data should trigger functions in a workflow via our Python client. The client creates buckets and configures triggers on the buckets using the primitives described in §3.2. Functions can then interact with the buckets by creating, sending and getting objects using the APIs listed in Table 2.

As an example, we refer to a stream processing workflow [40] as shown in Fig. 4 (right). This workflow first filters the incoming advertisement events (i.e., preprocess) and checks which campaign each event belongs to (i.e., query_event_info). It then stores the returned results into a bucket and periodically invokes a function (i.e., aggregate) to count the events per campaign every second. Fig. 7 gives a code snippet of configuring a bucket trigger that periodically invokes the aggregate function, where a ByTime trigger is created with the primitive metadata that specifies both the target function and the triggering time window (line 4). Developers can optionally specify a re-execution rule in case of function failures, e.g., by re-executing the query_event_info function if the bucket has not received query_event_info's output in 100 ms (line 5). We will describe the fault tolerance and re-execution in §4.4. A full script of deploying this workflow is given in our technical report [74].

To summarize, our data bucket abstraction, trigger primitives, and programming interface facilitate the data-centric function orchestration, and enable developers to conveniently implement their application workflows and express various types of data patterns and function invocations. In addition, the unified programming interface also obviates the need to make an ad-hoc selection from many APIs provided by various external services, such as a message broker, in-memory database, and persistent storage.

4 Pheromone System Design

This section presents the design of Pheromone, a new serverless platform that supports data-centric function orchestration.

4.1 Architecture Overview

Pheromone runs on a cluster of machines. Fig. 8 shows an architecture overview. Each worker node follows instructions from a local scheduler, and runs multiple executors that load and execute the user function code as needed. A worker node also maintains a shared-memory object store that holds the intermediate data generated by functions. The object store pro-

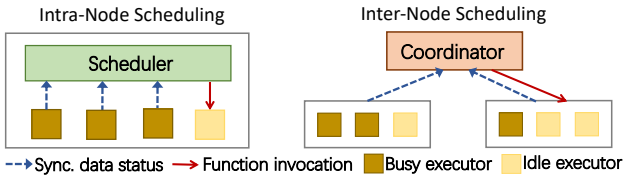


Figure 9: Intra-node (left) and inter-node (right) scheduling.

vides a data bucket interface through which functions can efficiently exchange data within a node and with other nodes. It also synchronizes data that must persist with a remote durable key-value store, such as Anna [71]. When new data are put into the object store, the local scheduler checks the associated bucket triggers. If the triggering conditions are satisfied, the local scheduler invokes the target function(s) either locally, or remotely with the help of a global coordinator that runs on a separate machine and performs cross-node coordination with a global view of bucket statuses.

4.2 Scalable Distributed Scheduling

We design a two-tier, distributed scheduling scheme to exploit data locality and ensure high scalability, enabled by the data-centric approach. Specifically, a workflow request first arrives at a global coordinator, which routes the request to a local scheduler on a worker node. The local scheduler invokes subsequent functions to locally execute the workflow whenever possible, thus reducing the invocation latency and incurring no network overhead.

Intra-node scheduling. In `Pheromone`, a local scheduler uses bucket triggers to invoke functions *as locally as possible*. The scheduler starts the first function of a workflow and tracks its execution status via its bucket. The downstream functions are triggered immediately on the same node when their expected data objects are put into the associated buckets and ready to be consumed. As no cross-node communication is involved, it reduces the function invocation latency and enables efficient consumption of data objects in a local workflow execution. Fig 9 (left) shows how the local scheduler interacts with executors when running a workflow locally. The executors synchronize the data status (e.g., the readiness of local objects in buckets) with the local scheduler, which then checks the associated bucket triggers and invokes downstream functions if the triggering conditions are met. The low-latency message exchange between the scheduler and executors is enabled via an on-node shared-memory object store.

A local scheduler makes scheduling decisions based on the status of executors. The scheduler only routes function requests to idle executors that have no running tasks, avoiding concurrent invocations and resource contention in each executor (similar to the concurrency model in AWS Lambda [9]). When the executor receives a request for the first time, it loads the function code from the local object store and persists it in memory for reuse in subsequent invocations. In case of multiple idle executors, the scheduler prioritizes those with function code already loaded to enable a warm start.[4]

Delayed request forwarding from overloaded nodes. If the requests received by a local scheduler exceed the capacity of local executors, the scheduler forwards them to a global coordinator, which routes them to other worker nodes with sufficient resources. Instead of forwarding the exceeding requests immediately, the scheduler waits for a configurable short time period: if any local executors become available during this period, the requested functions start and the requests are served locally. The rationale is that it typically takes little time for executors to become available as most serverless functions are short-lived [64], plus `Pheromone` has microsecond-scale invocation overhead (§6.2). Such a delayed scheduling has proven effective for improving data locality [77].

Inter-node scheduling. A global coordinator not only forwards requests from overloaded nodes to non-overloaded nodes, but also drives the execution of a large workflow which needs to run across multiple worker nodes that collectively host many functions of the workflow. This cannot be orchestrated by individual local schedulers without a global view.

As Fig. 9 (right) shows, a coordinator gathers the associated bucket statuses of the functions of a large workflow from multiple worker nodes, and triggers the next functions as needed. Each node immediately synchronizes local bucket status with the coordinator upon any change, such that the coordinator maintains an up-to-date global view. When the coordinator decides to trigger functions, it also updates this message to relevant workers, which reset local bucket status accordingly. This ensures a function invocation is neither missed nor duplicated. Note that, some bucket triggers (e.g., `ByTime`) can only be performed at the coordinator with its global view; here, worker nodes only update their local statuses to the coordinator without checking trigger conditions.

The data-centric orchestration improves data locality in the inter-node scheduling. The coordinator makes scheduling decisions using the node-level knowledge reported by local schedulers, including cached functions, the number of idle executors, and the number of objects relevant to the workflow. It then schedules a request to a worker node with sufficient warm executors and the most relevant data objects.

Scaling distributed scheduling with sharded coordinators. `Pheromone` employs a *shared-nothing* model to significantly reduce synchronization between local schedulers and global coordinators, thus attaining high scalability. Specifically, it partitions the workflow orchestration tasks across *sharded coordinators*, each of which manages a disjoint set of workflows. When executing a workflow, the responsible coordinator sends the relevant bucket triggers to a selected set of worker nodes and routes the invocation requests to them. A worker node may run functions of multiple workflows. For

[4] Many techniques have been proposed to deal with cold starts of executors [33, 37, 41, 44, 57, 64, 70], which can be applied directly in `Pheromone`.

each workflow, its data and trigger status are synchronized with the responsible coordinator only. This design substantially reduces communication and synchronization overheads, and can be achieved by running a standard cluster management service (e.g., ZooKeeper [5, 45]) that deals with coordinator failures and allows a client to locate the coordinator of a specific workflow. The client can then interact with this coordinator to configure data triggers and send requests. This process is automatically done by the provided client library and is transparent to developers.

4.3 Bucket Management and Data Sharing

We next describe how Pheromone manages data objects in buckets, and enables fast data sharing between functions.

Bucket management. Pheromone uses an on-node shared-memory object store to maintain data objects, such that functions can directly access them via pointers (i.e., EpheObject in Table 2). A data object is marked ready when the source function puts it into a bucket via send_object(). The bucket can be distributed across its responsible coordinator and a number of worker nodes, where each worker node tracks local data status while the coordinator holds a global view (§4.2). Bucket status synchronization is only needed between the responsible coordinator and workers, as local statuses at different workers track their local objects only and are disjoint.

Pheromone garbage-collects the intermediate objects of a workflow execution after the associated invocation request has been *fully* served along the workflow. In case a workflow is executed across multiple worker nodes, the responsible coordinator notifies the local scheduler on each node to remove the associated objects from its object store.

When a worker node's local object store runs out of memory, a remote key-value store is used to hold the newly generated data objects at the expense of an increased data access delay.[5] Later, when more memory space is made available (e.g., via garbage collection), the node remaps the associated buckets to the local object store. In case a data object is lost due to system failures, Pheromone automatically re-executes the source function(s) to get it recovered (details in §4.4).

Fast data sharing. Pheromone further adopts optimizations to fully reap the benefits of data locality enabled by its data-centric design. As intermediate data are typically short-lived and immutable [49, 67], we trade their durability for fast data sharing and low resource footprint. With an on-node shared-memory object store, Pheromone enables *zero-copy* data sharing between local functions by passing only the pointers of data objects to the target functions. This avoids the significant data copying and serialization overheads, and substantially reduces the latency of accessing local data objects.

To efficiently pass data to remote functions, Pheromone also enables the *direct* transfer of data objects between nodes. A function packages the metadata (e.g., locator) of a data object into a function request being sent to a remote node. The target function on the remote node uses such metadata to directly retrieve the required data object. Compared with using a remote storage for cross-node data sharing, our direct data transfer avoids unnecessary data copying, and thus leads to reduced network and storage overheads. While the remote-storage approach can ensure better data durability and consistency [24, 65, 66], there is no such need for intermediate data objects. Only when data are specified to persist will Pheromone synchronize data objects with a durable key-value store (see send_object() in Table 2).

Note that, Pheromone's data-centric design can expose details of intermediate data (e.g., the size of each data object), therefore we can further optimize cross-node data sharing. For large data objects, they are sent as raw byte arrays to avoid serialization-related overheads, thus significantly improving the performance of transferring large objects (see Fig. 13 in §6.2). For small data objects, Pheromone implements a shortcut to transfer them between nodes: it piggybacks small objects on the function invocation requests forwarded during the inter-node scheduling (see §4.2). This shortcut saves one round trip as the target function does not need to additionally retrieve data objects from the source function. In addition, Pheromone runs an I/O thread pool on each worker node to improve cross-node data sharing performance.

4.4 Fault Tolerance

Pheromone sustains various types of system component failures. In case an executor fails or a data object is lost, Pheromone restarts the failed function to reproduce the lost data and resume the interrupted workflow. This is enabled by using the data bucket to re-execute its source function(s) if the expected output has not been received in a configurable timeout. This fault handling approach is a natural fit for data-centric function orchestration and brings two benefits. First, it can simplify the scheduling logic as data buckets can autonomously track the runtime status of each function and issue re-execution requests whenever necessary, without needing schedulers to handle function failures. Second, it allows developers to customize function re-execution rules when configuring data buckets, e.g., timeout. Fig. 7 gives an example of specifying re-execution rules (line 5). Fig. 5 shows the interface to implement the logic of function re-execution for a bucket trigger (notify_source_func and action_for_rerun).

Pheromone also checkpoints the scheduler state (e.g., the workflow status) to the local object store, so that it can quickly recover from a scheduler failure on a worker node. In case that an entire worker node crashes, Pheromone re-executes the failed workflows on other worker nodes. Pheromone can also handle failed coordinators with a standard cluster management service, such as ZooKeeper, as explained in §4.2.

[5] Our current implementation does not support spilling in-memory objects to disk, which we leave for future work.

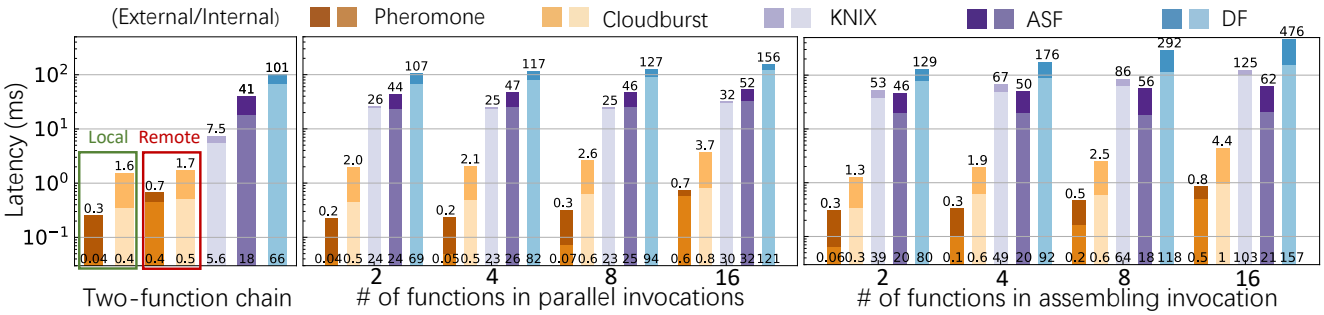


Figure 10: Latencies of invoking no-op functions under three interaction patterns: function chain, parallel and assembling invocations. Each bar is broken into two parts which measure the latencies of external (darker) and internal (lighter) invocations, respectively. The overall latency value is given at the top of the bar, and the internal invocation latency is given at the bottom.

5 Implementation

We have implemented Pheromone atop Cloudburst [66], a lightweight, performant serverless platform. We heavily re-architected Cloudburst and implemented Pheromone's key components (Fig. 8) in 5k lines of C++ code. These components were packaged into Docker [19] images for ease of deployment. Pheromone's client was implemented in 400 lines of Python code. Like Cloudburst, Pheromone runs in a Kubernetes [25] cluster for convenient container management, and uses Anna [71, 72], an autoscaling key-value store, as the durable key-value storage. On each worker node, we mount a shared in-memory volume between containers for fast data exchange and message passing. The executor loads function code as dynamically linked libraries, which is pre-compiled by developers and uploaded to Pheromone. The entire codebase of Pheromone is open-sourced at [26].

6 Evaluation

In this section, we evaluate Pheromone via a cluster deployment in AWS EC2. Our evaluation answers three questions:

- How does Pheromone improve function interactions (§6.2) and ensure high scalability (§6.3)?
- Can Pheromone effectively handle failures (§6.4)?
- Can developers easily implement real-world applications with Pheromone and deliver high performance (§6.5)?

6.1 Experimental Setup

Cluster settings. We deploy Pheromone in an EC2 cluster. The coordinators run on the `c5.xlarge` instances, each with 4 vCPUs and 8 GB memory. Each worker node is a `c5.4xlarge` instance with 16 vCPUs and 32 GB memory. The number of executors on a worker node is configurable and we tune it based on the requirements of our experiments. We deploy up to 8 coordinators and 51 worker nodes, and run clients on separate instances in the same `us-east-1a` EC2 zone.

Baselines. We compare Pheromone with four baselines.

1) Cloudburst: As an open-source platform providing fast state sharing, Cloudburst [66] adopts *early binding* in scheduling: it schedules all functions of a workflow before serving a request, and enables direct communications between functions. It also uses function-collocated caches. As Pheromone's cluster setting is similar to Cloudburst's, we deploy the two platforms using the same cluster configuration and resources.

2) KNIX: As an evolution of SAND [34], KNIX [24] improves the function interaction performance by executing functions of a workflow as processes in the same container. Message passing and data sharing can be done either via a local message bus or via a remote persistent storage.

3) AWS Step Functions (ASF): We use ASF Express Workflows [14] to orchestrate function instances as it achieves faster function interactions than the ASF Standard Workflows [14]. As ASF has a size limit of transferring intermediate data (see Fig. 2), we use Redis [6], a fast in-memory storage service, to share large data objects between functions.

4) Azure Durable Functions (DF): Compared with ASF, DF provides a more flexible support for function interactions. It allows developers to express workflows in program code and offers the Entity Functions [17] that can manage workflow states following the actor model [36, 54]. We include DF to study whether this expressive orchestration approach can achieve satisfactory performance.

Here, Cloudburst, KNIX and ASF focus more on optimizing function interactions of a workflow, while DF provides rich expressiveness. Note that, for the two commercial platforms, i.e., ASF and DF, we cannot control their orchestration runtime. To make a fair comparison, we configure their respective Lambda and Azure functions such that the number of function instances matches that of executors in Pheromone. The resource allocations of each function instance and executor are also maintained the same. In our experiments, functions are all warmed up to avoid cold starts in all platforms.

6.2 Function Interaction

Function invocation under various patterns. We first evaluate the overhead of invoking no-op functions without any

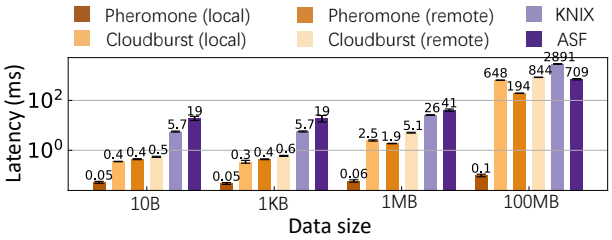


Figure 11: Latencies of a two-function chain invocation under various data sizes.

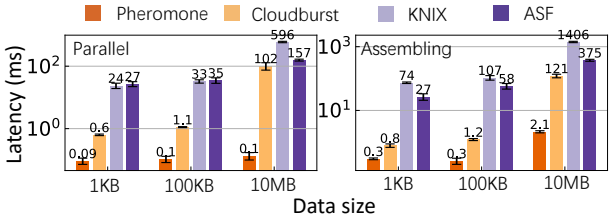


Figure 12: Latencies of parallel (left) and assembling (right) invocations under various data sizes, using 8 functions.

payload. We consider three common invocation patterns: sequential execution (e.g., a two-function chain), parallel invocation (fan-out), and assembling invocation (fan-in). We vary the number of involved functions for parallel and assembling invocations to control the degree of parallelism. Fig. 10 shows the latencies of invoking no-op functions under these three patterns. Each latency bar is further broken down into the overheads of external and internal invocations. The former measures the latency between the arrival of a request and the complete start of the workflow, and the latter measures the latency of internally triggering the downstream function(s) following the designated pattern. In Pheromone, the external invocation latency is mostly due to the overhead of request routing which takes about 200 μs [20]. Note that, functions can be invoked locally or remotely in Pheromone and Cloudburst, thus we measure them respectively in Fig. 10. In our experiments, we report the average latency over 10 runs.

Fig. 10 (left) compares the invocation latencies of a two-function chain measured on five platforms. Pheromone substantially outperforms the others. In particular, Pheromone's shared memory-based message passing (§4.3) only incurs an overhead of less than 20 μs, reducing the local invocation latency to 40 μs, which is 10× faster than Cloudburst. The latency improvements become significantly more salient compared with other platforms (e.g., 140× over KNIX, 450× over ASF). When invoking a remote function, both Pheromone and Cloudburst require network transfer, leading to a similar internal invocation latency. Yet, Cloudburst incurs higher overhead than Pheromone for external invocations as it needs to schedule the entire workflow's functions before serving a request (early binding), thus resulting in worse overall performance.

Fig. 10 (center) and (right) show the invocation latencies under parallel and assembling invocations, respectively. We also evaluate the cross-node function invocations in Pheromone and Cloudburst by configuring 12 executors on each worker, thus forcing remote invocations when running 16 functions. Pheromone constantly achieves the best performance and incurs only sub-millisecond latencies in all cases, even for cross-node function invocations. In contrast, Cloudburst's early-binding design incurs a much longer latency for function invocations as the number of functions increases. Both KNIX and ASF incur high invocation overheads in the parallel and assembling scenarios. DF yields the worst performance, and we exclude it from the experiments hereafter.

Data transfer. We next evaluate the interaction overhead when transferring data between functions. Fig. 11 shows the invocation latencies of a two-function chain with various data sizes in Pheromone, Cloudburst, KNIX, and ASF. We evaluate both local and remote data transfer for Pheromone and Cloudburst. For KNIX and ASF where the data transfer can be done via either a workflow or a shared storage (i.e., Riak and Redis), we report the best of the two choices.

For local data transfer, Pheromone enables zero-copy data sharing, leading to extremely low overheads regardless of the data size, e.g., 0.1 ms for 100 MB data. In comparison, Cloudburst needs the data copying and serialization, causing much longer latencies especially for large data objects. For remote data transfer, both Pheromone and Cloudburst support direct data sharing across worker nodes. Pheromone employs an optimized implementation without (de)serialization, making it more efficient than Cloudburst. Collectively, compared with Pheromone, the serialization overhead in Cloudburst dominants the latencies of both local and remote invocations under large data exchanges, which diminishes the performance benefits of data locality: saving the cost of transferring 100 MB data across network only reduces the latency from 844 ms to 648 ms. Fig. 11 also shows that KNIX and ASF incur much longer latencies. While KNIX outperforms ASF when data objects are small, ASF becomes more efficient for passing large data because it is configured in our experiments to use the fast Redis in-memory storage for large data transfer.

We further evaluate the overhead of data transfer under parallel and assembling invocations. For parallel invocation, we measure the latency of a function invoking parallel downstream functions and passing data to all of them; for assembling invocation, we measure the latency between the transfer of the first object and the reception of all objects in the assembling function. Fig. 12 shows the latencies of these two invocation patterns under various data sizes. Similarly, Pheromone constantly achieves faster data transfer compared with all other platforms for both invocation patterns.

Improvement breakdown. To illustrate how each of our individual designs contributes to the performance improvement, we break down Pheromone's function invocation performance and depict the results in Fig. 13. Specifically, for local invocation, "Baseline" uses a central coordinator to in-

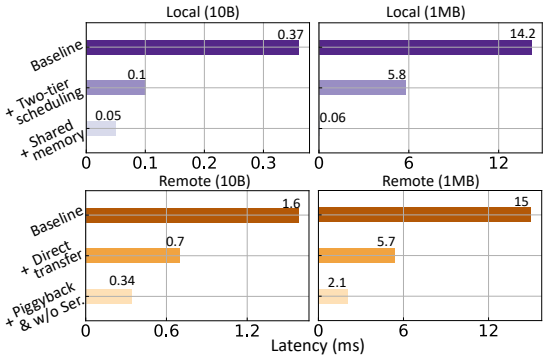


Figure 13: Improvement breakdown for local (top) and remote (bottom) invocations. Each case includes transferring 10 B (left) and 1 MB (right) of data in function invocations.

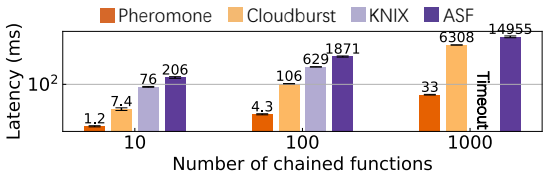


Figure 14: Latencies of function chains of different lengths.

voke downstream functions (i.e., no local schedulers), which is today's common practice [11]; "Two-tier scheduling" uses our local schedulers for fast invocations on the same worker node (§4.2), where intermediate data objects are cached in the scheduler's memory and get copied to next functions; "Shared memory" further optimizes the performance via zero-copy data sharing (§4.3). Fig. 13 (top) shows that applying two-tier scheduling can reduce network round trips and achieve up to 3.7× latency improvement over "Baseline". Applying shared memory avoids data copy and serialization, further speeding up the data exchange especially for large objects (e.g., 1 MB) by two orders of magnitude.

For remote invocation, "Baseline" uses a durable key-value store (i.e., Anna [71]) to exchange intermediate data among cross-node functions; "Direct transfer" reduces the communication overhead by allowing direct data passing between nodes (§4.3), where raw data objects on a node are serialized into a protobuf [27] message and then sent to downstream functions; "Piggyback & w/o Ser." further optimizes the data exchange by piggybacking small objects on forwarded function requests and eliminating serialization (§4.3). As shown in Fig. 13 (bottom), direct data transfer avoids interactions with the remote storage and improves the performance by up to 2.6× compared with baseline. The piggyback without serialization further speeds up the remote invocations with small (10 B) and large (1 MB) objects by 2× and 2.7×, respectively.

6.3 Scalability

We next evaluate the scalability of Pheromone with regard to internal function invocations and external user requests.

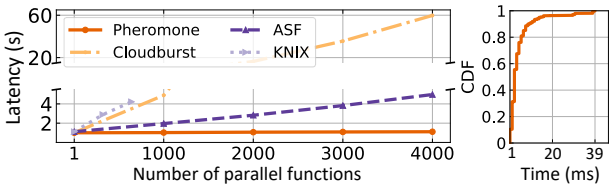


Figure 15: End-to-end latencies with various numbers of parallel functions (left), and the distribution of function start times when executing 4k functions in Pheromone (right).

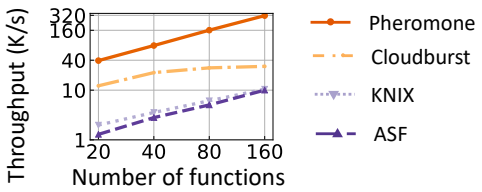


Figure 16: Request throughput when serving requests to no-op functions under various numbers of functions or executors.

Long function chain. We start with a long function chain that sequentially invokes a large number of functions [76]. Here, each function simply increments its input value by 1 and sends the updated value to the next function, and the final result is the total number of functions. As shown in Fig. 14, we change the number of chained functions, and Pheromone achieves the best performance at any scale. Moreover, Cloudburst suffers from poor scalability due to its early-binding scheduling, causing significantly longer latencies when the number of chained functions increases; KNIX cannot host too many function processes in a single container, making it ill-suited for long function chains; ASF incurs the longest latencies due to its high overhead of function interactions.

Parallel functions. Fig. 15 (left) evaluates the end-to-end latencies of invoking various numbers of parallel functions, where each function sleeps 1 second. We run 80 function executors per node in Pheromone and Cloudburst. Pheromone only incurs a negligible latency in large-scale parallel executions, while ASF and Cloudburst incur much higher latencies, e.g., seconds or tens of seconds. KNIX suffers from severe resource contention when running all workflow functions in the same container, and fails to support highly parallel function executions. To further illustrate Pheromone's behavior in parallel invocations, Fig. 15 (right) shows the distribution of function start times, where Pheromone can quickly launch all 4k functions within 40 ms.

User request throughput. Fig. 16 shows the user request throughput when serving requests to no-op functions using various numbers of executors. We configure 20 executors on each node in Pheromone and Cloudburst. We observe that Cloudburst's schedulers can easily become the bottleneck under a high request rate, making it difficult to fully utilize the executor's resources; KNIX suffers from a similar problem that limits its scalability. While ASF has no such an issue, it

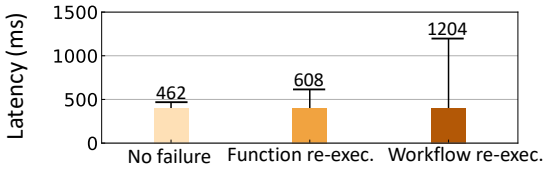


Figure 17: Median and 99th tail latencies of a four-function workflow with no failure, function- and workflow-level re-executions. The numbers indicate the tail latencies.

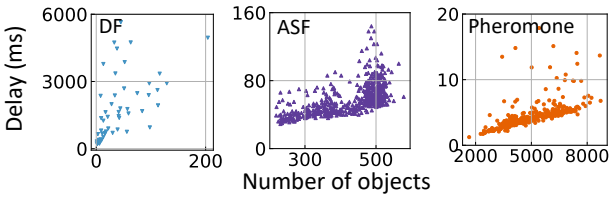


Figure 18: Delays of accessing the accumulated data objects in the advertisement event stream processing. Lower delays and more objects are better.

leads to low throughput due to its high invocation overhead (Fig. 10). Compared with these platforms, Pheromone ensures better scalability with the highest throughput.

6.4 Fault Tolerance

We evaluate Pheromone's fault tolerance mechanism (§4.4). We execute a workflow that chains four sleep functions (each sleeps 100 ms), and each running function is configured to crash at a probability of 1%. Fig. 17 shows the median and 99th tail latencies of the workflow over 100 runs using Pheromone's function- and workflow-level re-executions after a configurable timeout. In particular, the timeout is configured as twice of the normal execution, i.e., 200 ms for each individual function and 800 ms for the workflow. We also include the normal scenario where no failure occurs. Compared with the common practice of workflow re-execution, Pheromone's data-centric mechanism allows finer-grained, function-level fault handling, which cuts the tail latency of the workflow from 1204 ms to 608 ms, thus significantly reducing the re-execution overhead.

6.5 Case Studies

We evaluate two representative applications atop Pheromone: Yahoo's streaming benchmark for advertisement events [40], and a data-intensive MapReduce sort.

Advertisement event stream processing. This application filters incoming advertisement events (e.g., click or purchase), checks which campaign each event belongs to, stores them into storage, and periodically counts the events per campaign every second. As shown in Fig. 1 (right) and discussed in §2.2, the key to enabling this application in serverless platforms is to periodically invoke a function to process the events accumulated during the past one second.

In Pheromone, this is straightforward by using the ByTime primitive (§3.3 and Fig. 7). This application can also be implemented easily in DF by using an addressable Entity function for aggregation [16]. However, it is non-trivial in ASF and we have to resort to a "serverful" workaround: one workflow does the regular "filter-check-store" for each event and sends the event ID to an external, serverful coordinator; a separate workflow is set up to get triggered every second by the accumulated event IDs sent from the external coordinator, so that it can access and count the associated events per campaign.

Fig. 18 compares the performance on Pheromone, ASF, and DF. We measure the delays of accessing accumulated data objects (i.e., advertisement events), where the lower delays and more objects are better. For DF, data are not accessed in batches, and thus we measure the queuing delay between the reset request being issued and the Entity function receiving it. We use up to 40 functions in all these platforms. DF results in a significant overhead with high and unstable queuing delays, as its Entity function can easily become a bottleneck. Among the three platforms, Pheromone performs the best: it can access substantially more accumulated data objects in a much smaller delay. In summary, Pheromone not only simplifies the design and deployment for such a stream processing application, but also delivers high performance.

MapReduce sort. We next evaluate how Pheromone's data-centric orchestration can easily facilitate MapReduce sort, a typical data-intensive application. We have built a MapReduce framework atop Pheromone, called Pheromone-MR. Using the DynamicGroup primitive, Pheromone-MR can be implemented in only 500 lines of code, and developers can program standard mapper and reducer [2] without operating on intermediate data (§3.2). We compare Pheromone-MR with PyWren [48], a specialized serverless analytics system built atop AWS Lambda. Compared with Pheromone-MR, PyWren is implemented in about 6k lines of code and supports the map operator only, making it complicated to deploy a MapReduce application: developers need to explicitly transfer the intermediate data via a shared storage (e.g, Redis) to simulate the reducer, and carefully configure the storage cluster for improved data exchange. Even with these optimizations, PyWren still suffers from limited performance (and usability).

We evaluate the performance of Pheromone-MR and PyWren with MapReduce sort over 10 GB data, where 10 GB intermediate objects are generated in the shuffle phase. We allocate each Pheromone executor and each Lambda instance the same resource, e.g., 1 vCPU. We also configure a Redis cluster for PyWren to enable fast data exchange. We measure the end-to-end latencies on Pheromone-MR and PyWren using various numbers of functions, and break down the results into the function interaction latency and the latency for compute and I/O. The former measures the latency between the completion of mappers and the start of reducers. For PyWren,

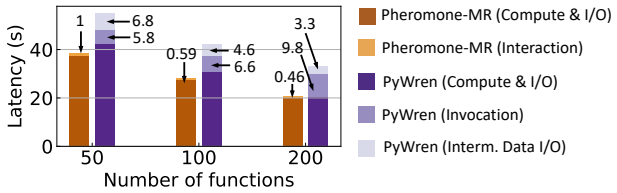


Figure 19: Latencies of sorting 10 GB data using various numbers of functions on PyWren and Pheromone-MR. The latency is broken down into: the interaction latency (for PyWren, the invocation and intermediate data I/O), and the latency for compute and I/O. The numbers indicate the former.

the interaction latency consists of two parts: 1) the invocation latency of triggering all reducers after mappers return, and 2) the I/O latency of sharing intermediate data via Redis. As shown in Fig. 19, running more functions in PyWren improves the I/O of sharing intermediate data, but results in a longer latency in parallel invocations. Compared with PyWren, Pheromone-MR has a significantly lower interaction latency (e.g., less than 1s), thus improving the end-to-end performance by up to 1.6×.

We note that, the limitations of AWS Lambada make PyWren less efficient. First, because Lambda does not support large-scale map by default [13], it needs to implement this operation but in an inefficient way which incurs high invocation overheads. Second, Lambda has a limited support for data sharing, forcing developers to explore an external alternative that incurs high overheads even though using a fast storage (i.e., Redis). Unlike AWS Lambda, Pheromone supports rich patterns of function executions while enabling fast data sharing, such that developers can easily build a MapReduce framework and achieve high performance.

7 Discussion and Related Work

Isolation in Pheromone. Pheromone provides the container-level isolation between function invocations, while functions running on the same worker node share in-memory data objects (§4.3). Commercial container-based serverless platforms often do not co-locate functions from different users to enhance security [1]. In this setting, functions on the same worker node can be trusted; hence, it is safe to trade strict isolation for improved I/O performance. We notice that current serverless platforms have made various trade-offs between performance and isolation. For example, AWS Lambda runs functions in MicroVMs for strong isolation [33]; KNIX isolates a workflow's functions using processes in the same container for better performance [34]; recent work proposes lightweight isolation for privacy-preserving serverless applications [52]. Pheromone can explore these different trade-offs, which we leave for future work.

Supported languages. Pheromone currently supports functions written in C++, but it can be straightforward to support other programming languages. Specifically, Pheromone's executor exposes data trigger APIs (Tabel 2) and interacts with other system components, and can serve as a proxy for functions written in different languages. That being said, Pheromone's optimization on fast data exchange via shared memory may not apply to all language runtimes – only those allowing the direct consumption of byte arrays without (de)serialization, e.g., Python ctype, can benefit from zero-copy data sharing. The other Pheromone designs are still effective regardless of language runtimes.

Data exchange in serverless platforms. Data exchange is a common pain point in today's serverless platforms. One general approach is to leverage shared storage to enable and optimize data exchange among functions [38,42,43,49,56,58,59]. One other approach is to exploit data locality to improve performance, e.g., by placing workflow functions on a single machine [46, 51, 53, 65–67]. Moreover, OFC [55] and Faa$T [61] provide the autoscaling cache for individual applications. Shredder [80] and Zion [62] push the function code into storage. Wukong [39] enhances the locality of DAG-based parallel workloads at the application level. Lambdata [67] makes the intent of a function's input and output explicit for improved locality; however, it does not provide a unified programming interface for expressive and simplified function interactions, and its performance is heavily bound to Apache OpenWhisk [3,51].

8 Conclusion

This paper revisits the function orchestration in serverless computing, and advocates a new design paradigm that a serverless platform needs to: 1) break the tight coupling between function flows and data flows, 2) allow fine-grained data exchange between functions of a workflow, and 3) provide a unified and efficient approach for both function invocations and data exchange. With this data-centric paradigm, we have designed and developed Pheromone, a new serverless platform which achieves all the desired properties, namely, rich expressiveness, high usability, and wide applicability. Pheromone is open-sourced, and outperforms current commercial and open-source serverless platforms by orders of magnitude in terms of the latencies of function invocation and data exchange.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Yiying Zhang, for their insightful comments that helped shape the final version of this work. We also thank Yuheng Zhao for his help in experiments, and Chenggang Wu for his valuable feedback at the early stage of this work. This work was supported in part by RGC GRF Grants 16202121 and 16210822. Minchen Yu was supported in part by the Huawei PhD Fellowship Scheme.

References

[1] Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[2] Apache Hadoop. https://hadoop.apache.org.

[3] Apache OpenWhisk. http://openwhisk.apache.org/.

[4] Apache OpenWhisk Composer. https://github.com/apache/openwhisk-composer/.

[5] Apache ZooKeeper. https://zookeeper.apache.org/.

[6] AWS ElastiCache. https://aws.amazon.com/elasticache/.

[7] AWS Kinesis. https://aws.amazon.com/kinesis/.

[8] AWS Lambda. https://aws.amazon.com/lambda/.

[9] AWS Lambda execution model. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html.

[10] AWS S3. https://aws.amazon.com/s3/.

[11] AWS Step Functions. https://aws.amazon.com/step-functions/.

[12] AWS Step Functions - Choice. https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-choice-state.html.

[13] AWS Step Functions - Map. https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-map-state.html.

[14] AWS Step Functions Standard vs. Express Workflows. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html.

[15] Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/.

[16] Azure Durable Functions aggregator pattern. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview#aggregator.

[17] Azure Entity Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities/.

[18] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[19] Docker. https://www.docker.com.

[20] Firecracker network performance numbers. https://github.com/firecracker-microvm/firecracker/blob/main/docs/network-performance.md.

[21] Google Cloud Composer. https://cloud.google.com/composer.

[22] Google Cloud Functions. https://cloud.google.com/functions.

[23] Invoking AWS Lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html.

[24] KNIX Serverless. https://github.com/knix-microfunctions/knix/.

[25] Kubernetes: Production-grade container orchestration. http://kubernetes.io.

[26] Pheromone codebase. https://github.com/MincYu/pheromone.

[27] Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf.

[28] Serverless applications scenarios. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/scenarios.html.

[29] Serverless reference architecture: Real-time stream processing. https://github.com/aws-samples/lambda-refarch-streamprocessing/.

[30] Serverless stream-based processing for real-time insights. https://aws.amazon.com/blogs/architecture/serverless-stream-based-processing-for-real-time-insights/.

[31] Use Amazon S3 ARNs instead of passing large payloads. https://docs.aws.amazon.com/step-functions/latest/dg/avoid-exec-failures.html.

[32] Using AWS Lambda with Amazon S3. https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html.

[33] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proc. USENIX NSDI*, 2020.

[34] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proc. USENIX ATC*, 2018.

[35] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proc. ACM SoCC*, 2018.

[36] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proc. ACM SoCC*, 2011.

[37] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proc. ACM EuroSys*, 2020.

[38] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ML workflows. In *Proc. ACM SoCC*, 2019.

[39] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proc. ACM SoCC*, 2021.

[40] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Proc. IEEE IPDPSW*, 2016.

[41] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proc. ACM ASPLOS*, 2020.

[42] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proc. USENIX ATC*, 2019.

[43] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. USENIX NSDI*, 2017.

[44] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proc. ACM ASPLOS*, 2021.

[45] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*, 2010.

[46] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proc. ACM ASPLOS*, 2021.

[47] Eric Jonas, Anurag Khandelwal, Karl Krauth, Johann Schleier-Smith, Qifan Pu, Neeraja Yadwadkar, Ion Stoica, Vikram Sreekanti, Vaishaal Shankar, Joseph E Gonzalez, David A Patterson, Chia-Che Tsai, Joao Carreira, and Raluca Ada Popa. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[48] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proc. ACM SoCC*, 2017.

[49] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proc. USENIX OSDI*, 2018.

[50] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proc. ACM SOSP*, 2019.

[51] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *Proc. USENIX ATC*, 2021.

[52] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proc. ACM/IEEE ISCA*, 2021.

[53] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, and Ana Klimovic. SONIC: Application-aware data passing for chained serverless applications. In *Proc. USENIX ATC*, 2021.

[54] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proc. USENIX OSDI*, 2018.

[55] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: an opportunistic caching system for FaaS platforms. In *Proc. ACM EuroSys*, 2021.

[56] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proc. ACM SIGMOD*, 2020.

[57] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proc. USENIX ATC*, 2018.

[58] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proc. ACM SIGMOD*, 2020.

[59] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proc. USENIX NSDI*, 2019.

[60] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. USENIX OSDI*, 2016.

[61] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proc. ACM SoCC*, 2021.

[62] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. Data-driven serverless functions for object storage. In *Proc. ACM/IFIP/USENIX Middleware*, 2017.

[63] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, 2021.

[64] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. USENIX ATC*, 2020.

[65] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proc. USENIX ATC*, 2020.

[66] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. In *Proc. VLDB Endow.*, 2020.

[67] Yang Tang and Junfeng Yang. Lambdata: Optimizing serverless computing by making data intents explicit. In *Proc. IEEE CLOUD*, 2020.

[68] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proc. ACM SoCC*, 2022.

[69] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *Proc. ACM HotNet*, 2012.

[70] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proc. USENIX ATC*, 2021.

[71] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: A KVS for any scale. In *Proc. IEEE ICDE*, 2018.

[72] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. In *Proc. VLDB Endow.*, 2019.

[73] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *Proc. USENIX NSDI*, 2021.

[74] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. *arXiv preprint arXiv:2109.13492*, 2021.

[75] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proc. IEEE ICDCS*, 2021.

[76] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proc. ACM SoCC*, 2020.

[77] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. ACM EuroSys*, 2010.

[78] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

[79] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *Proc. USENIX NSDI*, 2021.

[80] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proc. ACM SoCC*, 2019.

Doing More with Less: Orchestrating Serverless Applications without an Orchestrator

David H. Liu
Princeton University

Amit Levy
Princeton University

Shadi Noghabi
Microsoft Research

Sebastian Burckhardt
Microsoft Research

Abstract

Standalone orchestrators simplify the development of serverless applications by providing higher-level programming interfaces, coordinating function interactions and ensuring exactly-once execution. However, they limit application flexibility and are expensive to use. We show that these specialized orchestration services are unnecessary. Instead, application-level orchestration, deployed as a library, can support the same programming interfaces, complex interactions and execution guarantees, utilizing only basic serverless components that are already universally supported and billed at a fine-grained per-use basis. Furthermore, application-level orchestration affords applications more flexibility and reduces costs for both providers and users.

To demonstrate this, we present Unum, an application-level serverless orchestration system. Unum introduces an intermediate representation that partitions higher-level application definitions at compile-time and provides orchestration as a runtime library that executes in-situ with user-defined FaaS functions. On unmodified serverless infrastructures, Unum functions coordinate and ensure correctness in a decentralized manner by leveraging strongly consistent data stores.

Compared with AWS Step Functions, a state-of-the-art standalone orchestrator, our evaluation shows that Unum performs well, costs significantly less and grants applications greater flexibility to employ application-specific patterns and optimizations. For a representative set of applications, Unum runs as much as 2x faster and costs 9x cheaper.

1 Introduction

Serverless computing offers a simple but powerful abstraction with two essential components: a stateless compute engine (Functions as a Service, or FaaS) and a scalable, multi-tenant data store [27]. Developers build applications using stateless, event-driven "functions" which persist states in shared data stores. This abstraction allows users to leverage scalable data center resources with fine-grained per-invocation billing and frees them from server administration.

While serverless platforms originally targeted simple applications with one or a few functions, this paradigm has increasingly proven useful for more complex applications composed of many functions with rich and often stateful interaction patterns [19, 20, 25, 26, 40]. Unfortunately, building such applications using the basic FaaS is challenging. Event-driven execution makes depending on the results of multiple previous functions and therefore fan-in patterns difficult. At-least-once execution guarantee that is typical for FaaS functions complicates end-to-end application correctness as non-deterministic functions may pass inconsistent results downstream. Finally, the lack of higher-level programming interfaces for expressing inter-function patterns hinders application development.

Standalone orchestrators are recently introduced into the serverless infrastructure to support such complex applications (§2.1). Cloud providers commonly offer serverless orchestrators as a service [3, 6, 22, 23], though users may build custom orchestrators and deploy them in separate VMs or containers alongside their functions [19, 20, 38]. These orchestration services provide higher-level programming interfaces, support complex interactions and ensure exactly-once execution.

Though often internally distributed, standalone orchestrators operate as *logically centralized* controllers. Developers provide a description of an execution graph—nodes in the graph represent FaaS functions and edges represent invocations of a function with the output of one or more functions—and the orchestrator drives the execution of this graph by invoking functions, receiving function results and storing application states (e.g., outstanding invocations and function results) centrally.

Centralization simplifies supporting stateful interactions—e.g., an orchestrator can run fan-in patterns by simply waiting for all branches to complete before invoking an aggregation function. Similarly, an orchestrator can ensure that applications appear to execute exactly-once by choosing a single result from multiple executions for each function invocation.

However, standalone orchestrators have important drawbacks for both serverless providers and serverless users. As an additional service that is critical to application performance

and correctness, a standalone orchestrator is expensive to host and use. User-deployed orchestrators risk under-utilization and do not benefit from serverless' per-use billing. Provider-hosted orchestrators are multi-tenant and can thus multiplex over many users to improve resource utilization and amortize the cost. However, they still incur the costs of hardware resources and on-call engineering teams. These costs may be affordable for large platforms but can be a significant burden for smaller providers.

Furthermore, standalone orchestrators preclude users from making application-specific trade-offs and optimizations. While the interface and implementation of an orchestrator might efficiently support the needs of many applications, it cannot meet all applications' needs, resulting in a compromise familiar from operating systems [9,13], networks [17,39], and storage systems [21,28].

For example, applications that need orchestration patterns not supported by the provider-hosted orchestrator have to either compromise performance by using less-efficient patterns or first repeat the hard work of building, deploying and managing their own custom orchestrator. A video processing application that encodes video chunks and aggregates results of adjacent branches in parallel has to compromise performance if the orchestrator only supports aggregating results of all branches.

Similarly, applications that consist entirely of deterministic functions, such as an image resize application for creating thumbnails or an IoT data processing pipeline for aggregating sensor readings, can tolerate duplicate executions without weakening correctness. However, with a standalone orchestrator that always persists states to ensure exactly-once execution, this application would incur the overheads of strong guarantees regardless.

In this paper, we show that additional standalone orchestrators for serverless applications are unnecessary. Furthermore, we argue that application-level orchestration is better for both serverless providers and developers. It is better for developers as it affords applications more flexibility to implement custom patterns as needed and apply application-specific optimizations. It is better for providers as it obviates the need to host an additional complex service and frees up resources such that providers can focus on fewer, core services in their serverless infrastructure. Moreover, application-level orchestration built on top of existing storage and FaaS services in the serverless infrastructure can benefit automatically from improvements to cost and performance to these services.

To support these arguments, we present Unum, an application-level serverless orchestration system. Unum provides orchestration as a library that runs *in-situ* with user-defined FaaS functions, rather than as a standalone service. The library relies on a minimal set of existing serverless APIs—function invocation and a few basic data store operations—that are common across cloud platforms. Unum introduces an intermediate representation (IR) language to express execution graphs using only node-local information and supports front-end compilers that transform higher-level application definitions into the IR.

A key challenge in Unum is to support complex stateful orchestration patterns and strong execution guarantees in a *decentralized* manner. Our insight is that, scalable and strongly consistent data stores, already an essential building block of serverless applications, address the hardest challenge of orchestration: coordination. Using such data stores, we show that an application-level library running in-situ with user functions can orchestrate complex execution graphs efficiently with strong execution guarantees.

At a high level, Unum relies on the FaaS scheduler to run each function invocation *at least* once and consistent data store operations to coordinate interactions and de-duplicate extra executions of the same invocation. Unum uses checkpoints to commit to exactly one result for a function invocation and ensures workflow correctness despite duplicate executions of non-deterministic functions. Unum fan-ins use objects in a consistent data store as coordination points for aggregating branches. Both require generating globally unique names for nodes and edges in the execution graph *locally* (using only information available at each node) as well as cleaning up intermediate data store objects in a timely manner.

Our implementation of Unum (§4) includes a compiler for AWS Step Functions' description language, enabling Unum to run arbitrary Step Function workflows. We show that Step Function workflows compiled to Unum execute with the same execution guarantees as running natively using the Step Functions orchestrator.

Moreover, while performance and cost are difficult to compare objectively with existing black-box production orchestrators—both are influenced by deployment and pricing decisions that may not reflect the underlying efficiency or cost of the system—Unum performs well in practice (§5). We find that a representative set of applications run faster and cost significantly less with Unum than Step Functions (Table 2). We also demonstrate that Unum's IR allows applications to run faster by using application-specific optimizations and supporting a richer set of interaction patterns.

2 Background & Motivation

The basic serverless abstraction is simple and quite powerful. Developers build "functions", typically written in a high-level language and packaged as OS containers or virtual machines, which run short computations in response to a platform event. Events include storage events (e.g., the creation of an object) or HTTP requests. The platform can scale resources for each function to respond to instantaneous bursts in events and developers are absolved from capacity planning and resource management tasks.

This simple abstraction can be used to compose many simple applications with one or a few functions. For example, developers can chain functions for data pipelines using triggers. In trigger-based composition [10] each function in a chain invokes the next asynchronously or writes to a data store configured to invoke the next function in response to storage events. Alternatively, developers might use a "driver-function" [40] to drive more intricate control-flow logic. A driver function acts as a centralized controller that invokes other functions, waits for their outputs, and invokes subsequent functions with their outputs.

Such ad-hoc approaches work "out-of-the-box", that is, they require no additional platform provided infrastructure. However neither is well suited to complex applications with 10s or 100s of functions [20, 35]. Trigger-based composition can only support chaining of individual functions or fan-out from one function to multiple, but cannot, for example, fan-in from multiple functions to one. Moreover, trigger-based composition scatters control-flow logic across each function or in configured storage events, making development unwieldy when application complexity grows.

On the other hand, driver functions concentrate control flow in a single function and support arbitrary composition. However, most serverless platforms impose modest runtime limits on individual functions, and thus driver functions restrict the total runtime of applications. Furthermore, driver functions suffer "double billing" since they are billed for the entire call-graph execution despite spending most time idly waiting for callees to return.

Finally, both ad-hoc approaches require developers to handle function crashes, retries and duplicate invocations gracefully [1,7,8,14]. Application typically want to ensure "exactly once" semantics [10, 11, 24, 25, 40] for an entire call-graph, but failures and multiple invocations of individual functions can subvert this goal without careful consideration.

2.1 Standalone Orchestrators

A common solution to address the needs of complex serverless applications is to introduce a workflow orchestrator that provides a high-level programming interface with support for a rich set of patterns (e.g., branching, chaining, fan-out and fan-in) [3, 6, 19, 20, 22, 23, 38]. Many cloud providers offer serverless orchestrators as a service [3, 6, 22, 23] or users can build custom orchestrators [19, 20, 38] and deploy them in VMs alongside their functions.

Similar to driver functions, orchestrators operate as *logically centralized* controllers. They drive a workflow by invoking its functions and hosting application states such as function outputs and outstanding invocations.

However, different from driver-functions, orchestrators are standalone services. Orchestrators are not limited by function timeouts and can be arbitrarily long-running [4]. Moreover, as standalone services, orchestrators are often internally distributed and employ techniques such as replication and sharding to provide strong execution guarantees, fault-tolerance and scalability. For example, orchestrators can ensure that workflows appear to execute exactly-once by choosing one result for each function invocation, even if FaaS engines only guarantee at-least once execution. Orchestrators can also persist or replicate states during execution so that in face of orchestrator failures, applications do not lose executions or retry from the beginning.

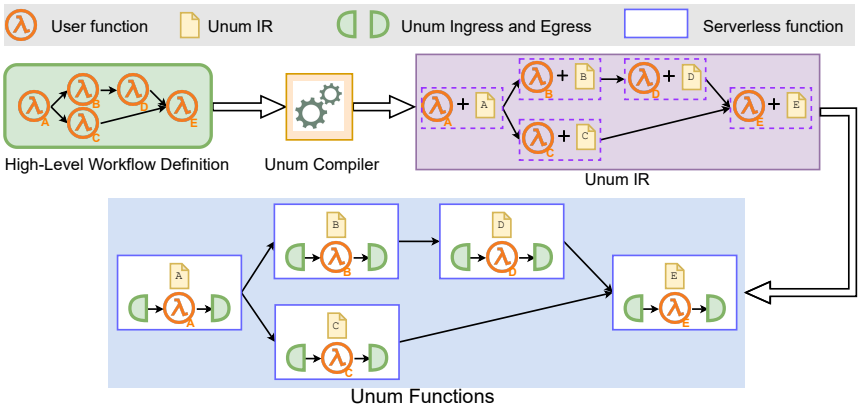
While orchestrators are able to address the needs of complex serverless applications, introducing a new standalone service has significant drawbacks. Building performant, scalable and fault-tolerant multi-tenant systems is hard and orchestrators introduce yet-another potential performance and scalability bottleneck. Indeed, we find that, in practice, production systems limit end-to-end performance for highly-parallel applications (§ 5).

Moreover, hosting such services is expensive. Deploying a custom orchestrator per user risks under-utilization as it cannot multiplex over many users and users pay even when the orchestrator is not actively in use, breaking the fine-grained billing benefit of serverless. Provider-hosted orchestrators are multi-tenant and can amortize this cost. But they still incur engineering expenses as they require teams on-call. Indeed, we find that provider-hosted orchestrators cost developers significantly and dominate the total cost of running applications(§ 5).

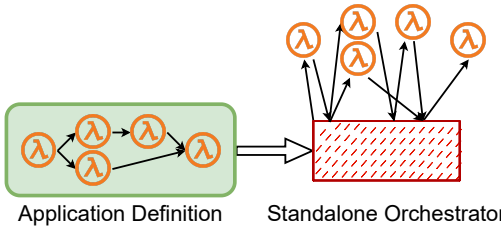
Lastly, provider-hosted orchestrators preclude users from making application-specific optimizations. Each provider typically offers just a single orchestrator service option. While the interface and implementation of the orchestrator might efficiently support many applications, it cannot meet all applications' needs, resulting in a compromise familiar from operating systems [9, 13], networks [17, 39], and storage systems [21, 28]. Indeed, we find that provider-hosted orchestrators force applications to compromise performance by using less-efficient patterns (§ 5).

3 Design

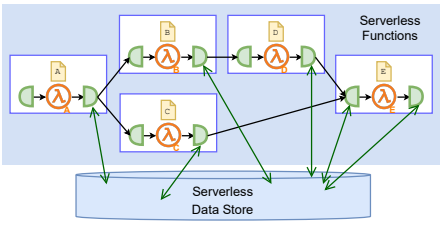
Unum is an application-level orchestration system that supports complex serverless applications without a standalone orchestrator. It does so by decentralizing orchestration logic in a library that runs in-situ with user-defined FaaS functions and leverages a scalable consistent data store for coordination and execution correctness. By removing standalone orchestrators, Unum improves application flexibility and reduces costs. Importantly, Unum does this while retaining the expressiveness and execution guarantees (§3.3) of standalone orchestrators.



(a) Serverless workflows form directed graphs. Unum partitions the graph into an intermediate representation where each function is embedded with an Unum configuration that encodes how to transition to its immediate downstream nodes. Developers package user function, Unum config and Unum's runtime library (a pair of ingress and egress components) together to create "unumized" functions.



(b) A typical standalone orchestrator operates as a logically centralized controller that drives the execution of applications by invoking functions, receiving function results and storing application states



(c) At runtime, Unum orchestration logic is decentralized and runs in-situ with the user functions on an unmodified serverless platform. For coordination and checkpointing, Unum relies exclusively on a standard data store of choice, such as DynamoDB or Cosmos DB.

Figure 1: Unum's Decentralized Orchestration. Unum partitions orchestration logic at compile time and a Unum runtime runs in-situ with user functions to perform only the orchestration logic local to its node.

3.1 Architecture

Figure 1a depicts how developers run serverless workflows using Unum. Developers write individual functions and describe the workflow using a high-level workflow language, such as Step Functions' expression language. An Unum front-end compiler uses these to extract portable Unum IR for each node in the graph and "attaches" it to the function (e.g. by placing a file containing the IR alongside the function code). A platform-specific Unum linker "links" each function with a platform-specific Unum runtime library. [1] Developers deploy each linked function along with its IR to the FaaS platform.

Each Unum workflow begins with an "entry" function. Invoking this function (e.g. using an HTTP or storage trigger) starts a workflow. Moreover, admission control rules for the workflow, such as access control and rate limiting, are implemented by setting appropriate rules on this entry function. For example, a workflow can be invoked by a particular principal if the entry function is exposed to that principal.

The runtime library is composed of an ingress and egress component that run before and after the user-defined function and unwrap and wrap the results of the function in Unum execution state, respectively (Figure 2). The ingress component coalesces input data from each incoming edge (e.g. in a fan-in), resolves input data if passed by name rather than by value, and passes the input value to the function. The egress component uses the function's result to invoke the next function(s), enforces execution semantics using checkpoints, performs coordination with sibling branches in fan-in, and deletes intermediate states no longer needed for the workflow, executing the workflow in-situ with the functions, in lieu of a centralized orchestrator (Figure 1c).

3.2 Unum Intermediate Representation

Similar to many standalone orchestrators, Unum applications are modeled as directed execution graphs where nodes represent user-defined FaaS functions and edges represent function invocations (incoming edges) with the output of one

[1] Since functions are typically written in dynamic languages, the Unum library source code is placed alongside the function and dynamically imported, rather than statically linking an object file

`Invoke(Fn)`	Queue a single invocation of a function.
`Map(Fn)`	Queue one function invocation for each element in the current function's result.
`FanIn(Set, Size, Fn)`	Queue a fan-in to a function using the provided coordination set object and size.
`Pop`	Pops the top frame of the execution state stack (passed via Unum requests).
`Next`	Increments the current execution state frame's iteration counter.
`CreateSet(Name)`	Creates a coordination set object in the data store with the provided name.

Table 1: Unum intermediate representation instructions.

or more other functions (outgoing edges).

An Unum graph may include fan-outs, where a node's output is used to invoke several functions or split up and "mapped" multiple times on the same function. Each such branch may be taken conditionally, based on the output value or dynamic states of the graph. Execution graphs may also contain fan-ins, where the outputs of multiple nodes are used to invoke a single aggregate function. Cycles are also supported and each iteration through a cycle is a different invocation of the target function.

The Unum intermediate representation (IR) is designed to encode directed execution graphs in a way that both allows decentralization of orchestration and is low-level enough to support application-specific patterns.

Each function's IR includes the function's name and a sequence of instructions (Table 1). Instructions direct the runtime to invoke functions and operate on Unum metadata passed between functions (Figure 2).

The egress component, which receives the function's user-code output, executes the IR and uses it to determine which next steps to take. An invocation can be protected by a conditional—a boolean expression that operates on the invocation request and the current function's output. Unum's IR provides three kinds of invocations:

- **Invoke** simply invokes the named function using the current functions output.
- **Map** treats the current function's output as iterable data (e.g. a list) and invokes the named function once for each item in the output.
- **FanIn** invokes the named function using the current function's output along with the outputs of all other functions fanning into the same node. Fan-in requires coordination among multiple functions and is described in detail in §3.4.

```
struct InvocationRequest {
  data: Vec<DatastoreObjectName>,
  workflowId: String,
  fanOut: Stack<FanOut>,
}

struct RequestData {
  reference: DatastoreObjectName,
  value: Option<Value>
}

struct FanOut {
  index: usize,
  size: usize,
  iteration: usize,
}
```

Figure 2: An Unum request wraps function outputs with metadata that allows function invocations to be named uniquely and assists in coordinating fan-ins. Unum IR instructions can reference and modify this metadata.

When multiple invocations occur, either using multiple instructions or a single `Map` invocation, each of the invocations adds a fan-out frame to the invocation request's fan-out stack. This allows different invocations of the same function to be differentiated for naming (§3.6) and to coordinate fan-in (§3.4).

The IR also includes instructions for manipulating the Unum request data and an instruction that creates a new coordination set, typically for use in later nodes to coordinate fan-in (§3.4) or garbage collection (§3.5).

This IR is sufficient to represent basic patterns, as well as more complex fan-in patterns (described in §3.4).

Chain & Fan-out. Unum encodes passing the output of a function to one (chain) or multiple (fan-out) subsequent functions, simply, with one or more calls to the `Invoke` instructions.

Map. Applications may perform the same operation on each component of a function's output. For example, an application may unpack an archive of high-resolution images in one function and perform compression on each of the images. Unum's `Map` instruction invokes the same `Fn` for each element of a function's output.

Branching. Applications may need to invoke different functions based on runtime conditions (e.g., the output of a function). For instance, an application may first validate that a user-uploaded photo is a valid JPEG. If it is, it invokes, e.g., one of the patterns above, otherwise it notifies the user of the error. Unum's invocation instructions are optionally protected by a conditional expression that has access to the function output and execution metadata (Figure 2).

3.3 Execution Guarantees Using Checkpoints

FaaS functions only provide weak execution guarantees. Functions can fail mid-execution and be retried. Even in the absence of failures, one function invocation may result in more than one execution because most FaaS engines only ensure at-least-once execution. This is problematic for applications whose functions are non-deterministic because a single workflow invocation can produce multiple *diverging* outputs.

An important benefit of orchestrators is strong execution semantics such that applications appear to execute *exactly-once* even if individual functions in the application run multiple times. Because standalone orchestrators are logically centralized, guaranteeing exactly-once is conceptually straightforward: the orchestrator can choose a single result from executions of the same invocation and use it as input for all downstream functions. At the end of the workflow, the result is consistent with an execution of the workflow where each function invocation executed exactly-once.

A key challenge for Unum is to provide the same semantics without centralizing orchestration. Moreover, because failures and, thus, retries are the exception, not the rule, Unum should provide these semantics without expensive coordination—function instances should be able to proceed without blocking in the common, fault-free case.

Unum leverages two key insights to achieve these semantics. First, it is correct for different executions of the same function invocation to return different results as long as Unum ensures downstream functions are always invoked with exactly one of those results. Second, a workflow's output is *correct* even if a function is invoked more than once, as long as the invocations uses the same input, since additional, but identical, invocations are indistinguishable from additional executions.

The Unum library employs an atomic `create_if_not_exists` operation in the serverless data store to *checkpoint* exactly one execution of each function invocation. The egress component of the Unum library attempts to write the result of the function to a checkpoint object in the data store. If such a checkpoint already exists, a concurrent or previous execution of the invocation must have already completed and the operation will fail. To invoke downstream functions, the egress component *always* uses the value stored in the checkpoint, rather than the result of the recently completed function. Essentially, Unum "commits" to result of the first successful executions of invocations.

Data stores need to be strongly consistent to support `create_if_not_exists`. It is important that a later attempt to create an existing checkpoint fails and the slower execution can read the existing checkpoint.

As a further optimization, the ingress component in the Unum library checks for the checkpoint object before executing the user-defined function. If the object exists, it bypasses the user-defined function and passes the checkpoint value directly to the egress component to invoke downstream functions. This is not necessary for correctness but helps reduce computation that we know will go unused.

```
def ingress(self, function):
    ...
    result = datastore_get(self.checkpoint_name):
    if result:
        self._egress(result)
    else:
        self.egress(function.handle())

def egress(self, result):
    ...
    if not datastore_atomic_add(self.checkpoint_name, result):
      result = datastore_get(self.checkpoint_name)
    self._egress(result)
    ...

def _egress(self, result)
    for f in next_functions:
        faas.async_invoke(f, result)
```

Figure 3: Pseudo-code showing Unum's checkpointing mechanism. As different executions of a function may return different results, Unum's egress component checkpoints the first successful execution using an atomic add data store operation. All subsequent executions will uses this committed value rather than the result their own execution returned.

Note that the exactly-once guarantee does not automatically extend to applications with external side effects, i.e. functions that directly call external services. In such cases, retries can lead to unexpected results if the effects are not idempotent. This issue is well known, and independent of the orchestrator architecture (centralized vs. decentralized). Thus, we consider the question of how to control such side effects to be orthogonal and beyond the scope of this paper. However, Unum does not preclude applications from using libraries, such as Beldi [40], that can solve this problem.

3.3.1 Fault Tolerance

Another source of multiple executions is retrying failed functions. Retries in Unum rely on FaaS engines' error handling support. All popular FaaS engines provide error handling so that applications do not just crash silently without a way to react to failures. Common mechanisms include "automatic retry" that re-executes the same function [7, 14, 32, 34] or failure redirection that triggers a pre-configured error-handler function [29, 33]. Unum can work with either mechanism.

The Unum error handler is part of Unum's standard library and is triggered in a separate FaaS function after an application function crashes. The error handler simply retries the crashed function by invoking it again. As part of the orchestration library, the error handler is assumed to be bug-free and relies on the FaaS scheduler to execute at least once.

Unum's checkpointing mechanism ensures that while faults may occur at any point during the execution of a function's

user code or the Unum library, and while downstream functions may be invoked multiple times by different executions of the same invocation, a single value is always used to invoke downstream functions.

If there is a fault after the user code completes but before creating the checkpoint, user code result is ignored (indeed, never seen) by other executions and another execution's value will be used to invoke downstream functions. If the "winning" function crashes after creating a checkpoint, and before invoking some or all downstream functions, other executions will use the checkpoint value to invoke downstream functions. Finally, even if multiple executions invoke some or all downstream functions, execution guarantees are still satisfied as these invocations will have identical inputs.

3.4 Fan-in Patterns

In fan-in patterns, the results of multiple nodes are used to invoke a single head node. Such patterns are a particular challenge for decentralized orchestration because invoking the target function cannot happen until all branches complete, but there is no standalone orchestrator to wait for this condition. Designating one of the tail nodes as the coordinator would address this directly. However, there is no guarantee that branches for a fan-in complete soon after each other, incurring a potentially large resource cost to do virtually no work, and risk exceeding platform-enforced function timeouts. Moreover, functions typically cannot communicate with each other directly, so it is not obvious how other branches would notify this coordinator of their completion.

Unum, instead, leverages the same insight as checkpoints—the data store provides strong consistency that can serve as a coordination point. Rather than designating a single branch function as the coordinator, all branches are empowered to invoke the fan-in function once all other branches have completed. To determine this condition, branches in a fan-in add the name of their checkpoint object to a shared "Set" in the data store. Any branch that reads the set with size equal to the total number of branches invokes the target function using all the branches' checkpoints as input.

Importantly, functions do not wait for any other to complete. As long as all functions complete eventually (in other words, they run at-least once), *some* function will read a full set and invoke the fan-in target function. More than one function may observe this condition, resulting in multiple invocations, but these invocations will be identical and are handled as spurious executions of the same invocation (§3.3).

In order to perform this coordination, branches must know the branching factor—the size of the set. The `FanIn` instruction includes this size, which is either specified explicitly, or using a variable from the invocation request, commonly the fan-out size.

Similar to checkpoints, the set data structure for coordination requires the data store to be strongly consistent. Updates to a set must be immediately visible to other branches otherwise the downstream fan-in function may ever be invoked. Moreover, the data store must support data structures that can implement a "set" abstraction.

Fan-in supports enable more patterns that commonly arise in applications:

Aggregation. After processing data with many parallel branches, applications commonly want to aggregate results. For example, to build an index of a large corpus, the application might process chunks in parallel and then aggregate the results. Aggregation is a common pattern to join back multiple parallel functions, by invoking a single "sink" function with the outputs from a vector of functions.

Fold. `fold` sequentially applies the same function on the outputs of a vector of source functions, while aggregating with the intermediate results of running the function so far. For example, a video encoding application might encode chunks in parallel and then concatenate the results in order: concatenating chunk 1 and 2, then concatenating chunk 3 to chunk [1–2], and so on. `fold` is an advanced pattern that is not supported by all existing systems (e.g., AWS Step Functions do not support `fold`) but is expressible in Unum.

3.5 Garbage Collection

Both checkpointing and fan-in require storing intermediate data (e.g., checkpoints and coordination sets) in the data store. These intermediate data is only temporally useful and grows with each invocation. This poses a garbage collection challenge. Deleting them too early can compromise execution guarantees while deleting too late incurs storage costs.

Checkpoint Collection. A checkpointing node does not know when *its* checkpoint is no longer necessary. If it deletes its checkpoint immediately after invoking subsequent functions, it may crash and the FaaS platform may re-execute it, yielding a potentially inconsistent result. However, downstream nodes know that once they have committed to a value by checkpointing, previous checkpoints are no longer necessary to ensure their own correctness. Once a node has committed to some particular output, future invocations, even with *different* inputs will produce the same output, as the node will *always* use the checkpointed value.

Note that a duplicate execution that checkpoints after the previous checkpoint is garbage collected has the same semantics as a separate invocation. It may result in multiple outputs from the workflow, though each output is still consistent with an execution of the workflow where each function was invoked exactly-once. Any GC policy, no matter how conservative, might lead to multiple executions if the FaaS platform could execute duplicates of a function invocation after an arbitrarily long time in the future.

Therefore, Unum collects checkpoints by relaxing the constraint that nodes always output the same value. Instead, they must only output the same value until all subsequent nodes

have committed to their own outputs. This means that, in non-fan-out cases, once a node checkpoints its result, it can delete the previous node's checkpoint.

Fan-outs are more complicated because deleting the checkpoint must wait until all branches have committed to an output. Unum repurposes the same set-based technique from fan-in to collect checkpoints in fan-out cases as well. The originating node of a fan-out creates a set for branches to coordinate when to delete its checkpoint. Branches add themselves to the set after checkpointing their own value. Any node that reads a full set deletes the parent's checkpoint as well as the set. This guarantees that the parent's checkpoint is deleted and ensures that all branches have first checkpointed.

Note that it is possible for one of the branches to re-execute *after* the set has been deleted. This is safe because it is the origin of the fan-out that creates the set, so a branch's attempt to add itself to a, now, non-existent set will simply fail.

Fan-in Set collection. Deleting sets used for fan-in works much like removing checkpoints—the target node of a fan-in deletes the set once it has generated a checkpoint. However, who *creates* the set?

If each branch in the fan-in creates the set if it doesn't already exist, a spurious execution of one of the branches *after* the fan-in target removes the original set will create a new one that is never deleted (because it never fills, and thus the target function is never invoked again). To avoid this, Unum places the responsibility to create the set on the node that originates the *fan-out* at the same level as the target node.

3.6 Naming

Much of Unum's functionality relies on unique naming. A workflow invocation must be named to differentiate it from other concurrent invocations of the workflow; functions must be named to invoke them; different invocations of functions must have different names to uniquely name invocation checkpoints and coordination sets for fan-in.

Each workflow invocation has a unique name that is passed through the execution graph. The name is either generated in the ingress to the first function using, e.g., a UUID library or, when available, is taken from the FaaS platform's invocation identifier for the first function. This enables functions to have different names when invoked as part of a specific workflow invocations. The function's name is either user-defined or determined by the FaaS platform (e.g. the ARN on AWS Lambda) and determined at "compile-time" (i.e. when generating Unum IR).

However, this is not sufficient as functions may be invoked multiple times in the same workflow due to map patterns—which invoke the same function multiple times over an iterable output—and cycles. Moreover, invocation names must be determined using local information only. Once running, each function only has access to it's own code (including the IR) and metadata passed in its input. Nonetheless a particular invocation must be able to determine its own name for checkpointing as well as, if it is part of a fan-in, the name of downstream invocations to coordinate with other branches.

As a result, Unum names function invocations using a combination of the global function name, a vector of branch indexes and iteration numbers (taken from the Unum request fan-out stack) leading to the invocation, and the workflow invocation name. Function names are global and the remaining items are propagated by Unum in invocation arguments.

During a fan-out pattern (multiple scalar invocations or a map invocation), a branch index is added to a list in the next functions' input. If the next function is an ancestor of the current function (a cycle), an iteration field in the input is incremented. Note that a single iteration field is sufficient even if there are nested cycles since it is only important that different invocations of the same function have *different* names, not that the iteration field is sequential. Thus, a monotonically increasing iteration field is sufficient.

We note that the format of this name is not significant and, importantly, it need not be interpretable. It must only be deterministic and unique for its inputs. For example, a reasonable implementation could serialize the inputs and take a cryptographic hash over the result, guaranteeing uniqueness (with very very high probably) while preventing names from growing too large to use as object names.

4 Implementation

We implement a prototype Unum runtime that supports AWS and Google Cloud. We also implement a front-end compiler that transforms AWS Step Function definitions to Unum IR. Currently our runtime only supports Python functions and is itself written in 1,119 lines of code. The Step Functions compiler is 549 lines-of-code.

Implementing the runtime primarily requires specializing high-level functionality the IR depends on for a particular FaaS platform and data store. The FaaS platform must support asynchronous invocation and the data store must be strongly consistent with support for atomic creation and set operations.

Importantly, we choose data stores and primitives that only incur per-use costs and scale on-demand. For example, we use DynamoDB in on-demand capacity mode, rather than provisioned capacity mode, and avoid long-running services such as a hosted Redis or cache. As a result, Unum incurs fine-grained costs only when performing orchestration (e.g., per-millisecond Lambda runtime costs to execute the Unum library, per-write DynamoDB costs to create checkpoints).

4.1 AWS Lambda & DynamoDB

Asynchronous invocation in Lambda is natively supported. In particular, the Lambda `Invoke` API is asynchronous when passed `InvocationType=Event`. In the event of a crash, we use Lambda's Failure Destination [29] to redirect the fault to

an error handler function which runs just the Unum runtime. The error handler checks if the failed function should be retried (e.g., based on the Step Function definition [15]) and if so, retries the function by explicitly invoking it again.

DynamoDB organizes data into tables, with each item in a table named by a key. Within tables, items are unstructured by default. Our implementation of Unum uses a single table for each workflow. Each item in the table corresponds to a checkpoint, or a coordination set for fan-in or garbage collection.

DynamoDB supports atomic item creation by passing the conditional flag `attribute_not_exists` to the `put_item` API call. We use this for creating both checkpoint blobs and coordination sets. DynamoDB supports set addition natively using the `Map` field type. In particular, we use update expressions to atomically set a named map element to true. As an optimization, we use the `ALL_NEW` flag when adding to a set to atomically get the new value after a set in a single operation.

4.2 Google Cloud Functions & Firestore

Google Cloud Functions (GCF) do not have an asynchronous invocation API. Instead, we allocate function-specific pub-sub queues and subscribe each function to its respective queue. Unum then asynchronously invoke a function by publishing the input data as an event to the function's queue.

GCF supports automatic retry for asynchronous functions [34]. In the event of a crash, the Unum runtime in the retry execution checks if the failed function should be retried and if so, retries the function by explicitly invoking it again.

Firestore organizes data into logical collections (which are created and deleted implicitly) containing unstructured items, named by a unique key. Similar to DynamoDB, we use a separate collection for each workflow. Atomic item creation is supported using a special `create` API call, which only succeeds if the key does not already exist. Firestore supports an `Array` field type which can act as a set by using the `ArrayUnion` and `update` operation, which atomically sets the field to the union of its existing elements and the provided elements. The `update` operation always returns the new value data.

5 Evaluation

Unum argues for eschewing standalone orchestrators and, instead, building application-level orchestration on unmodified serverless infrastructure using FaaS schedulers and consistent data stores. In this section, we evaluate how well application-level orchestration performs, reduces costs, and improves application flexibility. In particular, we focus our performance evaluation on whether decentralization comes at a reasonable overhead compared with standalone orchestrators.

Specifically, we answer the following questions:

1. What overhead does Unum incur in end-to-end latency and what are the sources of Unum's overheads?

2. How much does it cost to run applications with Unum and what are the sources of costs compared with Step Functions?

3. How well does Unum support applications that Step Functions cannot support well?

Though we evaluate the applications running on both AWS and Google Cloud, we focus our discussion on our AWS implementation with Lambda and DynamoDB, because it runs on the same serverless infrastructure as Step Functions.

5.1 Experimental setup

We run all experiments on AWS with Lambda and DynamoDB, and on Google Cloud with Cloud Functions and Firestore. All services are in the same region (`us-west-1` on AWS and `us-central-1` on Google Cloud). All functions are configured with 128MB of memory except for ExCamera where we use 3GB of memory to replicate the setup in the original paper [19, 20]. DynamoDB uses the on-demand provisioning option that charges per-read and per-write [12]. To avoid performance artifacts related to cold-starts, we ensure functions are warm by running each experiment several times before taking measurements.

All but one application were originally written as Step Function state machines. For Step Function experiments, we ran them directly with the "Standard" configuration [37], which provides similarly strong execution guarantees as Unum [16]. For Unum experiments, we first compiled the Step Functions definitions to Unum IR, linked the functions with the Unum runtime library and finally executed them as lambdas or Google Cloud functions. The notable exception is our Unum and Step Functions implementations of ExCamera, which differ due to a limitation in the Amazon State Language. As a result, the more efficient Unum implementation is written directly in Unum IR instead of compiled from the Step Functions definition (§ 5.4).

5.2 Performance

Unum's performance overhead results from the Unum runtime logic run in each function as well as API calls to data stores and FaaS engines. We characterize these overheads by measuring the latency to execute various patterns consists of `noop` functions as well as end-to-end performance of real applications. Overall we find that Unum performs comparably or significantly better than Step Functions in most cases owing to higher parallelism and a more expressive orchestration language, with modest slow downs in the remaining cases due to implementation deficiencies.

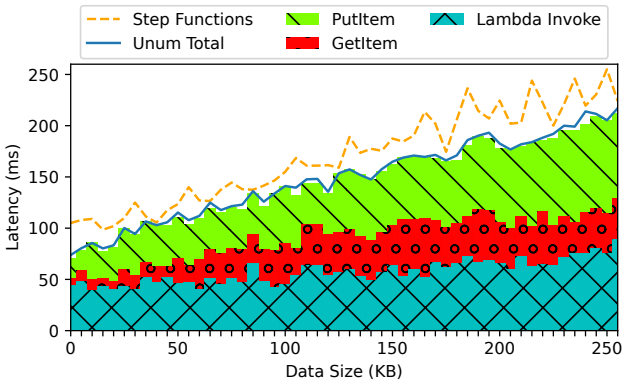


Figure 4: An orchestrator incurs a latency on each transition between functions. Unum's overhead is due to storage operations to ensure exactly-one-result semantics, Lambda invocation API overhead to enqueue the next function to run, and additional Unum runtime code in the function instance itself for the orchestration logic.

5.2.1 Chaining

For the simple chaining pattern, the Unum runtime performs a storage read to check whether a checkpoint already exists, a storage write to checkpoint the function's result, and an asynchronous function invocation to initiate the next function in the chain.

Figure 4 shows time to perform each of these operations for different result sizes. As expected, storage operations are slower when checkpointed results are larger, but the total overhead from the Unum runtime operations is consistently lower than an equivalent Step Function transition.

The Unum implementation of the IoT pipeline application benefits from this difference, with the Unum version running 1.9x faster than the Step Functions version (Table 2).

5.2.2 Fan-out and fan-in

Fan-out requires the same number of storage operations as chaining and similar orchestration logic, but the Unum runtime performs an additional asynchronous invocation at the source function for each branch. For fan-in patterns, each source branch performs an additional storage read to determine if it is the final branch to execute, and only the final branch performs the asynchronous invocation of the target function.

Figure 5 shows the latency of a fan-out followed by a fan-in at varying branching degrees for both Unum and Step Functions. At low branching degree, Unum incurs a modest overhead (up to 200ms) relative to Step Functions. We believe this is mostly due to our implementation initiating each branch invocation sequentially. However, at higher branching degrees (as low as 20 branches), Step Functions limits the number of outstanding fan-out branches [30] while Unum is limited only

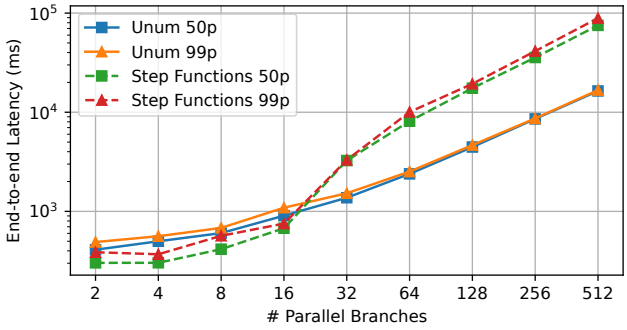


Figure 5: End-to-end latency of a fan-out and fan-in pattern with increasing branching degree. Unum is slower at lower branching degrees but significantly outperform Step Functions at moderate and high branching degree.

by Lambda's scalability, resulting in over 4x lower latency with 512 branches.

These differences manifest in real workloads as well. Wordcount is highly parallel (with 262 parallel mappers and 250 reducers) and performs over 2x faster on Unum than on Step Functions (Table 2).

Although it may not be the case that standalone orchestrators fundamentally have to impose limits on the number of outstanding function invocations, this example shows that it is at least not trivial to ameliorate the constraint. On the other hand, as a library, Unum is free from the need to design and implement yet-another service that supports parallel applications well, but can instead provide as much parallelism as FaaS schedulers and data stores permit. FaaS schedulers and data stores already support highly-parallel applications well, and Unum's performance will improve automatically when these underlying services further improve.

5.3 Cost

One of the main attractions of building applications on serverless platforms is fine-grained and often lower cost. In particular, because resources are easy to reclaim, applications are charged only for resources used to respond to actual events. Thus, the *cost* of orchestration matters as well as performance.

The source of costs for Unum and Step Functions is quite different. Step Functions imposes a cost to developers for each workflow transition [5], such as each branch in a fan-out. This abstracts the underlying, likely shared, costs to run the Step Functions servers, persist states and checkpoint data. Conversely, Unum incurs costs directly from those services. In particular, compute resources for executing orchestration logic is charged per-millisecond such as Lambda runtime cost [2] and storage for persisting states is charged per read and write such as DynamoDB reads and writes [12].

On AWS, Unum is much cheaper than Step Functions—AWS's native orchestrator. For a basic transition in a chaining

	Latency (seconds)			Costs ($ per 1 mil. executions)		
App	*Unum-aws*	*Unum-gcloud*	*Step Functions*	*Unum-aws*	*Unum-gcloud*	*Step Functions*
IoT Pipeline	0.12	0.81	0.23	$12.38	$6.3	$112.02
Text Processing	0.52	3.56	0.55	$60.42	$31.7	$225.29
Wordcount	408.88	484.12	898.56	$13,433.67	$11,727.3	$18,141.19
ExCamera	84.52	122.63	98.42	$62,684.29	$51,617.2	$114,633.13

Table 2: Application latency and costs comparison between Unum and Step Functions. Running applications on Unum is 1.35x to 9x cheaper than on Step Functions. Furthermore, Unum is faster than Step Functions especially for workflows with high degrees of parallelism.

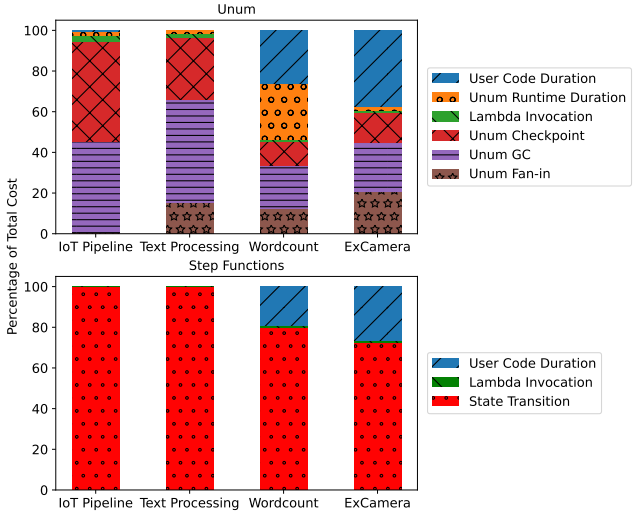


Figure 6: Step Functions state transitions dominate the total costs for all applications (99.5% in IoT Pipeline, 99.4% in Text Processing, 80.0% in Wordcount, 72.2% in ExCamera). While Unum runtime cost is also the majority, it accounts for a smaller portion of the overall costs (95.7% in IoT Pipeline, 97.8% in Text Processing, 72.5% in Wordcount and 61.0% in ExCamera).

pattern, Step Functions charges $27.9 per 1 million such transitions. On the other hand, Unum costs, for 1 million transitions, (1) $0.42 for ∼200ms extra Lambda runtime to execute orchestration library code, (2) $2.79 for 1 DynamoDB write to checkpoint, (3) $0.279 for 1 DynamoDB read to check checkpoint existence, and (4) $2.79 for 1 DynamoDB write to garbage collect the checkpoint. In total, a basic transition in Unum is about 4.4x cheaper than the provider-hosted orchestrator on the same platform ($27.9 vs $6.279).

Table 2 shows the cost to run each of the applications we implemented in the `us-west-1` region. Unum is consistently, and up to 9x, cheaper than Step Functions for the applications we tested.

Figure 6 shows the cost to run each application using Unum broken down to each component: data store costs for writing and reading checkpoints, data store costs for writing coordination sets, data store costs for deleting checkpoints and writing coordination sets for garbage collection, Lambda invocation, and Lambda CPU-time for both the Unum runtime and user function. Storage costs, using DynamoDB, are the largest portion of overall cost and costs for writing to DynamoDB is the majority[2]. This includes writing checkpoints, writing to coordination set (either for fan-in for garbage collection) and deleting checkpoints for garbage collection.

Of course, developer-facing pricing is only a proxy for *actual* costs of hardware and human resources. However, it is clear that, in practice, Unum's costs are reasonable and, in fact, often lower than Step Functions. This suggests that at least applications that currently run on Step Functions could afford to run using Unum instead.

Furthermore, services that Unum builds on—FaaS schedulers and data stores—are core multi-tenant services that likely multiplex over a larger audience of applications than orchestrators for greater economies of scales. These services typically have enjoyed long periods of improvement already to make them efficient. Unum's design obviates the need to host yet-another service which frees up resources such that providers can focus on fewer core services in their serverless infrastructure.

Moreover, Unum automatically benefits from improvements to the underlying infrastructure and pricing schemes. For example, Azure's Cosmos DB provides similar performance and consistency guarantees to DynamoDB but charges 5x less to perform a write operation (the dominant cost of Unum's data store operations).

5.4 Case Study: ExCamera

ExCamera [20] is a video-processing application designed to take advantage of high burst-scalability on Lambda using custom orchestration. We compare our Unum implementation with three others: (1) the original hand-optimized ExCamera using the mu framework, (2) an implementation using a generalized orchestrator (gg) by the same authors, and (3) an optimized Step Functions implementation we wrote.

Both gg and mu employ standalone orchestrators to proxy inter-function communications, store application states and

[2]Writes in DynamoDB cost about an order-of-magnitude more than reads

invoke lambdas. However, mu uses a fleet of long-running identical lambdas where all application code is co-located and raw video chunks are pre-loaded, whereas gg lambdas are event-driven, task-specific and cannot leverage pre-loading. The application logic, though, is identical for gg ExCamera and mu ExCamera. Unum's ExCamera replicates the application logic from gg and mu. However, the Step Functions ExCamera implementation must serialize the encode and re-encode stages because Step Function's Map pattern requires all concurrent branches to complete before any fan-in starts (Figure 7).

5.4.1 Performance

Using the same experimental setup as the prior work (i.e., encoding the first 888 chunks of the `sintel-4k` [31] video using 16 chunks per batch and Lambdas configured with 3GB of memory), Unum is 7.1% faster than gg [19] and 10.5% slower than the original, hand-optimized ExCamera (Table 3). The original authors attributes the slower performance of gg ExCamera to the lack of pre-loading which is likely also the reason for Unum's slower performance.

But different from gg, Unum executes orchestration in a decentralized manner while gg has a standalone coordinator on EC2. The reduced number of network communications likely explains why Unum is slightly faster.

Comparing with Step Functions, Unum's design allows the flexibility to implement ExCamera's original application pattern where tasks start as soon as their input data becomes available, whereas the Step Functions implementation had to use the less-efficient Map pattern without the flexibility to add new orchestration patterns easily. As a result, the Unum ExCamera enables more parallelism between branches and is 16.7% faster than Step Functions.

5.4.2 Cost

Unlike Unum, neither gg nor mu aimed to reduce the cost of running serverless applications and neither discussed costs in detail. Nevertheless, there are several important factors in comparing Unum with gg and mu in relation to costs.

First, similar to Step Functions, gg and mu both rely on standalone orchestrators. Thus, the fundamental costs difference is also similar, namely Unum's use of storage vs gg's and mu's use of VMs. mu's orchestrator consists of a coordinator server as well as a rendezvous server [20], while gg's only has a coordinator server [19]. In the mu authors' experiments, they used a 64-core VM (`m4.16xlarge`) as the rendezvous server. Neither mu nor gg specified the instance type of its coordinator server. However, the cost of the rendezvous server, at the time of writing, is $3.20 per hour, or approximately $2352 per month.

Furthermore, standalone orchestrators must separately consider fault-tolerance in case of orchestrator failures. Most commonly, fault-tolerance is achieved by running multiple coordinating instances (replicas) of the service. As a result, production deployments of mu and gg would likely cost more.

Lastly, deploying an orchestrator per application or per user limits the ability to amortize costs through multi-tenancy. A provider-hosted orchestrator, such as Step Functions, can achieve larger economies of scale by serving many users concurrently with a single deployment.

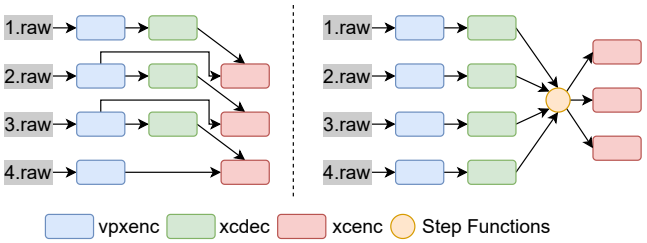


Figure 7: Unum ExCamera replicates the application logic from gg and mu where the re-encode stage (xcenc) of a branch can start immediately when the previous branch completes decoding (xcdec) and my own branch completes the initial encoding (vpxenc). Step Functions provides a Map pattern [30] for parallel workloads. However, branches in Map must be identical and Map does not support data dependencies between branches. As a result, to ensure previous branches' xcdec have completed, all branches must first finish and fan-in to Step Functions before starting the xcenc step, essentially serializing the stage.

ExCamera Implementation	Latency (seconds)
Original	76
Unum-aws	84
gg	90
Step Functions	98

Table 3: ExCamera performance. Unum is 7.1% faster than gg [19] and 10.5% slower than the hand-optimized implementation.

6 Related Work

Serverless Workflows. Many systems have recognized the need to augment serverless computing with support for composing functions to build larger and more complex applications. AWS Step Functions [3] defines serverless workflows as state machines using a JSON schema. Google Workflows [23] uses a YAML-based interface to list steps in a workflow sequentially and allows jumps among steps. Azure Durable Functions [6] uses a "workflow-as-code" approach, similar to driver functions, where the workflow logic is written in a programming language (e.g., C#, Python).

In all of these systems, orchestration is performed by a

standalone orchestrator. The nature and location of this component varies: in AWS Step Functions [3] and Google Workflows [23], it is provided by a cloud service that is separately hosted and billed. In Azure Durable Functions [6], it is an extension of the serverless runtime, and uses the same billing. In contrast to all of these, Unum proposes a novel **decentralized orchestration strategy** and runs entirely on unmodified serverless infrastructure without adding any new services or new components.

Kappa [41] addresses the lack of coordination between function and function timeout limits when executing large applications. Similar to Durable Functions, it also exposes a high-level programming language interface. Cloudburst [36] uses a specialized key-value store to enable low-latency execution of serverless functions. Users can express workflows as static DAGs and an executor program runs the DAG by passing data and coordinate via the key-value store. ExCamera [20] proposes the mu framework which uses a long-running coordinator to command a fleet of lambdas, each of which executes a state machines where user functions are the states. gg [19] proposes a thunk abstraction where each thunk executes as a deterministic lambda and expresses data dependencies between thunks as DAGs. gg uses a standalone coordinator to receive thunk updates and lazily launch thunks when their inputs become available.

Similarly, the above systems rely on a standalone orchestrator program. As the orchestrator program is not itself executing in a hosted environment, progress is not guaranteed when its host crashes. Also, progress is not checkpointed (except in Kappa), so workflows must restart from the beginning in that situation. In contrast, Unum relies only on a basic, highly available serverless platform. Thus, it guarantees progress under all faults, including the orchestrator. And Unum checkpoints each function result to minimize redundant computations when handling faults.

Beldi [40] and Boki [25] are two recent systems that provide exactly-once execution and transactions to stateful serverless applications. Both extend transactional features to specific application side effects supported by the system (e.g., DynamoDB writes). Developers use Beldi or Boki's library in user code when writing to a supported data store (e.g., DynamoDB) such that writes are executed only once. In comparison, Unum does not change how developers write user code and does not extend exactly-once guarantee to side effects in user code. Instead, Unum treats user code as a black box and ensures exactly-once semantics on a workflow-level. However, Unum users who want to ensure exactly-once when writing to DynamoDB can additionally use Beldi or Boki in their user code.

Programming Interface. Most serverless workflow systems require developers to write workflows with specialized interfaces. Some uses a declarative approach that defines workflows using JSON or YAML schemas (e.g., AWS Step Functions [3], Google Workflows [23]). Others allow expressing workflow as code (e.g., Durable Functions [6], Kappa [41], Fn Flow [18]).

Unum does not propose a new frontend for defining workflow. Instead, Unum aims to support any existing frontend that explicitly or implicitly expresses a directed graph where nodes are functions and edges are transitions between functions. Developers using Unum can choose the frontend that they prefer.

7 Discussion & Limitations

Unsupported applications. Unum supports a superset of applications that can be expressed using Step Functions, but there are applications that do not fit Unum's constraints. In particular, Unum only supports statically defined control structures. For example, Durable Functions expresses workflows dynamically as code and allows the developer to use arbitrary logic to determine what the next workflow step should be at runtime. This is not currently possible with Unum.

Measurement error. Due to the opaque design, implementation and pricing of production workflow systems, such as Step Functions, comparisons in our evaluations are limited in their explanatory power. In particular, we use the current *price* of Lambda, DynamoDB, and Step Functions as a proxy for the *cost* of providing these services. Of course, prices may be either lower or higher for a particular service than the underlying cost.

Code Complexity. While Unum affords users more flexibility, application-level orchestration increases code complexity for developers. Coordination and exactly-once execution require careful design and implementation to function correctly in a decentralized manner. Introducing application-specific optimization also needs additional developer efforts than using off-the-shelf patterns from provider-hosted orchestrators.

8 Conclusion

We designed and implemented Unum, an application-level, decentralized orchestration system that runs as a library on unmodified serverless infrastructure without requiring additional services. Our results show that basic serverless components—function schedulers and consistent data stores—are sufficient abstractions for building complex and fault-tolerant serverless applications. Moreover, Unum affords applications more flexibility, reduces costs and performs well compared with standalone orchestrators with similar execution guarantees.

Acknowledgments We thank the anonymous reviewers and our shepherd, Douglas Terry, for their insightful comments. We thank Landon Cox for his support and feedback during the early stage of this project. This work was funded in part by NSF Grant 2028869.

References

[1] Asynchronous invocation, AWS Lambda Developer Guide. https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html.

[2] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[3] AWS Step Functions. https://aws.amazon.com/step-functions/.

[4] AWS Step Functions Quotas. https://docs.aws.amazon.com/step-functions/latest/dg/limits-overview.html.

[5] AWS Step Functions Pricing. https://aws.amazon.com/step-functions/pricing/.

[6] Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[7] Azure Functions error handling and retries, Azure Functions Developers Guide. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-error-pages?tabs=csharp.

[8] Azure Functions reliable event processing, Azure Functions Developers Guide. https://docs.microsoft.com/en-us/azure/azure-functions/functions-reliable-event-processing#how-azure-functions-consumes-event-hubs-events.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995.

[10] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proc. VLDB Endow.*, 15(8):1591–1604, apr 2022.

[11] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[12] DynamoDB Pricing for On-Demand Capacity. https://aws.amazon.com/dynamodb/pricing/on-demand/.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, dec 1995.

[14] Error handling and automatic retries in AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html.

[15] Error handling in Step Functions, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html.

[16] Execution guarantees, Standard vs. Express Workflows, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/express-at-least-once-execution.html.

[17] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.

[18] Fn Flow. https://fnproject.io/.

[19] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[20] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[21] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An active distributed key-value store. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 323–336, USA, 2010. USENIX Association.

[22] Google Cloud Composer (GCC). https://cloud.google.com/composer.

[23] Google Workflows. https://cloud.google.com/workflows.

[24] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[25] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems*

Principles, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99 In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[28] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal extensions for Multi-Tenant Low-Latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, October 2018. USENIX Association.

[29] Introducing AWS Lambda Destinations. https://aws.amazon.com/blogs/compute/introducing-aws-lambda-destinations/.

[30] Map State, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-map-state.html.

[31] MPI Sintel Flow Dataset. https://paperswithcode.com/dataset/mpi-sintel.

[32] OpenFaaS Retries for functions. https://docs.openfaas.com/openfaas-pro/retries/.

[33] OpenWhisk Actions, Error Handling. https://github.com/ibm-cloud-docs/openwhisk/blob/master/error-handling.md.

[34] Retrying Event-Driven Functions, Google Cloud Functions. https://cloud.google.com/functions/docs/bestpractices/retries.

[35] Arnav Sankaran, Pubali Datta, and Adam Bates. Workflow integration alleviates identity and access management in serverless computing. In *Annual Computer Security Applications Conference*, ACSAC '20, page 496–509, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[37] Standard vs. Express Workflows, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html.

[38] Temporal Platform. https://docs.temporal.io/.

[39] David Tennenhouse. Active networks. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*, Seattle, WA, October 1996. USENIX Association.

[40] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020.

[41] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 328–343, New York, NY, USA, 2020. Association for Computing Machinery.

Enhancing Global Network Monitoring with *Magnifier*

Tobias Bühler* *ETH Zürich*
Romain Jacob *ETH Zürich*
Ingmar Poese *BENOCS*
Laurent Vanbever *ETH Zürich*

Abstract

Monitoring where traffic enters and leaves a network is a routine task for network operators. In order to scale with Tbps of traffic, large Internet Service Providers (ISPs) mainly use traffic sampling for such global monitoring. Sampling either provides a *sparse view* or generates *unreasonable overhead*. While sampling can be tailored and optimized to specific contexts, this coverage–overhead trade-off is unavoidable.

Rather than optimizing sampling, we propose to "magnify" the sampling coverage by complementing it with mirroring. *Magnifier* enhances the global network view using a two-step approach: based on sampling data, it first infers traffic ingress and egress points using a heuristic, then it uses mirroring to validate these inferences efficiently. The key idea behind *Magnifier* is to use *negative* mirroring rules; *i.e.*, monitor where traffic should *not* go. We implement *Magnifier* on commercial routers and demonstrate that it indeed enhances the global network view with negligible traffic overhead. Finally, we observe that monitoring based on our heuristics also allows to detect other events, such as certain failures and DDoS attacks.

1 Introduction

Monitoring transit traffic in Internet Service Provider (ISP) networks is difficult: most operators do not know precisely where traffic enters or leaves their infrastructure. This inability to correlate traffic network-wide makes it hard—if not downright impossible—to detect network-wide problems. As a consequence, operators occasionally learn about routing issues in their own network only via customers calling or opening up tickets.

Operators could use control-plane data to identify where traffic enters and leaves an ISP network; however, that is insufficient. Traffic towards the same destination is often load-balanced between multiple egresses; traffic from the same source prefix often enters via multiple ingresses; importantly, in case of failures or attacks, traffic may not follow the control plane. Data-plane measurements are thus necessary for accurate flow-level information. Unfortunately, such measurements are hard to scale with the Tbps of traffic crossing ISP networks nowadays.[1] Two common techniques to collect data-plane measurements are packet sampling and traffic mirroring. Both have advantages and disadvantages, making them suboptimal for detecting traffic ingresses and egresses.

Sampling-based approaches such as NetFlow [12] or sFlow [30] provide good coverage at the expense of precision and correctness. Often only a few flows are sampled, and even fewer are sampled at both the ingress and egress. We confirmed this by analyzing a 5-minute slice of NetFlow data (1/1024 sampling rate) extracted from *all* border routers of a Tier-1 ISP in Europe. The slice contains around 40 million flows, where a flow corresponds to packets sharing the same source and destination subnet as well as the same source and destination port. After discarding flows from/to the ISP-owned prefixes, we found that over all sampled transit flows, only 22% are sampled at both their ingress *and* egress, while 41% (resp. 37%) of flows are sampled only at the network ingress (resp. egress). Hence, a traffic matrix such as shown in Fig. 1a locates only 22% of sampled flows; we waste the information from all other sampled flows.

Mirroring-based approaches [33,39] provide high precision and *correctness* at the expense of scalability. Suppose we would mirror all traffic at network border routers. In that case, we could easily enhance packets sharing the same source and destination subnet with their ingresses and egresses, but that would double the network's traffic. Besides packet mirroring, techniques based on sketches [22] or in-band telemetry [25,26] also excel at gathering precise information but can only do so for a specific share of the traffic.

In this work, we ask ourselves whether we can combine the benefits of sampling and mirroring to mitigate their respective drawbacks. We answer this question positively and present

*The CRediT statement for this work is available in Appendix A.

[1] For example, Deutsche Telekom's IP network has a transit capacity exceeding 30 Tbps and reports up to 10 Tbps of IP traffic on average (3500 PB/month). Source: https://globalcarrier.telekom.com/business-areas/internet-content/ip-transit

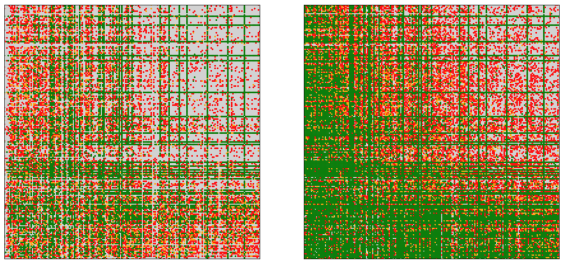
Magnifier, a system that enhances the global network view obtained via sampling using a two-step approach. First, we **infer the ingress and egress of flows** using a heuristic: packets that are "close" in the IP space tend to be routed similarly. This intuition has already been used successfully in other contexts, *e.g.*, to scale heavy-hitter detection using counters [25]. Assuming this holds, we search for the largest IP subnets for which packets appear to enter the network via the same ingress (resp. exit via the same egress) according to the sampling data available. We call these subnets *sentinels*. Fig. 1b shows the sentinel heuristic applied to our Tier-1 NetFlow data, which immediately magnifies the view: not only do we observe more flows for certain ingress–egress pairs (red to green), but we also reveal pairs which were not visible at all in the NetFlow-based matrix (Fig. 1a).

Naturally, this heuristic is not perfect; as sampling is sparse, we may lack important information to correctly identify ingress or egress points, or traffic may simply be rerouted over time. Thus, in a second step, we use **mirroring to validate the inferred ingresses and egresses**. To avoid mirroring a lot of traffic, the key idea is to install mirroring rules where we do *not* expect traffic; *i.e.*, if we infer that subnet s always enters via router R, we install a mirroring rule for s in all ingress routers, except R. In practice, this leads to little mirrored traffic because sentinels are most often correct. Thanks to mirroring, *Magnifier*'s ingress/egress inferences are **guaranteed correct**, which is a key feature of our design and an essential difference from other monitoring tools. Mirrored traffic reveals inference errors or traffic shifts in sub-seconds, which allows *Magnifier* to maintain a correct network view.

The main limitation of *Magnifier* is the number of mirroring rules to install, which, naively, is about one mirroring rule per sentinel on all border routers. For networks forwarding traffic which covers most of the IP space, this vastly exceeds the mirroring capabilities of today's routers. We thus investigate different strategies to cap the number of rules installed while harnessing most of *Magnifier*'s benefits.

Contributions

- We design *Magnifier*, a network monitoring system that combines sampling with mirroring to enhance the global view on traffic ingresses/egresses (*e.g.*, Fig. 1) while providing correctness guarantees.
- We implement *Magnifier* [3], run it on Cisco Nexus 9300 switches, and demonstrate that *Magnifier* increases the network view coverage with only limited traffic overhead and inference errors using real traffic traces (§ 6).
- We discuss (§ 4.2) and evaluate (§ 6.2.2) different strategies to scale *Magnifier* to large ISP networks by capping the number of mirroring rules required to *e.g.*, the top 1k sentinels while maintaining most of *Magnifier*'s benefits.
- We observe that, even without mirroring, changes in the number of found sentinels create an interesting signal for other monitoring applications, such as failure detection or DDoS protection (§ 6.4).



(a) NetFlow-based matrix. (b) *Magnifier*'s matrix.

Figure 1: By inferring ingress or egress points of sampled flows, *Magnifier* significantly improves the network-wide coverage (Fig. 1b) compared to using sampling only (Fig. 1a). These inferences are guaranteed correct by (the absence of) mirrored packets. *Dots represent the number of flows observed from an ingress router (x-axis) to an egress router (y-axis). Grey indicates no flow, red one flow, orange up to 4 flows, and green 5 or more flows. Data source: NetFlow samples from a large Tier-1 ISP.*

2 Overview

This section introduces the problem statement (§ 2.1) and *Magnifier*'s main building blocks (§ 2.2). Finally, we illustrate *Magnifier*'s behavior on a simple example (§ 2.3).

2.1 Problem statement

Can we combine the benefits of sampling and mirroring to design an *easy-to-deploy* system that produces *accurate*, *complete* and *timely* ingress/egress observations in ISP networks, where an "observation" consists of an IP subnet for which we know the correct ingress and egress points?

Ease of deployment The system should be usable in today's networks with no need for new or specialized hardware.

Accuracy The system should correctly infer subnets' ingress and egress points.

Completeness The system should generate observations for the largest possible portion of the IP space.

Timeliness The system should update observations in real-time based on newly-collected information; that is, information is processed quicker than it is collected.

2.2 Building blocks

Magnifier extends the coverage of ingress/egress observations using a two-step approach (Fig. 2): based on sampled data, it first *infers* the missing traffic ingress and egress points, then it *validates* these inferences using mirroring.

Inference *Magnifier* cross-correlates the sampled flows to identify IP subnets that are consistently routed via the same ingress or egress routers. For example, suppose we observe *all* sampled flows for a source prefix p enter via ingress router A. *Magnifier* learns that p is an implicit tag for "ingress A", which

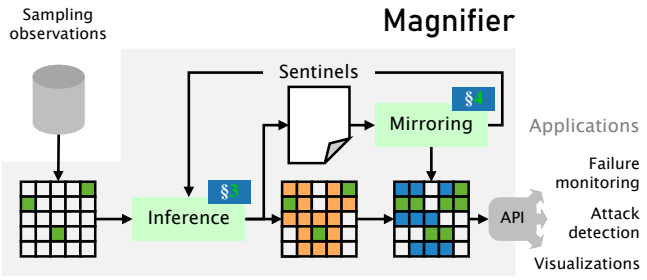


Figure 2: *Magnifier* uses sampled data to infer sentinels that predict IP subnets' ingress or egress points. *Magnifier* then validates sentinels at runtime using packet mirroring. This way, we can greatly extend the coverage of traffic ingress/egress observations usable by many applications.

enables to map any sampled flow sourced by p as entering via A—even if observed on a different router.

In addition, *Magnifier* leverages the hierarchical nature of the IP space: packets that are "close" in the IP space tend to be routed similarly. Thus, *Magnifier* searches for the largest IP subnets that share the same tags and postulates that all the IPs in these subnets are routed via the same ingresses or egresses. We call these largest subnets *sentinels*. Sentinels significantly extend the coverage of ingress/egress observation (compare Fig. 1). However, these sentinels may be incorrect; sampling may have missed important information, or traffic may simply be re-routed over time. Therefore, *Magnifier* uses mirroring to validate them at runtime.

Validation The key idea behind *Magnifier* is to validate the sentinel inferences using *negative mirroring*; *i.e.*, to deploy mirroring rules where we expect traffic *not* to go. Negative mirroring is efficient because sentinels are often correct in practice; therefore, we mirror only a little traffic. Fundamentally, this guarantees that *Magnifier*'s outputs are correct; all prefixes covered by sentinels either have correctly identified ingress/egress or carry no traffic. Otherwise, traffic is mirrored, which provides additional observations and allows *Magnifier* to maintain and improve its accuracy over time.

Optimization The main limitation of *Magnifier* lies in the number of mirroring rules that can be activated simultaneously on one router. By aggregating subnets together, sentinels effectively limit the number of mirroring rules that must be deployed, but this remains a constraint for large ISP networks. *Magnifier* supports multiple rule deployment strategies to respect a given rule budget per router while optimizing for different properties (*e.g.*, IP space coverage).

2.3 Illustrative example

While mirroring rules generate additional traffic, they are essential to *Magnifier*, illustrated with an example (Fig. 3): p_0 to p_7 are eight /24 prefixes belonging to the same /21; most of the traffic comes from p_0, with sporadic traffic from other prefixes. Let's assume that we only sample traffic from p_0,

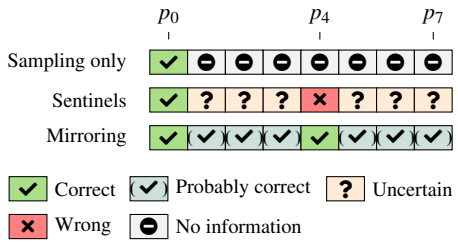


Figure 3: Sampling provides information about the sampled prefixes only. The sentinel inference extends the coverage, but it is uncertain and can make wrong assumptions without any means to detect them. With mirroring, these inferences can be validated, leading to either correct or probable inferences.

which enters at ingress A. One can hypothesize that all p_0 traffic enters via A, but nothing can be said about p_1 to p_7.

Since no sampled packet contradicts this hypothesis, we infer that all eight /24 enter via A; the whole /21 is a sentinel for ingress A. This inference is, however, uncertain for seven /24 prefixes without any data. Some traffic from p_4 enters via another ingress, but as long as we do not sample p_4 traffic, we will not detect the wrong inference.

We now use mirroring to validate the sentinel: all routers except A mirror packets for the /21. At first, no packet is mirrored: this indicates either that the sentinel is indeed correct or that there is no traffic *at all* on prefixes that would enter via another ingress. Thus, for the seven prefixes without sampling data, *Magnifier* concludes that the ingress is "probably A".

Finally, ingress router B mirrors packets coming from p_4. *Magnifier* now learns that the /21 sentinel was incorrect. We recompute sentinels, which leads to two /22 sentinels, one for A and one for B. Once the corresponding mirroring rules are installed, *Magnifier* confirms that p_0 and p_4 enter via A and B respectively, and that p_1 to p_3 (p_5 to p_7) probably enter via A (resp. B) as we would otherwise observe mirrored packets.

Conclusion The mirroring rules are essential to validate the sampling-based inferences. Once active, *Magnifier guarantees* that the inferences are either correct or that prefixes for which they are wrong do not carry any traffic at all.

3 Ingress & egress identification

In this section, we define the notion of "sentinels" and present an efficient algorithm to find them (§ 3.1). We then discuss sentinel subnet size tradeoffs (§ 3.2) and finally show how *Magnifier* uses these sentinels to match ingress and egress observations in sparsely sampled data (§ 3.3).

3.1 Sentinel search and definition

Definition A sentinel is an *IP subnet* which always enters or leaves the network via *one* network device. Therefore, a sentinel identifies this device whenever a flow from/towards

Algorithm 1 Sentinel search algorithm

```
start ← starting subnet size
end ← ending subnet size
table[IP, device] ← search IPs and network devices
sentinels ← {}
for start ⩽ S ⩽ end do
    table[IP_new] ← ((IP >> (32 − S)) << (32 − S))
    aggregated ← table.groupby(IP_new)[device]
    result[n] ← nunique(aggregated[device])
    sentinels += (result[n] == 1)
    table −= sentinels
end for
return sentinels
```

the IP subnet is observed *somewhere* in the network. As an example, if flows towards 1.2.3.0/24 *only* leave the network via egress router R, we say that 1.2.3.0/24 is an egress sentinel for R. Note that a single sentinel can cover numerous flows.

Types We can distinguish four types of sentinels depending on which IP address we look at (source or destination) and which traffic direction is identified by them (ingress or egress). For example, we can speak about ingress source sentinels. However, unlike specified differently, the remaining sections will only focus on two types of sentinels (i) ingress source sentinels (abbreviated as *ingress sentinels*); and (ii) egress destination sentinels (abbreviated as *egress sentinels*). Currently, *Magnifier* only considers IPv4 sentinels, but *Magnifier* can be applied to the IPv6 address space as well.[2]

Search algorithm *Magnifier*'s sentinel search algorithm takes flow samples as input. They contain, among others, the identifier of their origin router and the packet source and destination IP addresses. In addition, we define a start and end subnet size over which the algorithm searches for unique subnets to reveal sentinels. Algorithm 1 highlights the main sentinel search steps. The for loop iterates from the start to the end subnet size. In each iteration, we extract the corresponding subnets from the IP addresses of the collected flow samples. All flows belonging to the same subnet are aggregated. If one aggregate only contains samples from the same device, *Magnifier* has found a sentinel, removes the samples from further search iterations, and eventually returns the sentinels.

3.2 Sentinel subnet sizes

Algorithm 1 returns a subnet as a sentinel as soon as it only contains flow samples from one device. However, it is also possible that a smaller subnet would cover all these samples. Fig. 4 shows a simple example. The network forwards traffic from three different /24 subnets. We collect samples from the two green /24 subnets (ingress router X). Unfortunately, we do *not* sample a packet from the orange /24 subnet (ingress router Y). Algorithm 1 would return the corresponding /22 subnet as an ingress sentinel. After installing corresponding mirroring rules (§ 4.1), *Magnifier* will detect that this sentinel is invalid as it contains flows from two different ingress routers (X and Y). If we would search for smaller subnets, we could either return one valid /23 sentinel or two valid /24 sentinels.

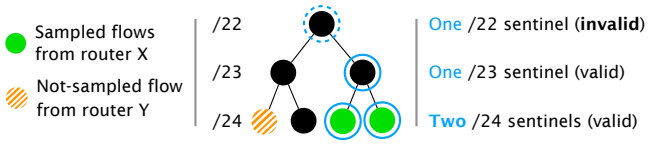


Figure 4: The sentinel amount and coverage depend on the subnet size. The "largest" /22 sentinel is invalid, whereas one /23 sentinel or two /24 sentinels are valid.

This simple example shows a fundamental tradeoff between the subnet size of found sentinels, the number of sentinels, and their validity. In general, sentinels based on smaller subnets are more likely to be valid but require more mirroring rules to be validated. Experimentally, we find that starting at /16 and ending at /24 yields good performance; starting at bigger sizes does not help as we rarely see such big prefixes in BGP, and it is unlikely that they are unique to a single ingress/egress, while /24 is the smallest globally routed prefix size [35]. As a consequence, the search sizes also influence the number of required validation mirroring rules (§ 4.1) and, therefore, the required router resources.

3.3 Sentinel-based ingress & egress detection

Magnifier uses the found sentinels in two ways. First, for each sentinel type, it tracks the number of sentinels found per device in the network. § 6.4 shows that the number of sentinels is rather stable and changes can reveal unexpected network behavior. The second use case exploits the uniqueness property of sentinels. Let's assume we have found a (valid) egress sentinel for router X. For each flow towards the sentinel's subnet—no matter if we observe a corresponding packet on an ingress or another device—we instantaneously know that it will leave the network over X. Similarly, we can identify flow ingresses based on ingress sentinels. *Magnifier* uses this information as input for its ingress/egress observations.

4 Mirroring-based validation

In this section, we explain how *Magnifier* uses traffic mirroring to validate the ingress and egress sentinels produced by the sentinel search algorithm (§ 4.1). To ensure that *Magnifier* can adhere to an operator-given budget of mirroring rules, we introduce two different rule deployment strategies and discuss additional optimization possibilities (§ 4.2).

[2] Current IPv6 allocation strategies (*e.g.*, https://www.ripe.net/publications/docs/ripe-738#5) are favorable for *Magnifier*. The same AS tends to be allocated large IP blocks that we can use as sentinels.

4.1 Validating found sentinels with mirroring

Magnifier uses *negative* rules to validate the sentinels it finds. Negative rules are placed on devices that are *not* expected to see matching traffic. For instance, to validate that an ingress sentinel belongs to ingress `I`, *Magnifier* places negative mirroring rules mirroring traffic for the sentinel's source subnet at all ingress routers except `I`. The negative mirroring rules will never generate any packets if the sentinel is valid. For invalid sentinels or sentinels which become invalid over time (*e.g.*, a forwarding change), the mirrored packets inform *Magnifier* immediately, and we can update our inferred ingress or egress observations. *Magnifier* then includes the mirrored data in the next sentinel search to find better sentinels.

ACL-based mirroring *Magnifier* relies on existing features such as ERSPAN [9] to mirror traffic from a router. Depending on the router model and capabilities, there are different ways to define the mirrored traffic. We can temporarily mark packets (*i.e.*, [39]) or directly assign a list of subnets to the mirroring session. We use so-called Access Control Lists (ACLs) in all these cases. An ACL is a list of subnets that matches on the forwarded traffic and defines the mirrored traffic. Our "mirroring rules" are entries in an ACL.

Deployment and activation of mirroring rules To deploy its mirroring rules, *Magnifier* interacts with a Python script that runs directly on the router CPU. Via its arguments, *Magnifier* tells the script the mirroring rules to add to the ACL. The script uses *e.g.*, Cisco's Python API [6] to perform the changes. However, naively adding entries to an ACL that is already connected with an active ERSPAN session can result in unexpected mirroring behavior for at least two reasons: (i) adding new entries takes some time and *Magnifier* cannot predict at which point in time a new mirroring rule is active; (ii) the Ternary Content Addressable Memory (TCAM) region which handles the ACLs/mirroring rules is limited. *Magnifier* handles (i) by pre-deploying inactive mirroring rules and (ii) with techniques explained in § 4.2.

To pre-deploy mirroring rules, *Magnifier* first adds entries to an ACL that is not yet active, *i.e.*, connected with an ERSPAN session. The ACL entries do not yet take space in the TCAM. Once the ACL contains all mirroring rules, another script activates the entire ACL, simultaneously enabling all mirroring rules. In practice, *Magnifier* always iterates between *two* ACLs. One is currently actively mirroring traffic while the other one is populated. Once the second ACL is ready, we switch between them. Due to this deployment strategy, *Magnifier* is not negatively influenced by frequently changing mirroring rules/sentinels (see § D.4) as we always activate a new pre-deployed ACL. Furthermore, this only affects the mirrored traffic; *Magnifier* does not impact the production traffic.

Magnifier can also add a parameter to the scripts which defines how long an ACL should be active. The script will then automatically, *i.e.*, without any external interaction, deactivate the mirroring rules once the defined timeout expires.

4.2 Limiting the amount of mirroring rules

The amount of mirroring rules which a single router can support is limited. Not only is the entire TCAM limited, other features (*e.g.*, traffic engineering) use the same memory space and compete with *Magnifier*'s mirroring rules. For this reason, *Magnifier* supports multiple deployment strategies to adhere to an operator-given budget of mirroring rules. In the following paragraphs, we describe two strategies, but network operators can easily define their own sorting algorithm to control which mirroring rules they deploy first.

Deployment based on sentinel size The first strategy maximizes the sentinel IP space covered by mirroring rules. As each mirroring rule is connected to a sentinel with a specific subnet size, *Magnifier* first orders all sentinels of an ingress or egress based on their subnet size. *Magnifier* then iterates through all network border routers in a round-robin fashion and deploys mirroring rules for the sentinel with the biggest subnet (*i.e.*, the subnet which covers the most IP space). This process ends if either the mirroring rule budget is reached or every mirroring rule is deployed.

Deployment based on sentinel activity The second strategy prioritizes the most active (amount of sampled packets) subnets/sentinels. In other words, we make sure that the inferred ingress or egress points for the most active subnets are validated by mirroring. To this end, *Magnifier* iterates through all border routers in a round-robin fashion and first deploys mirroring rules for the sentinels that are based on the largest number of sampled packets. Random packet sampling–by design–favors large, active flows. Therefore *Magnifier* indirectly deploys mirroring rules for the most active subnets. We evaluate both deployment strategies in § 6.2.2 and § D.3.

Network-specific optimizations *Magnifier* further reduces the amount of mirroring rules using network-specific knowledge. For example, some ISP border routers only connect to customers, and the operator knows exactly which IP addresses belong to them. That limits the possible source addresses entering the ISP over these ingresses (assuming no IP spoofing). On these devices, *Magnifier* does not need to install mirroring rules which belong to IP subnets outside of the customer's prefixes as we should never receive contradicting traffic.

5 *Magnifier*'s controller

Magnifier's controller collects and combines the sampled and mirrored packets, finds new sentinels, deploys and activates the corresponding mirroring rules, and uses the newest data to generate accurate and up-to-date ingress/egress observations. This section first explains how the different pieces work together before introducing *Magnifier*'s API. § B contains details about *Magnifier*'s controller placement.

Controller design *Magnifier*'s control flow works in iterations that align to the system component with the longest runtime.

As various tests on real hardware show, this is usually the time it takes to deploy mirroring rules on the routers. Fig. 5 shows the entire process. *Magnifier* uses the collected sampled and mirrored data in iteration `N-2` (and optionally `N-3` or older iterations) to compute sentinels and their mirroring rules. Based on the operator given rule budget, *Magnifier* sorts the sentinels according to the deployment strategies in § 4.2. While iteration `N-1` is running, *Magnifier* pre-deploys the newly computed mirroring rules on the routers. As soon as *Magnifier* deploys the last rule (or once we reach the defined iteration time), it switches to iteration `N` and activates the pre-deployed mirroring rules after deactivating the old ones. Finally, *Magnifier* uses the collected sampled and mirrored data and the inferred ingress and egress points from the newest sentinels to compute accurate and up-to-date ingress/egress observations.

***Magnifier*'s API** *Magnifier*'s API supports four distinct primitives. First, `enhance_subnet(S)` returns the available ingress and/or egress data related to subnet `S`. Second `get_interfaces()` returns the relationship between sentinels and their interfaces. *Magnifier* can infer the corresponding interfaces based on sampling data and/or the observed MAC addresses in mirrored packets. Third, `get_matrix()` generates the most up-to-date ingress/egress matrix. Each cell contains the number of observed packets and bytes (reported by sampled packets) for an ingress/egress observation. In addition, a validity bit indicates inferences that are currently validated with mirroring rules. Finally, `get_counts()` outputs the number of found sentinels per device grouped by sentinel type. In § 6.4 we use this API call to detect network problems based on data from a real Tier-1 ISP.

6 Evaluation

This section evaluates *Magnifier* in detail. After introducing the evaluation setup (§ 6.1), we first focus on *Magnifier*'s performance in simulation and on real hardware devices in our lab (§ 6.2). Afterward, we perform a detailed comparison with the Everflow system (§ 6.3) before we highlight that *Magnifier* also works with data from a real ISP (§ 6.4).

6.1 Evaluation setups, datasets, and metrics

Setups We evaluate *Magnifier* in a simulation setup without any resource constraints and a lab setup on real hardware with its corresponding limitations. Our lab setup contains two Cisco Nexus 9300 switches (C93108TC-FX) [10], and a larger Nexus 7009 switch (N7K-C7009) [8]: an older[3] but more resourceful model that we use for benchmark experiments.

We illustrate the lab setup in Appendix § D.1. We establish four parallel connections between the two Nexus 9300 switches, each emulating a network ingress. The first switch receives and samples the traffic using sFlow (sampling rate

[3]Released in 2011 and no longer sold.

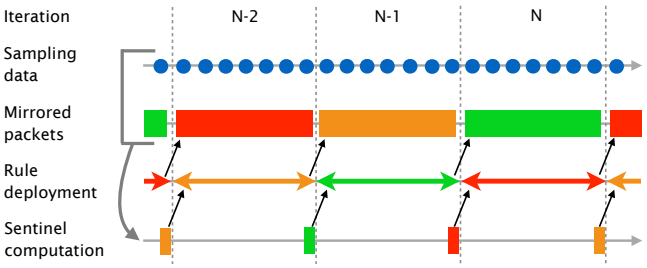


Figure 5: *Magnifier*'s control flow works in iterations based on the mirroring rule deployment time. Rules for sentinels based on `N-2` are deployed in `N-1` and active in iteration `N`.

1/4096[4]). It then forwards to the second switch, which mirrors the traffic according to the configured rules. *Magnifier*'s controller runs on a server and collects sampling and mirroring data. As these switches are limited to 512 mirroring rules, we used a fixed budget of 500 rules per emulated ingress point.[5] Unless otherwise specified, *Magnifier* prioritizes sentinels according to the `activity` ordering (§ 4.2).

Our simulation setup is an idealized version of the lab setup. It instantaneously starts mirroring for any prefixes, has unlimited memory space for mirroring rules, and removes rules after their first mirrored packet. The simulator is written in Python and publicly available [3]. Unless specified differently, we always consider *Magnifier*'s iterations to be 60 s long. For the `N`-th sentinel computations, we take sampling and mirroring data from iterations `N-1` and `N-2` (see Fig. 5).

We focus on ingress sentinels in the evaluation, *i.e.*, source IP prefixes unique to one ingress. However, the results also apply to egress sentinels. For example, BGP selects the best route for each prefix that is assigned to a single egress. *Magnifier* identifies (part of) these prefixes as egress sentinels (§ 6.4). As a result, one major problem is how to split traffic over different ingress points: *Magnifier*'s performance depends on the assumption that prefixes close in the IP space get routed similarly. We study this dependency using three IP space to ingress mappings: `random` (least favorable for *Magnifier*), `static`, and `permuted` (most favorable). The `random` approach splits the *destination* IP space into n[6] equal slices and assigns one destination IP slice to each ingress point; as a result, source IPs are randomly assigned to one ingress, and this assignment changes frequently. The `static` approach assigns each source /24 prefix *statically* to one random ingress point; however, close IP space is still distributed over different ingresses. Finally, `permuted` splits the *source* IP space into n equal slices and permanently assigns each slice to one of the n ingresses. Then we permute a fixed percentage of /24 source prefixes by moving them to different ingresses. This way, we preserve most of the existing IP structure. A `permuted` 0% assignment results in a perfect mapping for *Magnifier*.

[4]The highest configurable rate on this model; we get the *most* samples.

[5]The four emulated ingresses share the budget. We use TCAM carving [7] to increase the space for our mirroring rules to 2048 (by taking it from other features) to enable the original budget (512) per ingress.

[6]Lab: 1st: 0.0.0.0/2; 2nd: 64.0.0.0/2; 3rd: 128.0.0.0/2; 4th: 192.0.0.0/2

Datasets We use two datasets: one actual packet trace based on CAIDA data and NetFlow samples from a Tier-1 ISP.

The packet trace is based on a 2018 CAIDA trace [4] (1.5 billion packets; one hour long), adjusted to be used in both our setups: (i) We modified the packet MAC addresses to match the lab setup. (ii) We added random payload bytes (removed from CAIDA traces) to match the specified packet sizes. (iii) We moved all destination IPs from 224.0.0.0/4 to a different /4 prefix as this prefix is reserved for IP multicast and led to unexpected packet forwarding on the switches. Replaying the trace at normal speed using `tcpreplay` [16] exhibited anomalies (packet loss and delays). Therefore, we slowed the replay by 10x, resulting in an average of 45k packets per second. Our simulations also use normal and faster speeds to emulate increasing traffic load.

The second dataset contains sampled (rate 1/1024) NetFlow data from all border routers (more than 100) belonging to a large Tier-1 ISP in Europe. The dataset spans over one hour of peak time in the evening of a weekday in 2018. The IP addresses are anonymized by replacing source and destination IPs with the best matching prefix from the full BGP table or the corresponding /24 prefix, whichever is more specific.

Metrics We use the following performance metrics and report the mean over 60 iterations (30 for 2× replay speed).

Coverage Quantifies the amount of traffic for which the ingress point is *correctly* identified. We consider both per-prefix coverage—*i.e.*, the number of /24 covered—and per-packet coverage—*i.e.*, the percentage of covered packets from the input trace.

Mirrored traffic volume Quantifies the overhead in terms of mirrored traffic, as a percentage of the total traffic.

Mirroring rule space Quantifies the number of mirroring rules (ACL entries).

Deployment speed Quantifies how long it takes to either add new mirroring rules or deactivate an installed rule.

6.2 *Magnifier*'s performance

This section details *Magnifier*'s performance. We first show that *Magnifier* greatly enhances the prefix coverage compared to sampling only (up to 80×) *and* that the ingress points are validated with mirroring rules. This is achieved while mirroring less than 0.3% of traffic. We then analyze methods to limit the number of mirroring rules required. Finally, we confirm that *Magnifier* runs and performs well on real hardware.

6.2.1 Coverage and mirrored traffic volume

We first use our simulation setup to evaluate *Magnifier*'s coverage in different scenarios.

Setup We use our simulation setup and the CAIDA dataset. We vary the trace replay speed (traffic load) and compare the coverage achieved by *Magnifier* by using sampling only. We compute sentinels, install mirroring rules at the start of each iteration, and compute their coverage values at the end unless mirrored traffic invalidated them. Only these count to the shown coverage values (mean over all iterations).

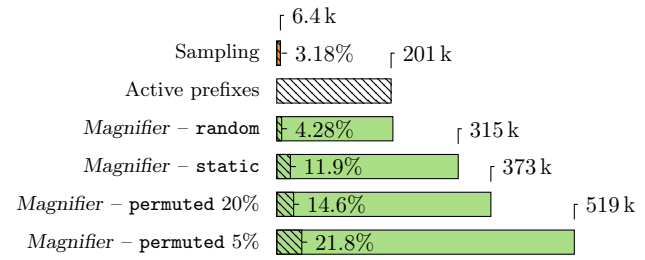


Figure 6: Amount of covered /24 source prefixes by *Magnifier* and sampled data assuming unlimited mirroring resources. *32 border routers, 1/1024 sampling rate, and real replay speed.*

Per-prefix results Fig. 6 shows the per-prefix coverage with 32 border routers, 1/1024 sampling rate, and real-time replay speed. Sampling covers ≈ 6.4k of the active /24 prefixes in the trace, for which we could consider the corresponding ingress point as identified, although without any confirmation that it is valid for all packets belonging to the /24 prefix.

By contrast, we immediately see that *Magnifier* enhances these inferences for all different prefix-to-ingress mappings in at least two ways. First the number of covered /24 prefixes increases to ≈200k (`random`), ≈315k (`static`), ≈370k (`permuted` 20%) and ≈520k (`permuted` 5%) respectively. Second, *Magnifier* covers prefixes that are currently active in the CAIDA traces (dashed boxes). The active prefixes increase from ≈4% (`random`) up to ≈20% (`permuted` 5%).

These observations highlight two principles of *Magnifier*: (i) our sentinel heuristic greatly enhances the prefix coverage around sampled data; and (ii) *Magnifier* remains a data-driven system. It has difficulties covering active prefix space that is not sampled using sentinels of reasonable sizes—hence the small % of active prefix coverage (Fig. 6, stripes).

Even more important, for every sentinel validated by mirroring rules, *Magnifier* immediately reports if an ingress inference is no longer valid or enhances new flows (which get active over time) with ingress information. These results are more visible in the per-packet coverage analysis.

Per-packet results Fig. 7 shows the per-packet coverage (left) and mirrored traffic volume (right) with 32 border routers and a 1/1024 sampling rate for varying replay speeds and traffic-to-ingress assignment strategies.

The left plot shows that *Magnifier* achieves an increasing per-packet coverage from ≈20% (`random`) up to ≈80% (`permuted` 5%) which can be surprising given the lower active prefix coverage (Fig. 6). This is explained by the nature of the CAIDA trace, which contains a small number of heavy-hitters and a lot of /24 source prefixes that only carry a few packets. 10% of source /24 IP prefixes account for more than 90% of the packets in 60 s trace data (§ D.2). Hence, *Magnifier* often samples and covers these prefixes with sentinels.

These results nicely show the different trade-offs of our assignment strategies. For `random`, the ingress of packets is constantly moving, which makes it difficult to find valid sentinels, while at `permuted` 5%, the assignments are static and *Magnifier* can often find large sentinels which cover a lot of packets. The "real" coverage value is somewhere in between.

To compare, Fig. 7 also contains two sampling-based inferences (without *Magnifier*'s enhancements). The violet line near zero represents a lower bound. We only infer the ingress for the sampled packets. As an upper bound, we consider all the sampled packets in the `permuted` 5% assignment and naively assume that a single sampled packet immediately reveals the ingress point for all other packets belonging to the same source /24 prefix. Note that we can only plot these values because we have the full ground truth data from the CAIDA trace. An operator would *not* know if these inferences are correct. For bigger traffic loads, the upper bound is better than *Magnifier*'s per-packet coverage. This is due to invalidated sentinels: *Magnifier* searches for large sentinels based on sampled packets, likely to come from heavy-hitter flows. Suppose a non-sampled /24 prefix covered by that sentinel is mapped to a different ingress and carries even only one packet. In that case, it triggers a mirroring rule and invalidates the entire sentinel, and *Magnifier* loses all its coverage.

The right plot in Fig. 7 shows a low percentage of mirrored traffic for all assignment strategies (between 0.3% and 0.01%). As expected, a `random` assignment often leads to invalid sentinels and thus more mirrored packets.

Finally, we observe that larger traffic loads yield better performance. With more traffic, *Magnifier* collects more samples per iteration, computes more accurate sentinels, and achieves better coverage and less mirrored traffic. We show additional results in § D.3: Performance decreases with the number of routers in the `random` case—as the previously discussed mapping strategy gets worse—but remains nearly unaffected in the `static` and `permuted` cases (Fig. 15). Moreover, more sampled packets (higher sampling rate) result in better input data and thus performance improvements (Fig. 16).

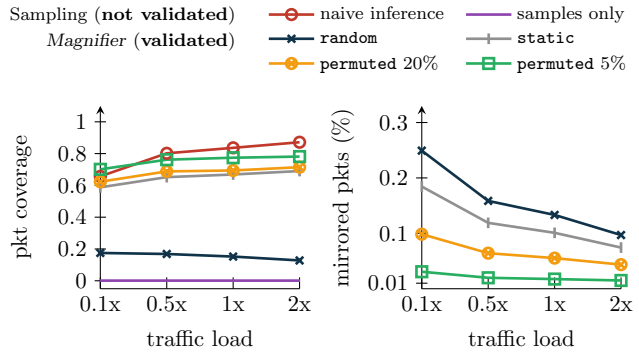


Figure 7: Amount of covered packets and mirrored traffic for different assignment strategies and inferences based on sampled packets only. *32 border routers and 1/1024 sampling rate.*

Conclusion *Magnifier* greatly increases the per-prefix coverage compared to sampling (up to 80×) while validating all ingress points with mirroring rules. *Magnifier* achieves this while mirroring less than 0.3% of traffic and translates into a per-packet coverage of up to 80%.

6.2.2 Impact of limited mirroring budget

We now show that *Magnifier* also performs well when limiting the number of mirroring rules installed per router.

Setup We use the same setup as before and compare the coverage achieved by *Magnifier* with different bounds on the number of validated sentinels for two sentinel selection strategies (§ 4.2): `activity` (covering most sampled packets) & `size` (largest subnet sizes). The number of validated sentinels is an upper bound for the number of mirroring rules required per router; in the worst case, all sentinels belong to one router, resulting in one rule per sentinel on *all* the other routers.

Results Fig. 8 compares *Magnifier*'s per-packet coverage achieved with different numbers of validated sentinels: 500, 1k, 5k and unlimited; using the same settings as in Fig. 7. We show results for the `permuted` 5% (left) and `static` (right) assignment strategy, additional plots can be found in § D.3.

More validated sentinels achieve a higher coverage and generate more mirrored traffic. The top `size` sentinels have the highest chance of being invalidated by un-sampled prefixes and generate more traffic than their `activity` counterparts.

The `activity` selection achieves much better per-packet coverage than `size`, which is expected since `activity` prioritizes sentinels covering the most active prefixes. As the trace contains many heavy-hitters (previous discussion), even as few as 500 sentinels are enough to yield good packet coverage. Note that for 0.1× traffic load in the top left plot, the number of sentinels is smaller than 5000, resulting in the same coverage

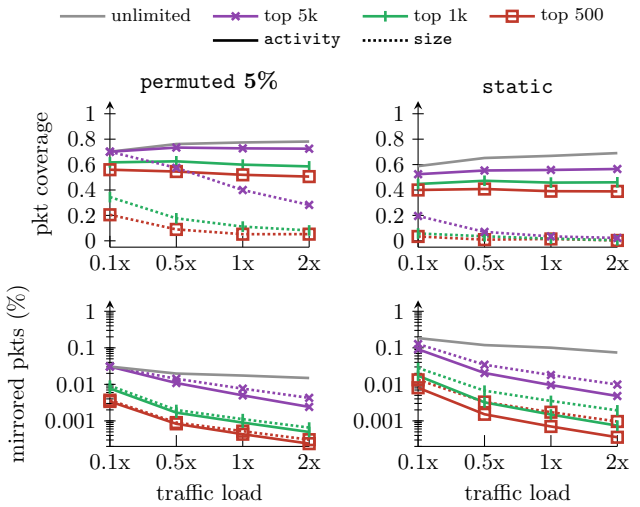


Figure 8: Coverage and mirrored traffic amount for different top sentinels ordered by `activity` or `size`.

Technique	Covered /24 prefixes
Sampling	6.4k
[activity] top 500 static	4.1k
[activity] top 500 permuted 5%	29.4k
[size] top 500 static	9k
[size] top 500 permuted 5%	47.3k

Table 1: Covered /24 source prefixes by *Magnifier* and sampling only considering the top 500 sentinels (activity and size ordering) in the static and permuted 5% assignments.

values for activity and size. We also see that the activity coverage values remain more or less constant even if the traffic load increases which is not the case for size.

The size selection favors sentinels centered around sparse samples in a relatively empty prefix space; this results in a low per-packet coverage (Fig. 8), but in a large per-prefix coverage (Table 1). As we can see, activity really prioritizes sentinels around a few selected prefixes resulting in fewer covered prefixes than sampling (static assignment). However, a size ordering can easily exceed the number of covered prefixes by up to seven times, even if we only take the top 500.

Conclusion *Magnifier*'s performance is maintained when limiting the deployed mirroring rules. The top 1k activity sentinels are sufficient to achieve up to ≈ 50% packet coverage while mirroring less than 0.05% of traffic (static case).

6.2.3 Comparison with the lab setup

We now show that our hardware-based results match the simulation ones, validating *Magnifier*'s performance in practice.

Setup We use our lab setup (Nexus 9300 switches), which has two main differences from the simulation: we only have 500 mirroring rules per router, and there are delays to install and delete rules. We use the random assignment strategy and fill the 500 mirroring rules with the top 500 activity sentinels. For a fair comparison with the simulation, we consider iteration times of 60 s. *Magnifier* needs ≈20 s to install all mirroring rules and then activate them. Afterward, we start to delete the rules which mirror packets. We compare this with the corresponding simulation results *i.e.*, 4 border routers, 1/4096 sampling rate, and 0.1× replay speed.

Results Fig. 9 shows the amount of covered /24 prefixes for sampled data only and the validated sentinels. We first notice that the coverage for sampled packets in our simulation (297) is slightly higher than on the switches (268). This can be explained by the different setups. All four ingress routers run on one Nexus 9300 (§ 6.1), which is not transparent to the sFlow-based sampling unit. Therefore, we get random packet sampling over all the traffic while the simulation performs packet sampling for each ingress device independently. This also shows in the achieved coverage values using the top 500 activity sentinels: ≈10.4k prefixes in the simulation, ≈8.7k prefixes on the hardware. We also have to consider that

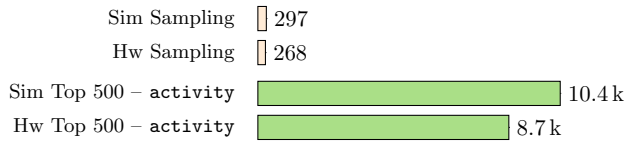


Figure 9: Covered /24 source prefixes by *Magnifier* and sampled data in simulations and on Nexus 9300 switches. random *assignment, 1/4096 sampling rate, and 0.1× replay speed.*

we need additional time to deploy the mirroring rules on the switch. Thus, a few more sentinels get invalidated compared to the simulations; and no longer count to the coverage values. The packet coverage values (not shown) are also comparable between the simulation (17.0%) and the hardware (16.1%).

Finally, we evaluate the percentage of mirrored traffic. We notice that the deactivation of active mirroring rules works well. In the worst case (active rules mirror for the entire 60 s), *Magnifier* would mirror 2.3% of the overall traffic. This value is reduced to 1.4% if we start to deactivate rules. However, we are still above the optimal simulation results (less than 0.1%), where we can deactivate mirroring instantaneously.

Conclusion The hardware results closely follow our simulations regarding achieved coverage. However, *Magnifier* needs more time to install and deactivate mirroring rules, resulting in additional mirrored packets. To reduce the amount of mirrored traffic, operators can use existing hardware features to rate-limit the mirrored traffic on the switch [11].

6.2.4 Micro-benchmarks

We now perform micro-benchmarks on the hardware switches to assess (i) how many mirroring rules each device supports, how long it takes to (ii) deploy them, and (iii) deactivate them.

Results–Mirroring rule space With the default configuration of the Nexus 9300 switch, we can deploy 512 rules and up to 2048 in the current lab setup (TCAM carving [7]). On the Nexus 7009, we can deploy ≈32k rules using one TCAM bank and ≈128k rules if we chain all four TCAM banks together.

Results–Rule deployment time We measure how long it takes to deploy a set of mirroring rules on our two devices. During our tests, we realized that deploying the rules over multiple parallel sessions between *Magnifier* and the switches is beneficial. Four parallel sessions worked well for us. Table 2 shows the mean deployment times over ten measurements each. They include the session setup and round-trip time between *Magnifier* and switch. We see that the deployment time is not strictly linear in the number of rules. We conjecture that caches and buffers allow deploying a small number of rules quickly, but this no longer works for larger number of rules. We also see that the (newer) Nexus 9300 switch needs less time than the Nexus 7009. We can deploy 2000 rules in ≈18 s, which matches our observations in § 6.2.3 (500 rules for 4 ingresses on one device). We expect that the rule deployment time will continue to decrease with more powerful/newer devices.

Number of rules	100	500	1000	2000	5000
Nexus 9300	3.5s	5.5s	8.4s	17.8s	112s
Nexus 7009	2.6s	7.5s	21.7s	74.9s	475s

Table 2: Mirroring rule deployment times.

Note that even if we cannot activate 5k+ rules on the Nexus 9300 switch with the current TCAM carving (§ 6.1), we can still deploy them. Overall, these results confirm *Magnifier*'s design (Fig. 5) which aligns the iterations to the rule deployment time. Especially as the sentinel computation time is negligible (≈ 1 s on the CAIDA trace.)

Results–Rule deactivation time We finally measure the rule deactivation time on the Nexus 9300 switch. We generate 100 ping probes per second and deactivate a matching mirroring rule as soon as we receive the first mirrored probe. The deactivation time is the difference between the timestamp of the first and last mirrored probe, including the round-trip (≈ 0.5 ms) and session setup time between switch and controller. This setup is representative of an ISP deployment where close, dedicated control servers could quickly deactivate rules (§ B). We repeated the experiment ten times. The mean deactivation time is 1.65 s (min: 1.62 s, max: 1.73 s). In the worst case, we would receive a burst of traffic for ≈1.7 s. The amount of mirrored packets can be further reduced by rate-limiting the mirrored traffic directly on the switch; we expect this would *not* affect *Magnifier*'s performance, as a single mirrored packet is enough to invalidate a given sentinel.

Conclusion Our tests show that hardware switches can contain thousands to tens of thousands of mirroring rules, which is more than sufficient for *Magnifier*. Mirroring rules can be deactivated quickly (≈ 1.7 s), which limits the risk of bursts of mirrored traffic. The rule deployment is the most time-consuming operation (≈ 20 s for 2k rules). As a result, we can adjust the number of deployable mirroring rules (number of validated sentinels) by changing *Magnifier*'s iteration time.

6.3 Comparison with Everflow

We compare *Magnifier* with Everflow [45] which is a monitoring tool designed for debugging datacenter networks. Like *Magnifier*, Everflow randomly samples packets (using mirroring rules). In addition, it also mirrors *all* TCP SYN, FIN, and RST packets. As far as we know, the Everflow code is not available. Therefore, we reimplemented the relevant features and integrated them into our simulation framework (see § C).

Setup We use our simulation setup and the CAIDA dataset, 32 border routers, a sampling rate of 1/1024 (for both systems), and we vary the trace replay speed. We compare the performance of *Magnifier*, and Everflow on the `static` and `permuted` 5% traffic-to-ingress mappings.

Results Fig. 10 shows the per-packet coverage and mirrored traffic of both systems. We consider three different approaches: (i) "Everflow sampling only", where we rely only on Everflow's sampled packets to compute the ingress points; (ii) "Everflow sentinel", where we use the sentinel idea on top of Everflow's sampled packets; and (iii) "Magnifier unlimited" and "top 1k `activity`", where we report *Magnifier*'s coverage for all and the 1k most active sentinels.

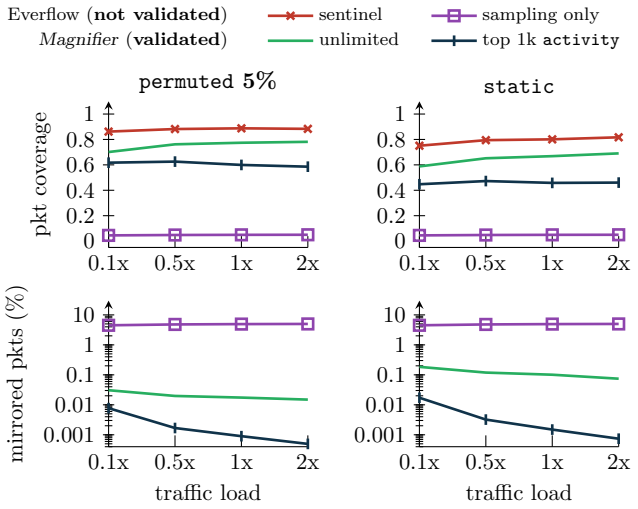


Figure 10: Comparison of coverage and mirrored traffic for *Magnifier* and Everflow under different traffic loads.

We first look at the coverage values (top plots in Fig. 10). Everflow's sentinel approach shows the best—although not validated—coverage values with up to 88% in the `permuted` 5% case. This is due to Everflow's sampled TCP flag packets. We do not reach 100% as traffic in some /24 prefixes is neither randomly sampled nor does it contain any TCP flags. These prefixes can invalidate found sentinels. Note again that the ground-truth data from the CAIDA trace allows us to compute these values. Everflow does not deploy any validation mirroring rules and does *not* know about the sentinel's validity. *Magnifier* follows closely with ≈80% (unlimited) and ≈60% (top 1k) coverage as we only have randomly sampled packets as input. Despite that, *Magnifier* manages to reach good coverage values, with validation from mirroring. Both systems' coverage values decrease in the more difficult `static` approach. For completeness, we also show Everflow's coverage if we only consider the sampled packets. This results in a poor packet coverage, although on a higher level than the "sampling only" line in Fig. 7 given that Everflow additionally also samples all TCP packets with SYN, FIN, and RST flags. These coverage values are constant between both assignment strategies as we observe the same TCP flag packets and roughly the same random samples.

Everflow's increased coverage has a high cost in the amount of mirrored traffic (lower plots in Fig. 10). Everflow generates the randomly sampled and TCP flag packets as mirrored traffic by design. *Magnifier* however, only generates targeted mirrored packets to validate found sentinels. If a sentinel is valid, it does not mirror any traffic. This is visible in the corresponding fraction of mirrored traffic.

Everflow constantly mirrors ≈5% of all traffic while *Magnifier* is more than one magnitude lower (≈0.1% of all traffic at real-time replay speed in the `static` case). This value decreases even further if we only consider the most active sentinels. We again observe that Everflow mirrors roughly the same amount of packets for both assignment strategies.

Conclusion Everflow yields better coverage but generates more mirrored traffic, which is more than one order of magnitude higher than *Magnifier*. Unlike Everflow, *Magnifier* validates the inferred ingress points, informing the controller as soon as a sentinel is no longer valid. In contrast, Everflow might need to wait a long time before receiving a mirrored packet indicative of an ingress point change, especially for long-running flows that do not often have TCP flags. In terms of mirroring rules, Everflow only needs around 20 of them [45]. *Magnifier* needs more mirroring rules but also uses them for *validation*—something that Everflow cannot achieve.

6.4 Sentinels in Tier-1 dataset

We now validate the practicality and benefits of sentinel-based monitoring by evaluating *Magnifier* on Tier-1 ISP data.

6.4.1 Existence of sentinels

Setup We divide our Tier-1 dataset into 30 s slices over which we compute sentinels and report the number of found ingress and egress sentinels. We only have sampling data available. Thus, we can only *approximate* the number of sentinels that would be found if *Magnifier* was deployed with mirroring.

Results We find a median of 145k egress sentinels and a median of 174k ingress sentinels. The lower and upper quartiles are within 1.4k around the median values in both cases.

We observe that we find more ingress than egress sentinels. This results from the typical forwarding behavior observed in an ISP: traffic from each of the ISP customers, which own specific prefixes, tends to enter via a single ingress point, which leads to a high number of ingress sentinels. At the same time, most ingress traffic goes to few popular destinations, which leads to few egress sentinels. We also see that the number of (ingress and egress) sentinels is stable over time, as shown by the small quartile ranges.

Conclusion We confirm that we find sentinels based on real sampling data from a Tier-1 ISP network. Furthermore, the number of sentinels is stable over time; this suggests that large changes in sentinel numbers can be used as a signal to detect various network events, which we discuss next.

6.4.2 Per-device sentinel changes

Setup We divide our Tier-1 dataset into 30 s slices over which we compute sentinels using *Magnifier*, focus on the number of sentinels found per border router, and search for large changes in the number of sentinels over consecutive slices.

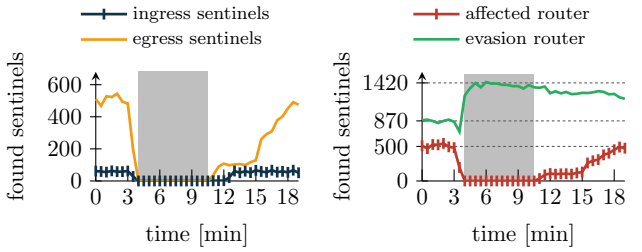
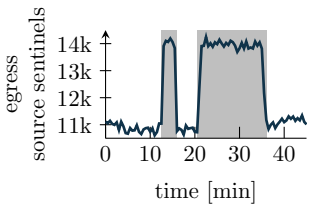


Figure 11: A temporary router outage (gray block) decreases the number of found sentinels (left) while we see similar increases on a close router (right).



Normal		DDoS-like	
prefix	flows	prefix	flows
P1	70	DDoS1	3700
P2	63	DDoS2	3560
P3	56	P1	72
P4	55	P2	65
P5	49	P4	47

Figure 12: A sudden burst of egress source sentinels (left) is likely to result from a DDoS-like event (right).

Results Fig. 11 shows the number of sentinels found following a single border router outage. As expected, *Magnifier* finds no more sentinels for the affected router. More interestingly, *Magnifier* also detects where the affected traffic was re-routed during the outage, as shown on the right: the number of egress sentinels of a geographically-close router increases shortly after and closely matches the number of lost sentinels.

Fig. 12 (left) shows a router with a burst of *egress source* sentinels (traffic from a given subnet exiting via a unique egress point) while no other router shows a matching decrease. Thus, we observe a sudden burst of packets from "new" source IPs towards a few destinations (table, right), indicating a possible Distributed Denial of Service (DDoS) attack. During this event, the egress traffic volume increased by less than 8%, which is less pronounced than the clear increase in sentinels. *Magnifier* also identifies the ingress of more than 75% of the "attack" flows via their ingress sentinels. Existing volumetric DDoS detection systems could use this information to block the DDoS traffic at the network ingress.

Conclusion Changes in the number of found sentinels reveal interesting network events. Operators could analyze the collected sampling data this way, even if they do not have the resources to deploy mirroring. With mirroring, *Magnifier* detects such changes in sub-seconds, long before similar events are visible in sampled flow data or SNMP counters.

7 Related work

Sampling-based network monitoring Many systems use NetFlow [12], J-Flow [27], sFlow [30] or related flow extraction tools for network measurements. Sampling suffers from a fundamental trade-off between coverage and accuracy. For

example, Teixeira et al. [37] use NetFlow data to detect egress changes due to BGP hot-potato routing but are limited by the collected ten-minute bins. In addition, Cunha et al. [13] uncover measurement artifacts in two J-Flow implementations.

Consequently, several works aim to improve sampling by optimizing the collection process. Estan et al. [14] propose router software updates to dynamically adapt the NetFlow sampling rate depending on the available traffic and memory amount. FlowRadar [21] uses flow sets to count flow observations in multiple array cells, then combines and decodes these counters centrally. Similarly, Flowyager [34] introduces Flowtrees, an efficient data structure to store flow information. These approaches are all limited by the sampling information. A key difference of *Magnifier* is to further improve the network visibility by leveraging mirrored traffic.

Mirroring-based systems Several monitoring systems use mirroring, which provides accurate visibility over a subset of the traffic, flows, or devices. Stroboscope [39] supports query-based monitoring under a strict budget of mirrored traffic. Everflow [45] provides the possibility to mirror some packets of *every* flow, *e.g.*, by mirroring packets with special TCP flags or debug header bits. Planck [33] takes a radical approach and mirrors all traffic over a single router port, which provides detailed insights but can also overload the network devices. Mirroring has also been used for troubleshooting [41], SDN monitoring [1], on in-network analysis [38, 42].

Mirroring suffers from three problems: (i) the flows of interest must be known in advance; (ii) it is limited by the routers' mirroring capacity, and (iii) it generates a potentially high volume of traffic. *Magnifier* mitigates these problems by leveraging sampling to derive the mirroring rules to deploy and uses negative mirroring to limit the traffic overhead.

In-network monitoring There has been many recent proposals for performing in-network monitoring based on in-band telemetry (*e.g.*, [2, 18, 25, 26, 31]) or sketches (*e.g.*, [5, 19, 22, 43, 44]). Both approaches boil down to implementing highly efficient data structures to gather traffic statistics, *e.g.*, packet counts. The main limitation is that these approaches depend on software-defined or P4-programmable hardware, which is not commonly deployed in ISP networks nowadays. Moreover, these approaches provide precise information, but over specific queries only; setting and collecting counters to track ingress and egress points of an arbitrarily large number of IP prefixes is hard to scale. Negative mirroring addresses this: while *Magnifier*'s inferences are correct, there is no traffic nor compute overhead—only TCAM usage. Packets that do get mirrored provide exact information—*i.e.*, source and destination IP—which allows for quick and precise reactions.

Detection of ingress/egress *Magnifier* is designed to detect traffic ingress/egress points, which has been previously studied: Feldmann et al. [15] provide foundation work for detecting different flow types in ISP networks as well as the ingress and egress of observed flows. To achieve good results, they need per-flow measurements on the ingress and up-to-date forwarding tables of the routers in the network, which are both costly to obtain. Mahajan et al. [23] use algorithms similar to our sentinel idea to build so-called "aggregates", a collection of packets with a common property, to free congested links. However, it is unclear how they extract the traffic to build the aggregates or validate their assumptions. Peng et al. [29] run a change point detection algorithm to detect changes in the number of new IP addresses, which is a good metric to detect (the ingress) of DDoS attacks. Most of these systems lack the global ingress/egress view that *Magnifier* provides.

Traffic matrix estimation Soule et al. [36] compare different techniques based on bias and variance properties. They show that direct measurements are required to reduce bias, which is an expensive process. Papagiannaki et al. [28] observe that the node fanout, *e.g.*, how traffic from an ingress is distributed towards different egresses, is stable over time. *Magnifier* confirms and leverages this behavior: sentinels are stable over time, which creates a valuable monitoring signal (§ 6.4). With mirroring, *Magnifier* also quickly detects changes and updates its traffic matrix estimation. OpenTM [40] uses a different approach, based on active polling of every source-destination pair, which is very precise but does not scale to large networks. Malboubi et al. [24] addresses the special case of SDN networks, which limits the system's applicability. Pingmesh [17] frequently generates pings to compute latency matrices. By contrast, *Magnifier* does not require active measurements and runs on traditional routers, which makes it easy to deploy.

Monitoring frameworks Several monitoring frameworks support rich sets of queries, *e.g.*, [20, 32, 42]. In particular, Flowyager [34] is similar to *Magnifier* as it builds primarily on sampling. The downside of these frameworks is their complexity and extensive storage and computational resource requirements. By contrast, *Magnifier* focuses on performing ingress/egress monitoring with little overhead.

8 Conclusion

Precise observations of traffic ingress and egress points are difficult to generate in large ISP networks. In this paper, we show how *Magnifier* combines the global view of sparsely-sampled flow observations with precise, targeted information from mirrored traffic. *Magnifier* enhances observed flows with validated ingress and egress points and scales to the largest ISP networks while only generating a small amount of mirrored traffic. *Magnifier*'s outputs can also help monitor outages or detect volumetric DDoS attacks.

Acknowledgments

We would like to thank Paul Stark and Derk-Jan Valenkamp for their great help with the hardware-based evaluation. Many thanks as well to Tibor Schneider for his support with TikZ.

References

[1] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Designing Heavy-Hitter Detection Algorithms for Programmable Switches. *IEEE/ACM Transactions on Networking*, 2020. doi:10.1109/TNET.2020.2982739.

[2] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic In-band Network Telemetry. SIGCOMM'20. ACM, 2020. doi:10.1145/3387514.3405894.

[3] Tobias Bühler. Magnifier GitHub repository, 2022. https://github.com/nsg-ethz/Magnifier Accessed: 2022-08-01.

[4] CAIDA. The CAIDA UCSD anonymized internet traces 2018. https://www.caida.org/catalog/datasets/passive_dataset Accessed: 2022-08-01.

[5] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. SIGCOMM'20. ACM, 2020. doi:10.1145/3387514.3405865.

[6] Cisco Systems. Cisco Python API. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus3600/sw/93x/programmability/guide/b-cisco-nexus-3600-nx-os-programmability-guide-93x/m-3600-python-api-93x.pdf Accessed: 2022-08-01.

[7] Cisco Systems. Nexus 9000 TCAM Carving, 2016. https://www.cisco.com/c/en/us/support/docs/switches/nexus-9000-series-switches/119032-nexus9k-tcam-00.html Accessed: 2022-08-01.

[8] Cisco Systems. Cisco Nexus 7000, 2021. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/Data_Sheet_C78-437762.html Accessed: 2022-08-01.

[9] Cisco Systems. Configuring ERSPAN, 2021. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus7000/sw/system-management/guide/b_Cisco_Nexus_7000_Series_NX-OS_System_Management_Configuration_Guide/b_Cisco_Nexus_7000_Series_NX-OS_System_Management_Configuration_Guide_chapter_010101.html Accessed: 2022-08-01.

[10] Cisco Systems. Cisco Nexus 9300-FX, 2022. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/datasheet-c78-742284.html Accessed: 2022-08-01.

[11] Cisco Systems. Configuring Rate Limits, 2022. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus3000/sw/security/92x/b-cisco-nexus-3000-nx-os-security-configuration-guide-92x/b-cisco-nexus-3000-nx-os-security-configuration-guide-92x_chapter_010000.html Accessed: 2022-08-01.

[12] Benoit Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), 2004. https://datatracker.ietf.org/doc/html/rfc3954 Accessed: 2022-08-01.

[13] Ítalo Cunha, Fernando Silveira, Ricardo Oliveira, Renata Teixeira, and Christophe Diot. Uncovering Artifacts of Flow Measurement Tools. In *International Conference on Passive and Active Network Measurement*. Springer, 2009. doi:10.1007/978-3-642-00975-4_19.

[14] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a Better NetFlow. *ACM SIGCOMM CCR*, 2004. doi:10.1145/1030194.1015495.

[15] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. *ACM SIGCOMM CCR*, 2000. doi:10.1145/347059.347554.

[16] Fred Klassen. Tcpreplay - Pcap editing and replaying utilities, 2020. https://tcpreplay.appneta.com/ Accessed: 2022-08-01.

[17] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. SIGCOMM'15. ACM, 2015. doi:10.1145/2785956.2787496.

[18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. SIGCOMM'18. ACM, 2018. doi:10.1145/3230543.3230555.

[19] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. SIGCOMM'17. ACM, 2017. doi:10.1145/3098822.3098831.

[20] Kentik. Network Observability, Performance and Security, 2022. https://www.kentik.com/ Accessed: 2022-08-01.

[21] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *NSDI'16*. USENIX Association, 2016. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-yuliang Accessed: 2022-08-01.

[22] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. SIGCOMM'16. ACM, 2016. doi:10.1145/2934872.2934906.

[23] Ratul Mahajan, Steven M Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM SIGCOMM CCR*, 2002. doi:10.1145/571697.571724.

[24] Mehdi Malboubi, Shu-Ming Peng, Puneet Sharma, and Chen-Nee Chuah. A Learning-Based Measurement Framework for Traffic Matrix Inference in Software Defined Networks. *Computers & Electrical Engineering*, 2018. doi:10.1016/j.compeleceng.2017.11.020.

[25] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-Defined Measurement. SIGCOMM'14. ACM, 2014. doi:10.1145/2619239.2626291.

[26] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. SIGCOMM'17. ACM, 2017. doi:10.1145/3098822.3098829.

[27] Juniper Networks. Configure J-Flow, 2017. https://supportportal.juniper.net/s/article/SRX-Getting-Started-Configure-J-Flow?language=en_US Accessed: 2022-08-01.

[28] Konstantina Papagiannaki, Nina Taft, and Anukool Lakhina. A Distributed Approach to Measure IP Traffic Matrices. IMC'04. ACM, 2004. doi:10.1145/1028788.1028808.

[29] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Proactively Detecting Distributed Denial of Service Attacks Using Source IP Address Monitoring. In *International Conference on Research in Networking*. Springer, 2004. doi:10.1007/978-3-540-24693-0_63.

[30] P. Phaal, S. Panchen, and N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational), 2001. https://datatracker.ietf.org/doc/html/rfc3176 Accessed: 2022-08-01.

[31] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *NSDI'19*. USENIX Association, 2019. https://www.usenix.org/conference/nsdi19/presentation/pontarelli Accessed: 2022-08-01.

[32] The Zeek Project. The Zeek Network Security Monitor, 2020. https://zeek.org/ Accessed: 2022-08-01.

[33] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. SIGCOMM'14. ACM, 2014. doi:10.1145/2619239.2626310.

[34] Said Jawad Saidi, Aniss Maghsoudlou, Damien Foucard, Georgios Smaragdakis, Ingmar Poese, and Anja Feldmann. Exploring Network-Wide Flow Data With Flowyager. *IEEE Transactions on Network and Service Management*, 2020. doi:10.1109/TNSM.2020.3034278.

[35] Philip Smith, Rob Evans, and Mike Hughes. RIPE Routing Working Group Recommendations on Route Aggregation, 2006. https://www.ripe.net/publications/docs/ripe-399 Accessed: 2022-08-01.

[36] Augustin Soule, Anukool Lakhina, Nina Taft, Konstantina Papagiannaki, Kave Salamatian, Antonio Nucci, Mark Crovella, and Christophe Diot. Traffic Matrices: Balancing Measurements, Inference and Modeling. In *ACM SIGMETRICS Performance Evaluation Review*, 2005. doi:10.1145/1071690.1064259.

[37] Renata Teixeira, Aman Shaikh, Timothy G Griffin, and Jennifer Rexford. Impact of Hot-Potato Routing Changes in IP Networks. *IEEE/ACM Transactions On Networking*, 2008. doi:10.1109/TNET.2008.919333.

[38] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. PacketScope: Monitoring the Packet Lifecycle Inside a Switch. SOSR'20. ACM, 2020. doi:10.1145/3373360.3380838.

[39] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative Network Monitoring on a Budget. In *NSDI'18*. USENIX Association, 2018. https://www.usenix.org/conference/nsdi18/presentation/tilmans Accessed: 2022-08-01.

[40] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In *International Conference on Passive and Active Network Measurement*. Springer, 2010. doi:10.1007/978-3-642-12334-4_21.

[41] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *ATC'11*. USENIX Association, 2011. https://www.usenix.org/conference/usenixatc11/ofrewind-enabling-record-and-replay-troubleshooting-networks Accessed: 2022-08-01.

[42] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *NSDI'19*. USENIX Association, 2019. https://www.usenix.org/conference/nsdi19/presentation/yu Accessed: 2022-08-01.

[43] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI'13*. USENIX Association, 2013. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/yu Accessed: 2022-08-01.

[44] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets. In *NSDI'21*. USENIX Association, 2021. https://www.usenix.org/conference/nsdi21/presentation/zhao Accessed: 2022-08-01.

[45] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. SIGCOMM'15. ACM, 2015. doi:10.1145/2785956.2787483.

A CRediT statement

This section lists the author's contributions to this work. The contributions are described using CRediT, the Contributor Roles Taxonomy, an ANSI/NISO standard.

All authors agree with this declaration of contributions.

Tobias Bühler	0000-0002-8876-1546
Conceptualization	Equal
Data curation	Lead
Formal analysis	Lead
Investigation	Lead
Methodology	Lead
Project administration	Lead
Software	Lead
Validation	Equal
Visualization	Equal
Writing – original draft	Lead
Writing – review & editing	Supporting

Romain Jacob	0000-0002-2218-5750
Conceptualization	Supporting
Data curation	Supporting
Formal analysis	Supporting
Methodology	Supporting
Project administration	Supporting
Software	Supporting
Supervision	Equal
Validation	Equal
Visualization	Equal
Writing – original draft	Supporting
Writing – review & editing	Lead

Ingmar Poese	
Data curation	Supporting
Investigation	Supporting
Resources	Equal
Supervision	Supporting

Laurent Vanbever	0000-0003-1455-4381
Conceptualization	Equal
Funding acquisition	Lead
Investigation	Supporting
Methodology	Supporting
Project administration	Supporting
Resources	Equal
Supervision	Equal
Writing – original draft	Supporting
Writing – review & editing	Supporting

B *Magnifier*'s controller placement

Magnifier needs a central controller to build its network-wide ingress/egress view. As we heavily depend on sampled flow observations, it makes sense to co-locate *Magnifier* with the *e.g.*, already existing, central collector of the sampling data. In large ISP networks, with routers around the globe, we can deploy additional sub-controllers that start and stop the mirroring rules and collect mirrored packets. More precisely, the main controller is needed to compute new sentinels and builds the final ingress/egress observations. It delegates mirroring to the sub-controllers which autonomously handle the deployment, activation and deactivation of rules while reporting back any mirroring-based observations.

C Everflow implementation

Following a few more details to our Everflow reimplementation in our simulation setup.

Everflow uses packet mirroring to produce its random packet samples. The paper [45] explains that Everflow mirrors based on a fixed number of bits in the IP identification header field (IPID). As an example, selecting 10 random bits in the IPID field will result in random packet sampling of 1 out of $2^{10} = 1024$ packets. However, this assumption is only true if the values in the IPID fields are more-or-less uniformly distributed. Taking our CAIDA trace as an example, we see that we have a huge number of packets which set the IPID field to zero. Depending on how we select the bits in the IPID field, we might get way more or less sampled packets than expected. For this reason, we implemented the random packet sampling aspect of Everflow in our simulation code by taking every n-th packet observed on a device, *e.g.*, every 1024th packet in the previous example.

Additional to the implemented mirroring techniques (random packet sampling and TCP flag packets), Everflow also supports mirroring of packets with a special debug bit. As this was not relevant for a direct comparison with *Magnifier*, we did not implement this feature in our simulation code. The same holds for Everflow's controller, storage and reshuffler components.

D Additional evaluation results

This appendix section first illustrates the lab setup used to evaluate *Magnifier*. We then analyze the used CAIDA trace in more detail. Afterward, we show additional evaluation results focused on *Magnifier*'s performance. We conclude with additional plots comparing *Magnifier* with Everflow.

D.1 *Magnifier* lab setup

Fig. 13 illustates the lab setup we used to evaluate *Magnifier*. We establish four parallel connections between the two Nexus

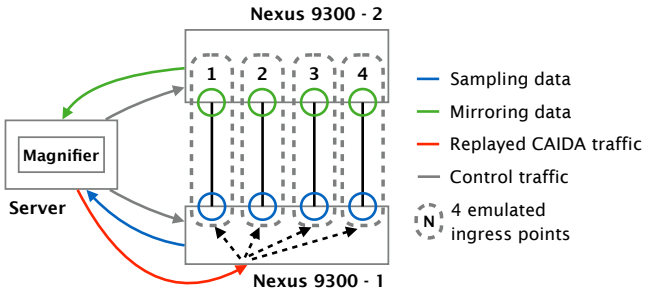


Figure 13: Two Nexus 9300 switches emulate four network ingress points. The traffic is replayed and sampled on the first switch, then forwarded to the second, which mirrors packets.

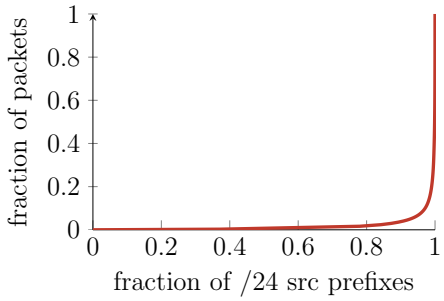


Figure 14: A CDF plot of the amount of packets observed per source /24 prefix in 60 s (one iteration at real speed) in our CAIDA trace.

9300 switches, each emulating a network ingress. The first switch receives and samples the traffic using sFlow (sampling rate 1/4096[7]). It then forwards to the second switch, which mirrors the traffic according to the configured rules. *Magnifier*'s controller runs on a server and collects sampling and mirroring data. As these switches are limited to 512 mirroring rules, we used a fixed budget of 500 rules per emulated ingress point.[8]

D.2 CAIDA data analysis

Fig. 14 shows a CDF of the amount of packets observed per source /24 in 60 s of our CAIDA trace used in the evaluation. 60 s represent one iteration at real-time replay speed. As we can see we have a very small number of heavy hitters which carry most traffic as well as a huge number of /24 prefixes which only contain a few packets. Roughly 10% of all /24 prefixes contain more than 90% of all the packets.

A lot of the /24 prefixes with very low packet counts are most likely DDoS attack traffic (e.g., TCP SYN packets). We decided to keep these packets in the trace as a real ISP network could also observe similar packet distributions in their transit traffic.

D.3 Additional Magnifier plots

This section contains additional plots which evaluate *Magnifier* in terms of packet coverage and mirrored traffic.

Fig. 15 shows the performance results if we consider an increasing number of border routers (from 4 to 64). For `random` and `static` traffic assignment we notice that the coverage slightly drops while we see an increased amount of mirrored traffic. However, this is not true for the `permuted` assignment strategies. `random` and `static` distribute the packets to their ingress points based on equal slices of the *destination* IP space. If we have more border routers, we also have additional slices

[7]The highest configurable rate on this model; we get the *most* samples.

[8]The four emulated ingresses share the budget. We use TCAM carving [7] to increase the space for our mirroring rules to 2048 (by taking it from other features), to enable the original budget (512) per ingress.

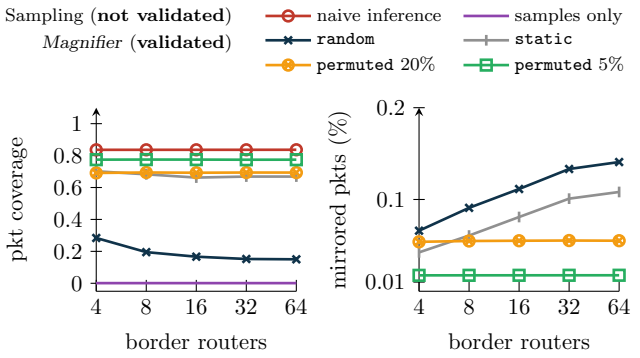


Figure 15: Simulation results for coverage and mirrored traffic when *Magnifier* runs with different amounts of border routers. CAIDA traces replayed at real speed, sampling rate 1/1024.

and close IP space is distributed over multiple ingresses which leads to the observed drop in coverage. This is not true for the `permuted` cases, where we always permute a fixed number of source /24 prefixes to different ingresses.

Fig. 16 considers different sampling rates. As expected, if we have fewer samples as input *Magnifier* can cover fewer packets and also produces fewer mirrored packets as it finds fewer sentinels to begin with. We observe this behavior for all traffic assignments.

Finally, Fig. 17 shows the missing assignment strategies (`random` and `permuted` 20%) if we consider different amounts of top sentinels (`activity` and `size` ordering). Following the results in Fig. 8, the `activity` strategy provides better coverage than `size` and a lower amount of mirrored traffic.

D.4 Stability of sentinels

Following, we evaluate how many sentinels change between simulation iterations.

Setup We use the results from our simulations with 32 border routers, real traffic speed and various traffic-to-ingress assignments (sampling rate 1/1024). The results show mean values over 60 iterations.

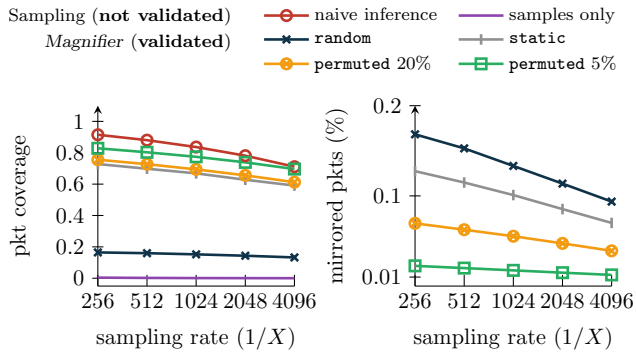


Figure 16: Simulation results for coverage and mirrored traffic when *Magnifier* runs with different sampling rates. CAIDA trace replayed at real speed, 32 border routers.

Results Table 3 shows the amount of changed sentinels for different amounts of deployed sentinels (based on `activity` and `size` ordering) for `random`, `static` and `permuted` 5% traffic assignment. We first observe that we have to change fewer sentinels if we base the ordering on sentinel activity. More active sentinels are often also stable over longer periods of time which means that we find them consistently. We see a different behavior for the ordering based on size. Here nearly all sentinels change between iterations. The largest sentinels are often based on sparse samples located in empty prefix space. That means, we might not be able to find the same big sentinel between multiple iterations if the covered flows are no longer visible (e.g., in the sampled data).

As expected, the number of changed sentinels also depends on the difficulty of the traffic assignment. In the `random` case, ingress assignment changes frequently even during a single iteration. That means we often find new sentinels in the following iteration. For `permuted` 5%, the assignment is much more stable and we can always keep around 50% of all sentinels between iterations.

	# sentinels	`random`	`static`	`permuted` 5%
`activity` ordering	Top 100	84	50	35
	Top 500	406	292	220
	Top 1000	836	610	466
	Top 5000	4487	3662	2769
`size` ordering	Top 100	94	87	45
	Top 500	472	453	224
	Top 1000	950	918	461
	Top 5000	4817	4698	2776

Table 3: Number of changed sentinels between iterations for different assignment and sentinel ordering strategies.

Conclusion The top sentinels often change between iterations, however *Magnifier* is not really impacted by that. As we describe in § 4.1, *Magnifier* works with two ACLs and switches between them. While one is active, the other one gets populated.

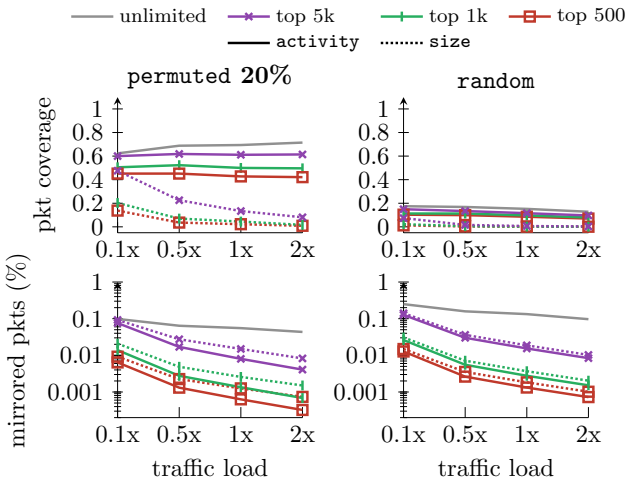


Figure 17: Simulation results for coverage and mirrored using different amount of top sentinels according to a `activity` and `size` ordering. We show the `random` and `permuted` 20% traffic assignment strategies (compare with Fig. 8). The plots show values for different traffic replay speeds of the CAIDA trace with 32 border routers and sampling rate of 1/1024.

The frequent sentinel changes between iterations are therefore not a big problem as we anyway need to build a completely new ACL.

D.5 Additional comparison with Everflow

In this section we show additional comparison plots between *Magnifier* and Everflow. Fig. 18 shows different number of ingress routers while Fig. 19 considers varying sampling rates. For both figures we show the results for `permuted` 5% and `static` traffic assignments. Everflow's packet coverage *and* amount of mirrored packets show only small reactions to the different ingress routers and/or sampling rates. Everflow's mirrored packets mainly contain packets due to TCP SYN, FIN or RST flags. The randomly sampled ones contribute only in a small amount. As a result, changes in the sampling rate (Fig. 19) have more impact on *Magnifier* than on Everflow. *Magnifier*'s performance is tightly related to the amount and distribution of the randomly sampled packets.

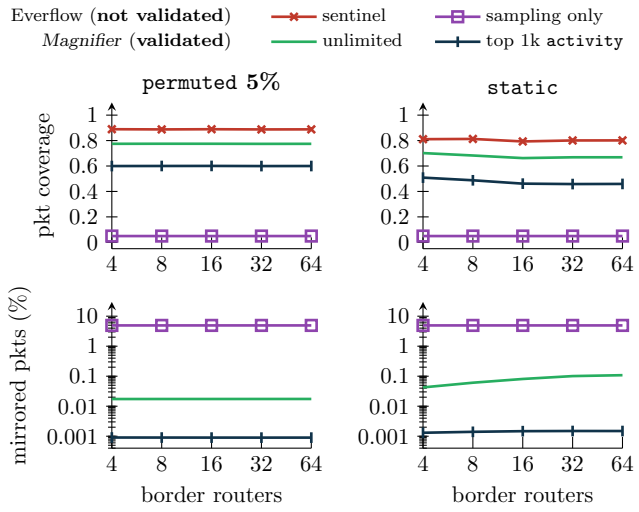


Figure 18: Comparison of coverage and mirrored traffic for *Magnifier* and Everflow for different amounts of border routers. We show the `static` and `permuted` 5% traffic assignment. We replay the CAIDA trace at real speed and use a sampling rate of 1/1024.

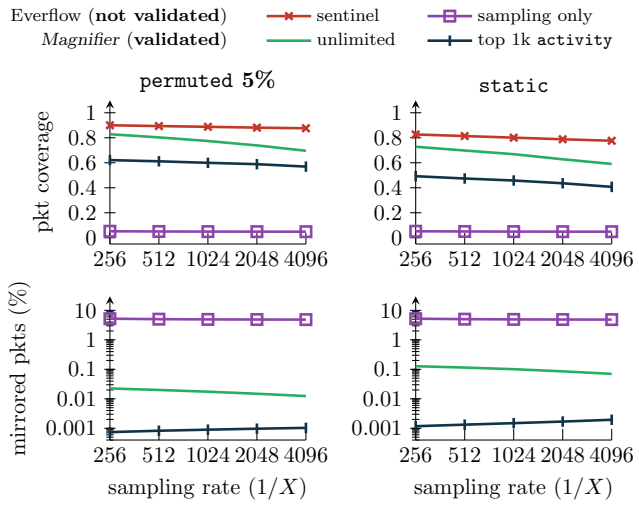


Figure 19: Comparison of coverage and mirrored traffic for *Magnifier* and Everflow for different sampling rates. We show the `static` and `permuted` 5% traffic assignment. We replay the CAIDA trace at real speed and distribute traffic over 32 simulated ingresses.

NetPanel: Traffic Measurement of Exchange Online Service

Yu Chen[†], Liqun Li[◇], Yu Kang[◇], Boyang Zheng[†], Yehan Wang[†], More Zhou[†]
Yuchao Dai[†], Zhenguo Yang[†], Brad Rutkowski[‡], Jeff Mealiffe[‡], Qingwei Lin[◇]
[†]*Microsoft 365, China,* [◇]*Microsoft Research, China,* [‡]*Microsoft 365, USA*

Abstract

Global cloud applications are composed of thousands of components. These components are constantly generating large volumes of network traffic, which is a major cost of cloud applications. Identifying the traffic contributors is a critical step before reducing the traffic cost. However, this is challenging because the measurement has to be component-level, cost-effective, and under strict resource restrictions. In this paper, we introduce NetPanel, which is a traffic measurement platform for the Exchange Online (EXO) service of Microsoft. NetPanel fuses three data sources, namely IPFIX, Event Tracing for Windows (ETW), and application logs, to jointly measure the service traffic at the component level, where each component is owned by a service team. NetPanel uses several schemes to reduce the measurement overhead.

NetPanel has been in operation for more than one year. It has been used to profile network traffic characteristics and traffic cost composition of EXO. With the insights obtained through NetPanel, we have saved millions of dollars in network resources. The overhead of running NetPanel is relatively small, which requires less than 1% CPU and disk I/O on production servers and less than 0.01% of EXO computation cores to process the data in our big-data platform.

1 Introduction

Cloud applications, such as Exchange Online (EXO), are composed of thousands of components running on hundreds of thousands of servers, developed and maintained by engineers from many different teams. In EXO, one component is a module that performs a specific function, as an entire or part of a process. The Internet Information Services (IIS) [4] based proxy, running on frontend (FE) servers, routes traffic for different components such as REST [23], EWS [5] and MAPI [6]. The traffic of each component is owned by a specific engineering team. These components are sending tremendous traffic across data centers, which incurs great costs. Any defect in a single component may lead to widespread traffic flood. Furthermore, due to the massive number of components, the limited shared bandwidth could be easily drained by low-priority traffic. In these cases, customers could suffer from long latency or even connection loss [7, 9, 12, 14]. For example, an incident caused by anomalous traffic was reported by Azure [2] on June 14th, 2021 where some customers received errors when performing service management operations. The root cause was high CPU consumption and request timeouts caused by an unexpected surge in internal traffic. The issue was mitigated by adding rules to block internal traffic on a subset of backend servers.

Cloud application owners have built plenty of monitors for incidents and performance regressions [1, 3, 20]. Such monitors are typically based on availability or latency metrics, which are insensitive to traffic issues. Therefore, there is still undesired traffic caused by various reasons, such as code bugs or misconfigurations. Over time, these hidden bugs become extremely difficult to trace as everyone takes them as necessary bandwidth requirements. Unnecessarily more capacity planning budget is therefore needed in subsequent years. The extra cost will be millions of dollars per year given the application scale. For example, in one case, we caught a configuration error that nearly quadrupled its peak traffic. In another case, we found that one service was sending requests globally, while these requests can actually be handled within a location. These cases will be detailed in Section 5.2. Although there is existing work to detect anonymous traffic bursts [29, 31, 39, 44], few have provided insights for continuous traffic optimization. There is a body of work on misconfiguration detection [51, 53] and safe deployment [34], but their solutions are not specially designed for traffic-related issues. In particular, some traffic issues can only be identified through long-term continuous monitoring.

While it is vital to continuously monitor the traffic flow for a cloud application, the dynamic and heterogeneous nature of a global cloud application makes this difficult. An efficient traffic monitoring system for cloud applications must satisfy the following requirements:

(1) The measurement should provide component-level

results. To efficiently identify the owner of the traffic, component-level measurement is required. A component is typically owned by a single engineering team. Researchers have been exploiting network measurement tools such as IPFIX/Netflow in the last decade [22, 37, 55] to gain insights into the network traffic in large-scale cloud infrastructures. These approaches operate on network routers, so they cannot identify components at the application layer. Server network analysis tools [8, 11, 36, 43] can observe the traffic sent by each process. However, multiple components can share a single process. These tools cannot distinguish the traffic emitted by components sharing the same process.

(2) The size of daily measurement results should be small (in GB). To draw an effective conclusion on traffic observation, engineers need to query data over a long period of time. In addition, there are cases where successive interactive analysis is needed, such as the case we discuss in Section 5.2.1. A user query response should be returned within a few seconds. On the other hand, global cloud applications are constantly generating a vast amount of traffic logs. For instance, the size of IPFIX data is more than 10TB per day. Event Tracing for Windows (ETW) [11] data and application logs are in PB. Directly joining PB and TB data at a daily frequency is impractical due to resource constraints. Directly running queries on these log data imposes huge data processing costs and unacceptable query latency.

(3) The collector in production environment should run under strict resource restrictions. Cloud applications such as EXO provides high Service Level Agreements (SLA) [13] to the customers. Therefore, the service has strict restrictions in terms of the resource used by any single component on the servers to ensure a quick response to customer requests. To collect event logs, ETW needs to run on the servers by the side of service components. Pulling all the network metrics from ETW regularly is prohibited in the production environment because it will exhaust the CPU and disk I/O on the production servers.

To address the aforementioned challenges, we design NetPanel, a cost-effective continuous traffic profiling tool for EXO. NetPanel takes three data sources, including IPFIX, ETW, and application logs. We fuse these data sources to jointly provide component-level measurement results. We reduce the data size with feature translation, data splitting, and data aggregation to keep all measurement results at several GB per day, which will be detailed in Section 4. To reduce the resource consumption of ETW, we only retain the data for the top k ports obtained from IPFIX.

NetPanel has been safeguarding the network traffic of EXO for more than 1 year. It brought us valuable insights into our traffic and helped us save millions of dollars per year. We introduce 4 real-world cases in Section 5.2. NetPanel runs with negligible impact on our production servers (less than 1% increase in CPU and disk IO). The data processing cost for our big-data platform is also minor given the scale of EXO (less than 0.01% of EXO cores). For a query for data in a 60-day period, the response can be returned within 30 seconds.

Roles	Abbr.	Functionality
Frontend	FE	Connects with customers and routes customer requests
Backend	BE	Stores mailboxes, delivers emails, and provides site resilience
Active Directory	AD	Holds and queries customer metadata

Table 1: Server roles and their functionalities.

Our key contributions and insights in this work are summarized as follows:

- We discuss the requirements and challenges of measuring the traffic for a global scale application, i.e., EXO. We show that telemetry data should be attributed to an organizational structure, such as a team of engineers, to actually drive cost reduction. Moreover, daily data size should be small enough to provide insight into how traffic data changes over time.
- We present our novel traffic measurement design which fuses IPFIX, ETW, and application logs to achieve component-level measurement. We demonstrate that, with proper data volume reduction, it is feasible to join data sources across routers and servers. We show that cross-validating data for integrity is feasible and crucial.
- We share our observations on traffic characteristics of EXO in production environment. Specifically, we figure out that heavy hitters (top ports/components) are stable in EXO, and this feature can be used to reduce data volume. We also demonstrate how NetPanel can help reduce traffic costs through real-world case studies.

We believe that the experience of operating NetPanel provides valuable guidance to other cloud applications on how to monitor and optimize their network traffic.

2 Background

This section introduces the EXO service traffic and explains how these traffic flows are generated. Then, we share the measurement tools available in EXO and their capabilities.

2.1 EXO Service Traffic

EXO operates in numerous datacenters around the world. There are hundreds of thousands of servers all over the world serving its enormous user community. The servers are categorized into three server roles: frontend routing proxy (**FE**), backend mailbox (**BE**), and directory (**AD**), as summarized in Table 1. FE servers, which sit behind load-balancers, serve customer requests over direct connections. AD servers hold information about users, mailboxes, and other customer metadata. BE servers provide storage for mailboxes and are responsible for the delivery of emails to/from mailboxes. Multiple

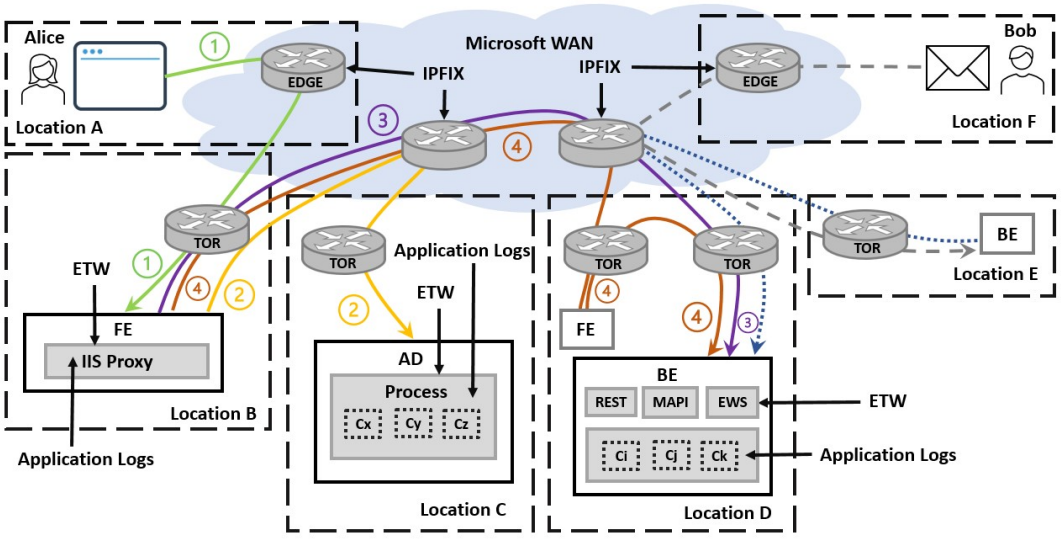


Figure 1: EXO traffic overview. When Bob sends an email to Alice, replication occurs, and Alice later reads the email replica. During the email-reading process, 6 Internal Long-haul traffic flows and 2 Internet traffic flows are generated.

copies of mailboxes are geographically-dispersed to provide high availability and site resilience. Each role of servers (i.e., FE, AD, or BE) hosts a group of services in order to provide functionalities as designed.

EXO servers communicate with each other when serving customers. We divide EXO traffic into two types: *WAN* traffic and *Metro* traffic. The WAN traffic goes through routers in WAN; and the rest, namely Metro traffic, travels within data-centers in the same location. We are particularly interested in the WAN traffic because it costs more than 90% of the annual bill. There are two subtypes of WAN traffic. The first is the traffic between EXO servers and user clients (Internet traffic) and the other among EXO servers (Internal Long-haul traffic). Internet traffic is responsible for around 10% of WAN traffic and Internal Long-haul traffic takes the rest.

The Internal Long-haul traffic are assigned to different priority tiers, *Tier 0* and *Tier 1*, by Bandwidth Broker [52]. Tier 0 is of higher priority with higher Quality of Service (QoS). Tier 1 traffic has a lower priority, and it is routed through sub-optimal paths and is dropped by routers first when congestion occurs.

2.2 How EXO Traffic Generated?

We use a typical scenario shown in Figure 1 to describe how the traffic is generated in EXO. In this scenario, Bob uses the Desktop client to send an email to another user Alice. Alice then reads her email through a web client. Bob in Location F sends the email to Alice's mailbox in Location E. The mailbox is replicated to Location D for high availability. These are annotated by the gray dashed line and the blue dotted line in Figure 1. Later, Alice reads her email from the mailbox. There are 4 steps in this email-reading process, detailed as follows:

Step 1: Alice uses the web client in Location A to send a request to the closest FE server. In this example, we assume the closest FE server resides in another Location B.

Step 2: The FE server talks to an AD server in Location C to query which BE server knows where the active copy of Alice's mailbox is hosted. In this case, the BE server in Location D is returned by the AD server.

Step 3: The FE server queries the BE server in Location D to ask which BE server hosts Alice's mailbox. In this example, the BE server in Location D happens to host Alice's mailbox, so it responds with itself.

Step 4: The FE server in Location B forwards the request to another FE server in Location D and that FE server in Location D will further transfer the request to the BE server in Location D. The mailbox's response is returned to Alice in the opposite direction along the paths of Step 1 and Step 4.

During the email-reading process, the traffic in Step 1 and the traffic from FE to the web client in the mailbox's response is Internet traffic because the FE server is talking to an external client outside of Microsoft. The traffic in Steps 2 and 3, and the traffic from the FE in Location B to the FE in Location D in Step 4 is Tier 0 traffic because the source and destination EXO servers are in different locations. In Step 4 (the orange arrow in Figure 1), the traffic between FE servers is Tier 0 traffic, while the traffic between FE and BE servers in Location D is Metro traffic. We use the example shown in Figure 1 only to explain the generation of Tier 0 traffic. Most

of the traffic would be Metro traffic when the servers involved are in the same location.

The replication traffic, represented as the blue dotted line in Figure 1, is Tier 1 traffic. Tier 1 traffic mostly consists of background traffic that does not serve user activities and therefore does not have rigorous latency requirements. Typical cases include: (1) data replications for high availability; (2) non-urgent mailbox migrations; (3) uploading logs.

In the example in Figure 1, Alice's request first hits the FE server in Location B (step 1), then is routed to the FE server in Location D (step 4), and finally reaches the BE server in Location D (step 4). The request is served by the REST [23] component on the BE server. In this situation, only the application logs of the FE servers capture this long-haul traffic between the two FE servers (Location B $\Rightarrow$ D). The components on BE servers in Location D, such as REST [23], MAPI [6], and EWS [5], are behind the FE proxy in the same location. As a result, the BE servers are unaware if the request originates from another location. Sometimes, multiple components on one BE server may even share the same process. Therefore, server network analysis tools [8, 11, 36, 43] are insufficient to provide component-level traffic measurements. In summary, application logs are necessary to perform component-level traffic measurements.

2.3 Traffic Measurements

There are two kinds of traffic measurement methods available in our data centers: *on-router* and *on-server* measurements.

One of the commonly used on-router measurements is flow monitoring [26, 35, 41]. There are two standards, i.e., Netflow [21] and IPFIX [15], that have been used for years. Flow monitoring samples packets with a certain probability and aggregate them into flows. A flow is a sequence of packets with the same IP 5 tuples (*src./dst. addresses*, *src./dst. ports*, and *protocol*). Flows are uploaded to a centralized storage.

When measurements are made on the servers, common toolkits include Tcpdump [8] and Event Tracing for Windows (ETW) [11]. Tcpdump is a well-known library that provides powerful packet analysis capabilities on Linux, while Windows uses ETW to collect system network events. These tools can monitor the traffic usage of all processes on a machine.

We annotate these measurement schemes in Figure 1. IPFIX collects traffic data on WAN routers. ETW collects system network events and provides traffic statistics for processes on servers. We further add application logs, which are generated within the services and are owned by different teams for debugging purposes. Application logs are request-focused. For a specific request, application logs record the timestamp, the component that serves the request, the local server name, the remote server name, the latency, the request and response content size, the remote port, etc.

We summarize the measurement capabilities of the three schemes in Table 2. TimeStamp, IP, Port, Process, and Traffic Size are general definitions. TimeStamp, an IP pair (source and destination), and a Port pair identify a unique flow. Process is the information of processes that are sending and receiving the traffic. We need a Differentiated Services Code Point (DSCP) tag [18] because Bandwidth Broker uses it for traffic QoS classification. Packets with different DSCP tags are classified into different priority tiers. IPFIX covers IP, port, and DSCP but cannot cover the process and component information which are available only on servers. ETW can further measure processes, but cannot cover the exact component as discussed in our example in Sec. 2.2. Application logs contain the request and response content sizes of components but not counting the sizes of the packet headers. In addition, application logs do not capture packet loss or retransmission. In conclusion, we need to fuse IPFIX, ETW, and application logs to achieve component-level measurement.

	Timestamp	IP	Port	DSCP	Process	Component	Traffic Size	Request Size
IPFIX	✓	✓	✓	✓	✗	✗	✓	✗
ETW	✓	✓	✓	✗	✓	✗	✓	✗
App logs	✓	✓	✓	✗	✓	✓	✗	✓

Table 2: Available Measurement Methods

3 Motivation and Design Goals

In this section, we state our motivation to design NetPanel and further define the goals to be met for our design along with the challenges to be resolved.

3.1 Motivation

In the EXO service, many components are working together to serve customers. These components send large amounts of traffic globally, which is very expensive. The large cost has motivated the application owner to understand the current traffic, reduce the traffic cost, and ensure there is no traffic waste. Furthermore, when a development team adds a new feature to reduce their traffic, they also need a tool to validate the traffic change of their component. Before NetPanel, each team only monitors their own request amount, leading to an isolated and incomplete view, which makes it hard to motivate traffic optimization efforts, verify data correctness, and detect traffic-related issues. On the other hand, without component-level information, it is non-actionable even if anomalous traffic is detected. With the increasing complexity of modern global-scaled software, this requirement becomes more and more urgent, which motivates us to build NetPanel.

3.2 Design Goals and Challenges

The design of NetPanel has to meet the following goals and address the corresponding challenges.

Goal-1: The measurement should provide component-level results. The overall EXO traffic should be divided into

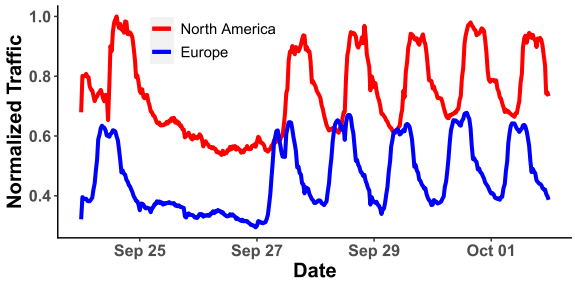
various components. EXO runs on hundreds of thousands of machines of three server roles. Machines of the same server role hold the same set of components. A component is owned by an engineering team. As long as the component's traffic is identified, the engineering team can take action to optimize its cost. NetPanel should provide the capability to establish the mapping between the components and their traffic flow.

Recent measurement works [33,45,46,48,54] are on-router measurements, so they cannot support measurement at component granularity. Traffic Refinery [19] identifies the component flows by inspecting DNS queries and manually specifying matches between components and their IP prefixes. This does not work in EXO because it assumes that each IP must correspond to at most one component. However, in our case, an IP can be shared by many components at the same time. Google uses Bandwidth Enforcer (BwE) [32] to allocate bandwidth at task granularity. A task may contain multiple processes, and therefore, BwE cannot be used to monitor the bandwidth for components sharing a process. NetPanel addresses this challenge by leveraging application logs to fill in the component property. We will describe how to jointly consider the three data sources, i.e., IPFIX, ETW, and application logs, to recover component-level traffic throughout Section 4.

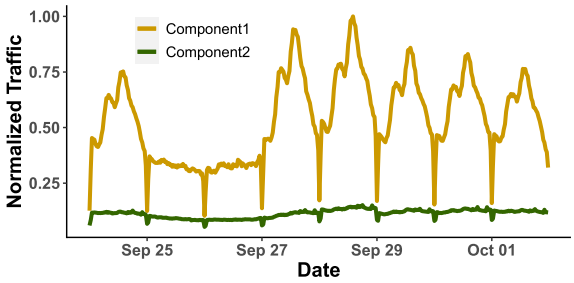
Goal-2: The daily measurement result size should be small (in GB). In EXO, IPFIX generates several TB of data every day, while the daily application log in PB. After compression, the data size will be reduced to ∼10%. However, IPFIX data is more than 10TB per day, so the data size will still be too large after compression. To draw an effective conclusion on traffic, engineers usually have to do consecutive analysis over long time intervals and compare the results. The system should provide quick responses to user queries and consume few resources. From our experience, we need to limit the result size to several GB per day, so that analysis over a long time interval is allowed.

The data used for analysis should cover a continuous time interval of at least several weeks to overcome the dynamic nature of network traffic. Figure 2a shows the traffic variation in EXO over a time interval of more than one week in North America and Europe. The traffic volume highly aligns with user activities, with more traffic during working hours and much less at night and on weekends. The valley values are nearly half of the peak values. The large fluctuation rate is likely to override the traffic change introduced by a new feature if we make queries only over a short time interval of a few hours. Moreover, different components can have different traffic patterns. We show the traffic patterns of two components in Figure 2b. The traffic of Component 1 fluctuates greatly, while that of Component 2 is relatively stable.

Many approaches have been proposed to reduce the data size. However, they cannot satisfy our requirements for different reasons. IBM cloud [39] is dealing with a 2.35TB log each day, but their approach only reports anomalies without any details on current and historical traffic usage. Analysis



(a) All geographical regions have fluctuations between daytime and night but follow a similar weekly pattern. The valley values could be half of the peak values.



(b) Different components have various traffic patterns.

Figure 2: Huge variations in time and components domain.

farm [47] proposes a cloud log analysis platform and aggregates IP addresses to IP-groups. We achieve something similar with feature translation (detailed in Section 4.2.1), but this alone would only reduce the data size to hundreds of GB per day. Anwar et.al. [16] claim to reduce the data size by up to 80% using different sampling frequencies and storage aggregation for different metrics. In our case, we collect only one metric but the data size must be reduced thousands of times. NetPanel addresses the challenge with multiple steps, which will be introduced in Section 4.2.

Goal-3: The collector in the production environment has to run under strict resource restrictions. The data collector has to run continuously in the production environment. There are strong resource restrictions (CPU, memory, disk I/O, etc.) for measurement tools in order to reserve as many resources as possible to serve user requests. The resource consumption of the collectors has to be very small.

We abandoned pulling all network metrics from ETW in the EXO production environments because of performance issues. In EXO, every single component should use no more than 5% CPU. It is restricted to log no more than 32MB of data on local disks every five minutes. Running ETW and writing all the metrics to the local disk exceeds the limit as shown in Section 6.1. The ETW data size for a single day is in PB. NetPanel reduces ETW collectors' resource consumption by only recording the traffic data of the top k ports. This greatly reduces the log size as well as the computation resource and disk I/O throughput. We explain how the top k ports are

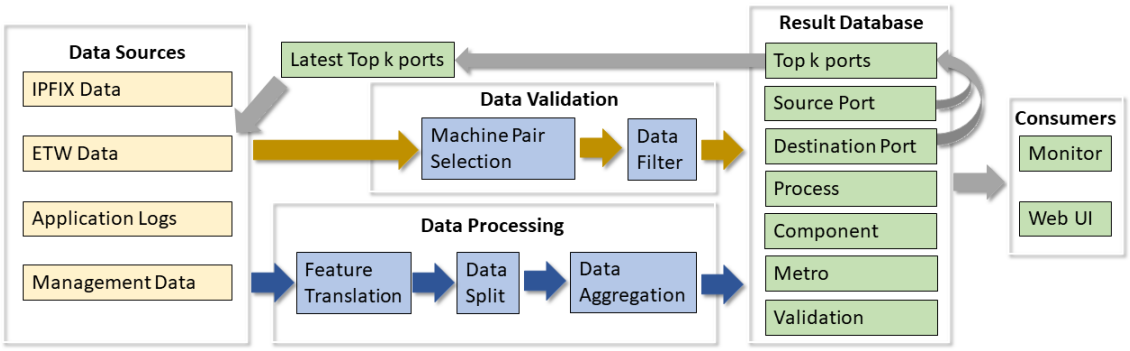


Figure 3: NetPanel Architecture Overview.

identified in Section 4.2.2.

4 System Design

Figure 3 shows the overview of our design. The data sources include IPFIX, ETW, application logs, and management data. There are two separate pipelines in the system. The first one (blue arrows) is responsible for reducing the data size while preserving important attributes. The second one (yellow arrows) is used to cross-validate the data from IPFIX and ETW to ensure data integrity.

The output of both pipelines is fed into the result database. The result database is a light-weighted database that can respond to web services as well as user queries in near real-time. We use Azure Data Explorer (Kusto) [10] for the result database. The result database contains tables for different features. The schema of the tables is available in Table 3. We use port tables in the result database to derive the top k ports with the largest traffic volume and use them as a filter for ETW collectors to reduce their resource consumption in the production environment (grey arrows). We also set up a monitoring service to help component owners detect anomalies and continuous upticks in their traffic usage. A WebUI is provided for traffic data visualization.

4.1 Data Sources

Before NetPanel, EXO deployed a background packet trace monitor on its servers to provide network statistics such as throughput, RTT, etc. However, without component-level information, this background tracking is not adequate to drive owners to optimize their traffic. This experience drives us to choose a different set of data sources for NetPanel.

NetPanel takes three measurement inputs: IPFIX, ETW, and application logs. The data is uploaded to COSMOS [38] on an hourly basis. COSMOS is a Hadoop-like distributed data storage and processing platform. We convert different types of data into a uniform format like Table 2. In addition to the measurement data, NetPanel also takes in management data. The management data contains the mapping between an IP address and its location and server role. The management data is uploaded daily because it is relatively static.

NetPanel handles the three data sources independently. IPFIX data size is more than 10TB per day. ETW and application logs data sizes are several PB per day. It is too expensive to directly join the three data sources based on shared features (i.e., TimeStamp, Port, IP, and Process). We thus process the raw data independently and only store aggregation results in the result database as detailed in the next section.

4.2 Data Processing

The data processing pipeline consists of three consecutive steps. (1) Feature translation: translate the raw information of the traffic data into a set of features. (2) Data split: divide the traffic data associated with source-destination port pairs into two separate views, i.e., source-port view and destination-port view. (3) Data aggregation: aggregate the traffic data into various feature tables.

4.2.1 Feature Translation

In Microsoft, different services occupy different blocks of IP ranges, so we use IP addresses[1] to retrieve EXO traffic. Then, we use management data to translate machine IPs into locations and server roles. The server role is among AD, FE, and BE. Location is the metropolitan area where the server locates. This translation reduces the storage requirements from trillions of IP pairs to millions of feature pairs.

We translate location pairs to Rate-Regions. Location pairs are only used to get the prices of traffic flows. A longer distance implies a higher price. We thus use a new feature called Rate-Region to replace the location pair of a flow. Azure charges Microsoft internal services a unified price ($/Mbps) for the flows traveling a geographical continent or an ocean, like North America, Europe, Atlantic, etc. There are only

[1]The IP addresses here are all Microsoft IPs.

Table	Data Source	Schema - Keys	Schema - Value (Bytes)
Source/Destination-Port	IPFIX	TimeStamp, ServerRole, RateRegion, Port, DSCP	TrafficSize
Process	ETW	TimeStamp, ServerRole, RateRegion, Port, Process	TrafficSize
Component	Application logs	TimeStamp, ServerRole, RateRegion, Port, Process, Component	RequestSize

Table 3: Columns in result tables.

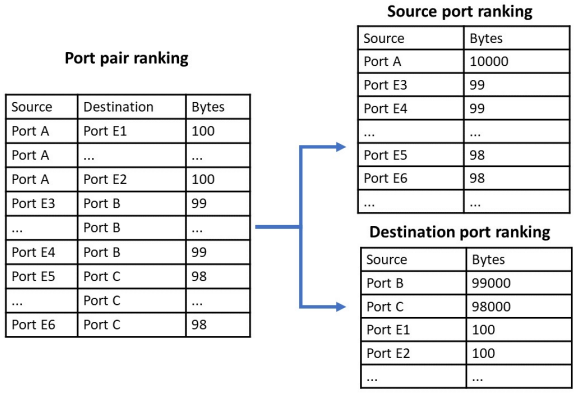


Figure 4: Split the port pair view into the source-port view and the destination-port view, separately. Port Es are ephemeral ports. Well-known ports A, B, and C pop up in the ranking list after the split.

about 10 Rate-Regions. Translating location pairs to Rate-Regions reduces the data size by ∼99.1%. We note that the feature translation does not hinder traffic debugging. One component is deployed on all machines of the same role. In case we detect anomalous traffic for one component in a Rate-Region, engineers can randomly pick a server in that Rate-Region and start the debugging from there.

4.2.2 Data Split and Aggregation

We aggregate the raw data based on the source port and destination port, separately, ranked based on the traffic volume. This results in two views: the source-port view and the destination-port view, which have two benefits: (1) significantly reduces the table size, i.e., from O(# source port $\times$ # destination port) to O(# source port + # destination port); (2) helps pop up "well-known" ports in ranking because the traffic of ephemeral ports will converge to relatively smaller numbers than the traffic of "well-known" ports after aggregation. An example of port view splitting is shown in Figure 4.

Once we have identified the "well-known" ports, we take an additional step to filter the ephemeral ports in the source-port table and destination-port table. For every single time slot, we aggregate all ports that contribute less than 1% of the total traffic to one record and mark it with a special tag. Recall that our goal is to reduce overall traffic cost, the threshold of 1% is small enough. We will show in Section 5.1.1 that a handful of ports dominate the traffic usage. The outputs are streamed into Kusto on a daily basis.

We aggregate the records from different data sources to obtain different feature tables. The schemas of these tables are shown in Table 3. We generate the Process table from ETW data, the Component table from application logs, and the Source-Port table and Destination-Port table from IPFIX data. We aggregate the TimeStamp with 5-minute intervals. TrafficSize is aggregated with sum operation for all the records with the same key. For example, the tuple of <TimeStamp, ServerRole, RateRegion, Port, DSCP> is the key for the Source-Port table. The TrafficSize in the Source/Destination-Port table and the Process table is the sum of network traffic while the RequestSize of the Component table is the sum of request/response content sizes, excluding packet headers.

4.3 Data Validation

As a global-scale system, there are many factors that result in data corruption such as broken hardware and data loss. During the development of NetPanel, we experienced a partial data loss of the IPFIX data due to the data collector pipeline change. In fact, data missing issues are non-trivial to detect due to the vast amount of data being processed. NetPanel resolves the problem by cross-validating the results obtained from its multiple data sources.

The key idea is to cross-validate the ETW data and IPFIX data because the traffic size captured with ETW should match that with IPFIX. However, because IPFIX does packet sampling while ETW captures all traffic, we need to recover IPFIX data before comparing them. The recovery function is shown in Eq. (1). We do not validate the traffic size using application logs, because they only capture the content sizes without the request headers.

$$IPFIXBytes = \frac{(PacketSize + HeaderSize) * PacketNumber}{SamplingRate} \quad (1)$$

PacketSize, PacketNumber, and SamplingRate are available from the IPFIX data. Note that we add the ethernet header length (i.e., HeaderSize) to PacketSize to get the actual ethernet frame size on the wire. Based on the law of large numbers, we believe that: Given a machine pair that continuously send a lot of traffic to each other, an effective estimation of IPFIX should be close to ETW data. We conducted an experiment to validate our recovery approach. The result shown in Figure 5

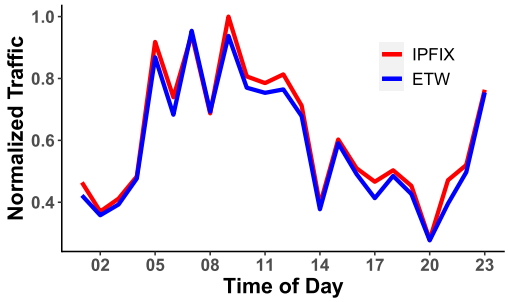


Figure 5: Recovered traffic for a single pair of machines in one day from IPFIX matches the traffic from ETW.

can confirm that the recovered IPFIX traffic size matches the ETW measurement precisely.

We selected the top 1000 pairs of machines in EXO that send the most traffic in a day for data validation. Machine pairs are re-selected daily as machine pairing relationships may change when machines become online or offline. After selecting the machine pairs, we insert their IPFIX data and ETW data into the validation table. Direct data comparison could be done with queries to the validation table. Data validation is done on a daily basis.

4.4 Result Database

The result database contains the Source-Port table, the Destination-Port table, the Process table, and the Component table. These tables are used to analyze traffic for component owners. A validation table is stored for data integrity checks. A top k ports table is created based on the Source-Port table and Destination-Port table. It picks the top k ports in both tables. These top k ports will be used as filters for the ETW collector in the future.

NetPanel uses the latest top k ports as a filter for the ETW collector to save production resources. These top k ports are usually obtained last day. IPFIX and Application Logs have been deployed in our environment for a long time. These data collectors have been optimized to guarantee no impact on SLA. ETW collector is a newly added collector running on production servers. As we will show in Section 6.1, directly collecting all ETW records will result in tens of times the CPU and disk IO usage. We must reduce ETW collector's resource consumption on production servers. With the observation in Section 5.1.1 that the top 10 ports cover more than 94% of the traffic and stay stable over weeks. We decide to use the Top-k algorithm as a filter in the ETW collector to reduce the data to be collected and thus reduce its resource consumption.

4.5 Component Traffic Estimation

We built a web UI to provide engineers with a convenient way to analyze the traffic. The dashboard shows component-level traffic like shown in Figure 2b (in Section 3). If the component monopolizes a process, its traffic could be directly obtained from the Process table as described in Table 3. Otherwise, we calculate its traffic in an approximated fashion. The details are presented in Algorithm 1.

Input: Component C
Output: Traffic size of Component C per second
1 **if** Process Contains C **then**
2 | **return** Process[C]
3 **else**
4 | P = Component C's RemotePort
5 | PortTraffic = Source-Port[P] + Destination-Port[P]
6 | **return** $\frac{Component[C,P]}{Component[*,P]} \times$ PortTraffic
7 **end**

Algorithm 1: Calculation Algorithm

If a component shares a process with other components, we have to estimate its network traffic following Line 4 to 7. In Line 4, we first find the port P used by component C with the Component table. Then, in Line 5, we calculate the total network traffic of this port P with the Source-Port table and Destination-Port table. In Line 6, Component[*, P] is the total request/response size of all components (* is the wildcard) going through port P. Component[C, P] is the request/response size of Component C going through port P. We use the ratio of Component[C, P] and Component[*, P], and the PortTraffic of P to estimate the TrafficSize of component C. The underlying assumption is that different components suffer from similar packet loss patterns (i.e., similar retry rate) and the packet header size is proportional to the payload size.

4.6 Monitoring

Aside from the dashboard, we also create monitors on the Source-Port table and Destination-Port table in the result database to safeguard the overall traffic usage of EXO. We now support two types of monitors, one to capture the static trend and the other to capture the dynamic changepoints.

We use Mann-Kendall trend test [25] to obtain the static trend and use LinkedIn's Greykite [27] to capture dynamic change points. The static trend monitor could help us find feature roll-out that potentially generates sub-optimal traffic. The dynamic change-points monitor could discover sudden bursts in traffic. These sudden bursts are usually caused by code regression or configuration errors.

5 Production Results

We used NetPanel to support traffic analysis in EXO. In this section, we share the observed traffic characteristics and use case studies to show the effectiveness of NetPanel.

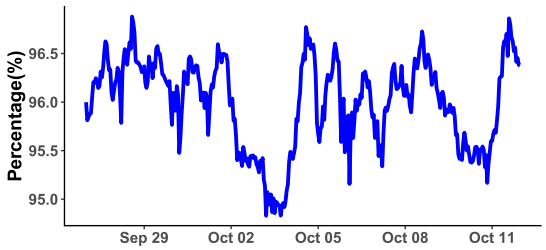


Figure 6: Traffic percentage taken by the top 10 ports. The percentage stays stable during the half-month observation.

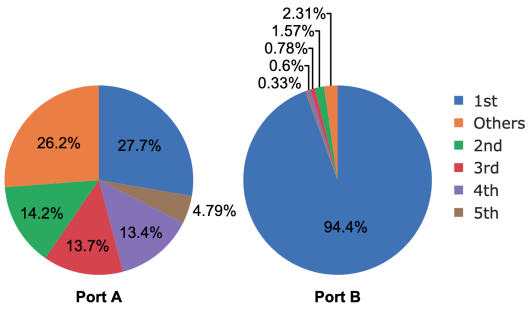


Figure 7: Component-level traffic distribution for the top 2 ports.

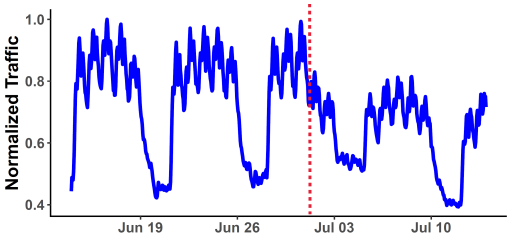


Figure 8: The traffic with destination port A decreased by 20% after switching from external to internal endpoint. The vertical dotted line indicates the recover time.

5.1 Traffic Overview

Recall that there are two tiers of traffic in Sec. 2.1. To use Tier 1 traffic, the owner team must register a specific port. All traffic to or from that port is labeled with a Tier 1 DSCP tag. The registration prevents other components from using the same port. Therefore, all Tier 1 traffic through that port is used exclusively by a single component. In the following, we only discuss the characteristics of Tier 0 traffic.

5.1.1 Several ports dominate the overall traffic

From the Source-Port and Destination-Port tables, we observe that several top ports contribute to most traffic usage. Figure 6 shows the percentage of total traffic generated by the top 10 ports over half a month period. The lowest value during this period is 94.8%. This result confirms that the Top-k ports filter algorithm could cover at least 94% of the total traffic for ETW collector when k = 10. Furthermore, the list of top ports remains unchanged during our observation. This observation supports us to use the latest top k ports obtained as a filter for the ETW collectors as described in Section 4.4. The concentrated traffic distribution at a few top ports saves us a lot of traffic optimization work. We can focus on a small number of ports when we want to reduce network traffic costs. We engage the partner teams that use these ports instead of calling every team in EXO.

5.1.2 Several components dominate the traffic of a top port

When multiple components share the same port (as in Figure 1 REST and EWS share the same port), there are many contributors to the port traffic. We analyze the traffic distribution for top ports. We show the traffic contribution of the top 5 components for the top 2 ports separately in Figure 7, labeled as A and B. For both ports, the contribution of the 6th component is less than 5%. Investigation into lower-ranked components does not have a significant benefit in reducing overall traffic usage. With the help of NetPanel, engaging with a few partner teams is usually enough to optimize the traffic of a top port. For example, when we want to reduce the traffic for Port A, we only need to engage the owners of the top 5 components.

5.2 Case Studies

In this section, we show how we use NetPanel to refine the WAN traffic for EXO. We present 4 cases where we leveraged NetPanel for: (1) service traffic optimization; (2) legacy traffic discovery; (3) anomaly traffic burst detection; and (4) WAN feature validation. These actions save millions of dollars per year for EXO.

5.2.1 Service Traffic Optimization

In this case, we introduce how we use NetPanel to identify sub-optimal traffic and optimize the utilization of network resources. NetPanel provides long-term data visualization. This makes it easy for the application to identify its major traffic contributor. After engaging the owning team, it is easy to make a conclusion on whether the traffic is necessary.

During our investigation of the traffic composition of EXO, we discovered that the Tier 0 traffic from our BE servers to port A on FE servers is too large. Because EXO has optimized the internal service endpoint (URL) to serve requests with the nearest FE server, we expect that there should be little Tier 0 traffic from BE to FE. We then used the Component table to find the main contributors. We identified that a component

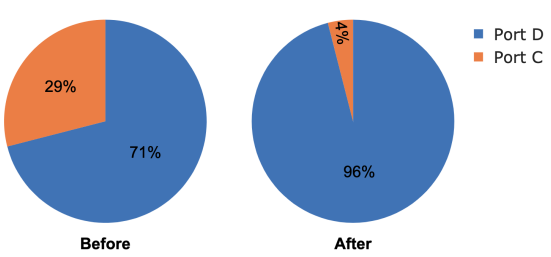


Figure 9: The ratio of the traffic from Port C to the traffic from Port D before and after the fix.

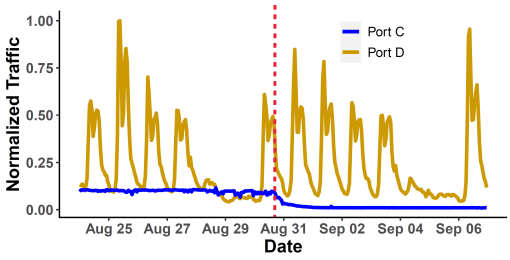


Figure 10: Traffic volume change with source port C and source port D after the removal of legacy code. The vertical dotted line indicates the change time.

that is used to extract plain text content from different formats of documents in emails took up most of the traffic. Then, we contacted the owning team and figured out that the component was using an external service endpoint. It had no control of where the requests were sent to and thus caused massive Tier 0 traffic worldwide. After identifying the issue, we worked together with the owning team to change to use an internal service endpoint so that this component could serve its requests within a single location. This action saved millions of dollars per year. The traffic change is shown in Figure 8. The traffic with destination port A decreased by 20% after the change. Prior to NetPanel, it relied heavily on code reviews to prevent such cases. This manual approach was likely to lead to omissions.

5.2.2 Legacy Traffic Discovery

There is another case where NetPanel helped discover legacy traffic and its source code. As the system grows, the source code becomes overwhelming. The work to remove outdated settings and services is necessary. However, it turns out that it is hard to discover legacies while do not break anything. We figured out there was one component on BE machines that was sending traffic to certificate authorities to validate some certificates that had been retired. We started from the continuous unexpected high traffic volume from port C in the NetPanel Source-Port table. According to our knowledge, EXO had finished the migration from Port C to Port D, so there should be very little traffic from port C, but massive

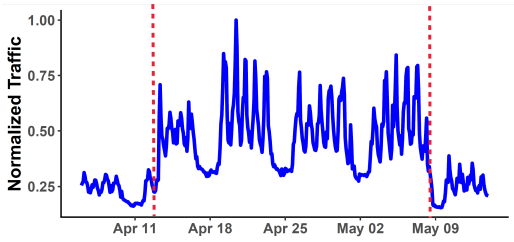


Figure 11: Traffic volume change of the log uploading service during the configuration error. The vertical dotted lines indicate the start time and the end time, respectively.

traffic from port D. The left pie chart in Figure 9 shows the ratio of the traffic from port C to the traffic from port D. The traffic from port C is 42% of the traffic from port D, which is much higher than our expectation. We used NetPanel to find the suspicious component and its owning team. After reaching out to the team, they investigated the issue and removed the legacy code. After the fix, the ratio of the traffic from port C to the traffic from port D became much smaller, where the traffic from port C was reduced to 4.2% of the traffic from port D, as shown in the right pie chart of Figure 9. The traffic of both ports during the fix is shown in Figure 10. After the removal, the traffic with source port C dropped to lower than 10% of the original traffic. That fix led to a savings of inbound capacity in the Gbps for Microsoft.

5.2.3 Anomaly Traffic Burst Detection

This case shows how we leverage NetPanel to detect an anomaly traffic burst caused by a configuration error. Some configuration errors at the application level may cause abnormal network behaviors. For example, a configuration error may cause the component to keep sending requests to an endpoint and thus leads to a traffic burst. These bugs are hard to detect unless the affected machines get so hot and break critical services [28]. In NetPanel, the change in request volume can be detected. When anomalies are reported by NetPanel, we could contact the component owner to debug the issue.

Figure 11 visualizes the traffic change during a configuration error for one component in EXO. An engineer accidentally changed a log uploading compression algorithm to an older version. The decrease in compression rate led to an increase in the total data volume sent to the log center and eventually hit the throttling limit. However, the component had a retry mechanism and kept sending data. This resulted in nearly quadruple the traffic volume sent during peak hours. The error was detected late after the change rolled out worldwide and drove a dramatic increase in traffic volume. We caught this abnormal burst and contacted the owner to dig into their component. After rolling back the change, the abnormal traffic disappeared and the traffic dropped back to the previous level. Without NetPanel, the problem could only be

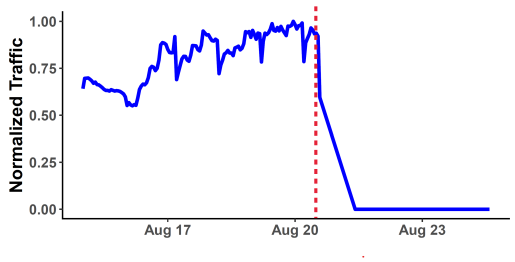


Figure 12: The volume change of Tier 0 traffic sent to registered ports after RWA devices support QoS Policy. The vertical dotted line indicates the change time.

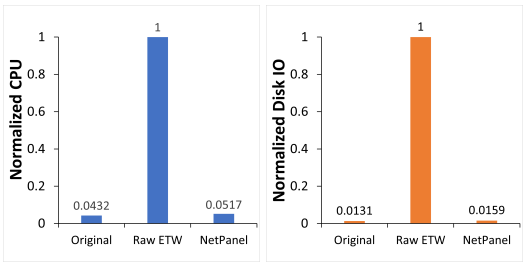


Figure 13: Normalized CPU and Disk IO usage on a BE machine when only services are running, when raw ETW is running, and when NetPanel is running.

discovered until the availability of the affected component is severely impacted, which would take a much longer time to fix the error and waste much more bandwidth.

5.2.4 WAN Feature Validation

This section introduces how NetPanel triggers a feature validation on newly deployed devices. When a new type of device starts to be deployed in the network, it may miss some functionality. Regional WAN Aggregator (RWA) is a kind of core router recently added to the Azure Network that sits on the top of datacenters and is responsible to connect the datacenter network to the backbone WAN. In this case, the newly deployed RWA devices do not support the QoS policy. They ignore the DSCP tag and treat all traffic with Tier 0 priority. NetPanel helped fix the bug at an early stage and avoid purchasing tens of millions of dollars in redundant capacity.

Bandwidth Broker routes traffic with different priorities. Our replication service has registered ports on the Bandwidth Broker service to lower the priority of non-urgent replications to Tier 1. However, from NetPanel data, we found that these ports have a significant amount of traffic classified as Tier 0. Together with the owning team, we picked a machine pair and captured packets on routers along the routing path. We found that the RWA device on the path does not support the QoS policy. Even though only 2 locations had RWA devices deployed at an early stage, this fix already saved several million dollars each year. If RWA devices were deployed worldwide, the value of this fix would be tens of millions of dollars per year. As shown in Figure 12, the Tier 0 traffic for those registered ports dropped to zero after the fix. NetPanel builds a map between components and ports, which makes it easy for component owners to monitor the priority tier of its traffic. Once any bug is introduced on the routing plane, they could discover and fix it quickly.

6 Evaluation

In this section, we evaluate the overhead of NetPanel in two categories: overhead inside and outside the production environment. The production environment serves customer requests and thus has strong restrictions on resource consumption, while the restrictions on overhead outside the production environment are more relaxed.

6.1 Overhead in the Production Environment

Figure 13 shows our evaluation of NetPanel overhead in the production environment. We ran the test on a single server. The overhead of NetPanel in the production environment is introduced by the ETW collector, which has extremely high CPU and disk IO consumption. If we enable ETW fully on machines, it will occupy 25× more CPU and 77× more disk IO compared to those when ETW is disabled. When we deploy NetPanel on production machines, less than 1% rise of the total available CPU and disk IO are observed because of retaining only top ports as mentioned in Section 4.4. It saves 99.1% CPU and 99.7% Disk IO compared with Raw ETW.

6.2 Overhead outside the Production Environment

The overhead outside the production environment includes two parts: data storage and computation. These are the overhead of processing and storing data in the big-data platform.

6.2.1 Data Storage

Table 4 shows the ratio between result data and raw data using our approach. The result size is 0.00361% of the origin for IPFIX, 0.00076% for ETW, and 0.00003% for application logs. The daily result sizes of all three data sources are in GB and are close to each other. The huge difference in the ratios is caused by the huge difference in the size of the original data. ETW data is much larger than IPFIX because it does not perform data sampling and covers Metro traffic (traffic within a location). Application log data is even larger because a machine pair could send multiple requests in one connection. The log data contains many extra columns for debugging.

Data	Size Ratio	Daily Calculation
IPFIX	0.00361%	1.1 hours
ETW	0.00076%	2.5 hours
Application Logs	0.00003%	6.8 hours

Table 4: Result data size ratio (compared to the raw data) and calculation time.

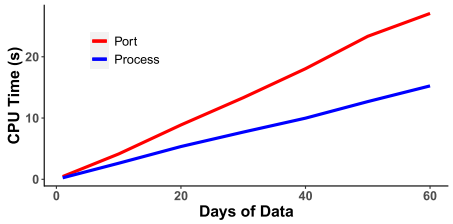


Figure 14: The CPU time for a query on a port and a query on a process increase linearly with the days of data. It takes 27 CPU seconds to get the 60 days of traffic for a port, and 15 CPU seconds to get the 60 days of traffic for a process.

6.2.2 Computation

The computational overhead consists of two parts. The first part is the overhead of background data processing and validation pipelines. The jobs run every day. The second part is the overhead of user queries.

We show the computational overhead for NetPanel's data processing and validation pipelines in Table 4. We use less than 0.01% of our production computation resources to process the data. It takes 1.1 hours to process IPFIX data, 1.6 hours to process ETW data, and 6.8 hours to process application logs every day. The validation pipeline takes around 1.5 hours. The tight resource restriction leads to a long calculation time. We make this trade-off because the goal of NetPanel is to support the reduction of traffic costs in the long run, and therefore a delay of hours is acceptable.

It typically takes no more than 30 seconds for NePanel to respond to a user query for the traffic of a set of specific ports or processes over months. We show the CPU time NetPanel needs to respond to a user query in Figure 14. The CPU time is proportional to data size. The actual waiting time is usually much smaller than the CPU time because multiple CPUs can calculate the result in parallel. The overall overhead of user queries is usually negligible.

7 Related Work

EXO runs on Azure architecture, further details of Azure architecture are shown in [30]. Gunawi [24] provided a summary of 597 cloud outages from 2009 to 2015. Ardelean [17] used Gmail as an example to analyze the performance of cloud applications. They studied the dynamic nature of cloud applications in short time intervals. There are multiple efforts to minimize the cost of operating data centers. Cascara [42] tried to optimize the edge cost. Yang et. al. [50] scheduled the bulk transfer among datacenters. ROTOS [49] designed an optical DCN architecture to improve power efficiency.

Large investments have been made in data-center monitoring. There is a body of work on server monitors for Linux and Windows [8, 11, 36, 43]. Trumpet [36] is a monitor that processes every packet at line rate on end-hosts and tests the presence of user-specified network events. PathDump [43] designs a server stack to retrieve metadata from arrived packets on edge devices to help debug network issues. Monitoring data-center networks has been a hot topic for years. Commonly used protocols include IPFIX/Netflow and sFlow [15, 22, 37]. A detailed review [26] of the general flow monitoring technique is provided. In recent years, researchers have refined flow monitoring for different purposes [33, 40]. To monitor all the flows without sampling, FlowRadar [33] encodes per-flow counters at switches and leverages the computing power at the remote collector to perform decoding. To handle the large-scale challenge of data-centers, many have turned to query-based solutions, including Stroboscope [46], OFRewind [48], and PacketScope [45]. Stroboscope [46] mirrors millisecond-long traffic slices on routers according to user queries to monitor network forwarding behavior including traffic paths, one-way delays, and load-balancing ratios. IBM [29] uses HTTP logs to detect component failure and provide reports over the last 48 hours when an anomaly is detected.

8 Conclusion

The vast amount of network traffic generated by the components in cloud applications consumes significant resources. It is vital to identify the composition of network traffic to reduce the cost. Component-level measurement is needed to drive the traffic optimization effort for systems developed by a large number of teams. NetPanel analyzes the network traffic for EXO at the component level and at a low cost. One primary challenge is caused by the huge amount of measurement data. We design several schemes to reduce the data size without losing fidelity. We discuss real cases where NetPanel is applied to save millions of dollars per year. We believe that the insights we gained during the design and operation of NetPanel provide valuable experience in traffic measurement and reduction of other cloud applications.

Acknowledgements

We thank the anonymous reviewers and our shepherd Srinivas Narayana, for their valuable comments and insightful recommendations to improve the quality of the paper. We thank all Microsoft engineers who have been closely working with us for their great support.

References

[1] Amazon cloud watcher. https://aws.amazon.com/cloudwatch/.

[2] Azure status history. https://status.azure.com/en-us/status/history.

[3] Google cloud monitoring. https://cloud.google.com/monitoring/.

[4] IIS Application Pool. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-r2-and-2008/cc735247(v=ws.10).

[5] Outlook EWS Reference. https://docs.microsoft.com/en-us/exchange/client-developer/exchange-web-services/start-using-web-services-in-exchange.

[6] Outlook MAPI Reference. https://docs.microsoft.com/en-us/office/client-developer/outlook/mapi/outlook-mapi-reference.

[7] Salesforce disruption: Na2 and next day, cs1. http://iwgcr.org/salesforce-disruption/.

[8] TCPDUMP/LIBPCAP public repository. https://www.tcpdump.org/.

[9] Update on today's gmail outage. https://gmail.googleblog.com/2009/02/update-on-todays-gmail-outage.html.

[10] Azure Data Explorer: a big data analytics cloud platform optimized for interactive, adhoc queries over structured, semi-structured and unstructured data, 2018. https://azure.microsoft.com/mediahandler/files/resourcefiles/azure-data-explorer/Azure_Data_Explorer_white_paper.pdf.

[11] Event Tracing for Windows, 2018. https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing.

[12] Aws outage analysis: December 15, 2021, 2021. https://https://www.thousandeyes.com/blog/aws-outage-analysis-december-15-2021/.

[13] Service Level Agreements (SLA) for Online Services, 2021. https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services.

[14] us-central1: Google app engine standard experiencing increased latency and pending queue aborted error request rate., 2021. https://status.cloud.google.com/incidents/uaRinwS8pu2yyjdYbDaM.

[15] Paul Aitken, Benoît Claise, and Brian Trammell. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, September 2013.

[16] Ali Anwar, Anca Sailer, Andrzej Kochut, and Ali R. Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In Proceedings of the 6th Asia-Pacific Workshop on Systems, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 405–417, Renton, WA, April 2018. USENIX Association.

[18] Fred Baker, David L. Black, Kathleen Nichols, and Steven L. Blake. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, December 1998.

[19] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Hyojoon Kim, Renata Teixeira, and Nick Feamster. Traffic refinery: Cost-aware data representation for machine learning on network traffic. Proc. ACM Meas. Anal. Comput. Syst., 5(3), dec 2021.

[20] Matt Calder, Manuel Schroder, Ryan Gao, Ryan Stewart, Jitu Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. Odin: Microsoft's scalable fault-tolerant cdn measurement system. In USENIX NSDI, April 2018.

[21] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, October 2004.

[22] Luca Deri, Ellie Chou, Zach Cherian, Kedar Karmarkar, and Mike Patterson. Increasing data center network visibility with cisco netflow-lite. In Proceedings of the 7th International Conference on Network and Services Management, CNSM '11, page 274–279, Laxenburg, AUT, 2011. International Federation for Information Processing.

[23] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. University of California, Irvine, 2000.

[24] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16, page 1–16, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Khaled H. Hamed and A. Ramachandra Rao. A modified mann-kendall trend test for autocorrelated data. Journal of Hydrology, 204(1):182–196, 1998.

[26] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. IEEE Communications Surveys Tutorials, 16(4):2037–2064, 2014.

[27] Reza Hosseini, Albert Chen, Kaixu Yang, Sayan Patra, and Rachit Arora. Greykite: a flexible, intuitive and fast forecasting library, 2021.

[28] Ryan Huang, Chuanxiong Guo, Lidong Zhou, Jay Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In Proceedings of the ACM Workshop on Hot Topics in Operating Systems (HotOS). ACM, May 2017.

[29] Mohammad S. Islam, William Pourmajidi, Lei Zhang, John Steinbacher, Tony Erwin, and Andriy Miranskyy. Anomaly detection in a large-scale cloud platform. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 150–159, 2021.

[30] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Peter Kromkowski, Shaoran Li, Wenxi Zhao, Brendan Abraham, Austin Osborne, and Donald E. Brown. Evaluating statistical models for network traffic anomaly detection. In 2019 Systems and Information Engineering Design Symposium (SIEDS), pages 1–6, 2019.

[32] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 1–14, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), pages 311–324, Santa Clara, CA, March 2016. USENIX Association.

[34] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An intelligent, End-To-End analytics service for safe deployment in Large-Scale cloud infrastructure. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020.

[35] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 101–114, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 129–143, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Sonia Panchen, Neil McKee, and Peter Phaal. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176, September 2001.

[38] Hiren Patel, Alekh Jindal, and Clemens Szyperski. Big data processing at microsoft: Hyper scale, massive complexity, and minimal cost. In Symposium on Cloud Computing, page 490. ACM, November 2019.

[39] William Pourmajidi, Andriy Miranskyy, John Steinbacher, Tony Erwin, and David Godwin. Dogfooding: Using ibm cloud services to monitor ibm cloud infrastructure. In Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON '19, page 344–353, USA, 2019. IBM Corp.

[40] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, page 407–418, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Vyas Sekar, Michael K. Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC '10, page 328–341, New York, NY, USA, 2010. Association for Computing Machinery.

[42] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 201–216. USENIX Association, April 2021.

[43] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, page 233–248, USA, 2016. USENIX Association.

[44] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with SwitchPointer. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 453–456, Renton, WA, April 2018. USENIX Association.

[45] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In Proceedings of the Symposium on SDN Research, SOSR '20, page 76–82, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 467–482, Renton, WA, April 2018. USENIX Association.

[47] Jianwen Wei, Yusu Zhao, Kaida Jiang, Rui Xie, and Yaohui Jin. Analysis farm: A cloud-based scalable aggregation and query platform for network log analysis. In 2011 International Conference on Cloud and Service Computing, pages 354–359, 2011.

[48] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In 2011 USENIX Annual Technical Conference (USENIX ATC 11), Portland, OR, June 2011. USENIX Association.

[49] Xuwei Xue, Fulong Yan, Kristif Prifti, Fu Wang, Bitao Pan, Xiaotao Guo, Shaojuan Zhang, and Nicola Calabretta. Rotos: A reconfigurable and cost-effective architecture for high-performance optical data center networks. Journal of Lightwave Technology, 38(13):3485–3494, 2020.

[50] Zhenjie Yang, Yong Cui, Xin Wang, Yadong Liu, Minming Li, Shihan Xiao, and Chuming Li. Cost-efficient scheduling of bulk transfers in inter-datacenter wans. IEEE/ACM Transactions on Networking, 27(5):1973–1986, 2019.

[51] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. Static Detection of Silent Misconfigurations with Deep Interaction Analysis. In Proceedings of the 36th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'21), October 2021.

[52] Lixia Zhang, Van Jacobson, and Kathleen Nichols. A Two-bit Differentiated Services Architecture for the Internet. RFC 2638, July 1999.

[53] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. An Evolutionary Study of Configuration Design and Implementation in Cloud Systems. In Proceedings of the 43rd International Conference on Software Engineering (ICSE'21), May 2021.

[54] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. LightGuardian: A Full-Visibility, lightweight, in-band telemetry system using sketchlets. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 991–1010. USENIX Association, April 2021.

[55] Tanja Zseby, Benoît Claise, Juergen Quittek, and Sebastian Zander. Requirements for IP Flow Information Export (IPFIX). RFC 3917, October 2004.

DOTE: Rethinking (Predictive) WAN Traffic Engineering

Yarin Perry[1], Felipe Vieira Frujeri[2], Chaim Hoch[1], Srikanth Kandula[2], Ishai Menache[2], Michael Schapira[1], and Aviv Tamar[3]

[1]Hebrew University of Jerusalem, [2]Microsoft Research, [3]Technion

Abstract– We explore a new design point for traffic engineering on wide-area networks (WANs): *directly* optimizing traffic flow on the WAN using only *historical* data about traffic demands. Doing so obviates the need to explicitly estimate, or predict, future demands. Our method, which utilizes stochastic optimization, provably converges to the global optimum in well-studied theoretical models. We employ deep learning to scale to large WANs and real-world traffic. Our extensive empirical evaluation on real-world traffic and network topologies establishes that our approach's TE quality almost matches that of an (infeasible) omniscient oracle, outperforming previously proposed approaches, and also substantially lowers runtimes.

1 Introduction

To meet the constant rise in traffic, service providers invest huge effort into traffic engineering (TE)—optimizing traffic flow across their backbone WANs [11, 22, 24, 28, 37, 39, 57], which interconnect their datacenters with each other and with external networks. The production state-of-the-art involves periodically solving a (logically centralized) optimization problem to determine how to best split traffic across network paths. Changes to TE configurations are realized using software-defined control of network hardware [11, 22, 24, 35, 38, 39].

A key challenge for WAN TE is uncertainty regarding future traffic demands. The standard approach for contending with this is twofold. For time-sensitive traffic, providers measure application-specific usage data from switches (e.g., using sampled netflow or ipfix counters) and attempt to *predict* future usage. For bandwidth-hungry, scavenger-class traffic [22], providers deploy so called agents/shims in the OS of hosts from which traffic originates. These agents explicitly signal applications' traffic demands to "brokers" that, in turn, aggregate demands, relay them to the centralized optimizer, and enforce the resulting rate allocations [22, 24].

Both of the above approaches for handling traffic uncertainty have drawbacks. Demand predictions can naturally be erroneous and, more importantly, there is an objective mismatch between the loss functions to predict future traffic demands (e.g., mean-squared-error, L1 norm error) and the *end-to-end objective* of producing high-performance TE configurations. For example, mean-squared-error would weight error in *any demand* equally, yet errors on demands that are more problematic to carry on a given topology will exert a disproportionately large effect on TE quality. The other approach – brokering and explicitly specifying demands – entails nontrivial operational overheads, including changes to end-hosts and applications. This can increase the lag experienced by application requests (which is why this approach is used in practice only for bandwidth-hungry, scavenger-class traffic [22]).

The demand uncertainty challenge is further amplified for *customer-facing traffic* (web, images, e-mails, videos, etc.), which constitutes a large and growing share of the total traffic traversing some providers' backbones. For such traffic, which originates in unmodified apps or clients, brokering in the host OS is not applicable. Moreover (see §2.1), such traffic exhibits high variability and is difficult to predict accurately.

We explore a new design point for WAN TE: training a TE decision model on *historical* data about traffic demands to *directly* output high-quality TE configurations. We present the *DOTE* (Direct Optimization for Traffic Engineering) TE framework. *DOTE* applies *stochastic optimization* to *learn* how to map recently observed traffic demands (e.g., empirically-derived traffic demands from the last hour) to the next choice of TE configuration. Using *DOTE*, providers need only *passively* monitor traffic to/from datacenters and do not have to onboard applications onto brokers. Directly predicting TE outcomes that optimize TE performance also resolves the objective mismatch between demand prediction and TE performance, yielding TE outcomes that are more robust to traffic unpredictability. We show how *DOTE* can scale to handle large WANs and real-world traffic by harnessing the expressiveness of deep learning.

We evaluate *DOTE* both analytically and empirically. Our theoretical results establish that if the TE optimization objective satisfies desirable convexity/concavity properties, *DOTE provably* converges to the optimum. We prove that this is indeed the case for standard TE optimization objectives such as minimizing the maximum-link-utilization (MLU) [8, 14, 27], maximizing network throughput [4, 22, 24, 37], and maximizing concurrent-flow [11, 29].

Our empirical evaluation compares *DOTE*, in terms of both quality and runtimes, to TE with explicit demand estimates from end-hosts, demand-prediction-based TE, demand-oblivious TE, deep-reinforcement-learning-based TE, and more. Evaluating data-driven TE schemes like *DOTE* requires substantial empirical data regarding traffic conditions

for both training and performance analysis. We conduct a large-scale empirical study using both publicly available datasets and historical data from Microsoft's private WAN. These datasets span months of traffic demands at few-minutes granularity, amounting to tens of thousands of demand snapshots. Our evaluation covers small (10s of nodes) and large (100s of nodes) WANs, different types of traffic (including inter-datacenter and customer-facing), and different TE optimization objectives. To facilitate reproducibility, our code is available at [2].

Our evaluation results show that:

- ***DOTE* achieves TE quality almost matching that of an *infeasible* oracle with *perfect knowledge* of future demands**. Across all evaluated network topologies, traffic traces, and considered TE objectives, *DOTE* compares favorably to all other evaluated TE schemes. We also demonstrate *DOTE*'s robustness to changes in traffic conditions and to network failures.
- **By invoking a DNN for the online computation of TE configurations, *DOTE* achieves runtimes 1-2 orders of magnitude faster** than solving a linear program (LP), even for large WANs, matching the gains from recent proposals for fast (approximate) LP optimizations [4,40]. Our approach thus also holds promise for expediting decision making for TE.

We view our investigation of direct optimization for WAN TE as a first step and discuss current limitations of our approach that we hope future research can address.
This work does not raise any ethical concerns.[1]

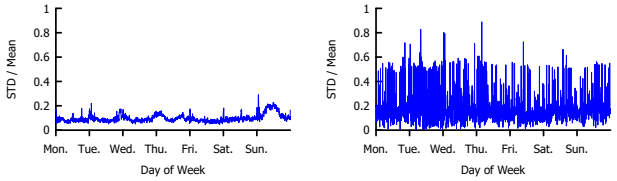
2 Motivation and Key Insights

2.1 Inter-DC *vs.* Customer-Facing Traffic

Enterprise WANs carry traffic between the provider's own datacenters (e.g., geo-replication of datasets, newly computed search indices) as well as traffic traversing the backbone towards/from customers (e.g., web traffic, videos).

To motivate our direct optimization approach, we present analyses of traffic on Microsoft's production WAN. Figure 1(a) plots the standard deviation in *inter-datacenter traffic* demands, normalized by the mean, across 11 consecutive weeks, for the pair of datacenters with the highest average demand. Demands are collected at 5-minute granularity. Similarly, Figure 1(b) plots the normalized standard deviation in *customer-facing traffic* demands over 4 consecutive weeks for the pair of nodes with the highest average demand. Observe the substantial difference; in the inter-datacenter traffic trace, demands are *significantly* less variable.

[1] In particular, the measured traffic demands, used in our evaluation, are aggregate counters between pairs of datacenters at the granularity of minutes (or coarser). They do not contain user IP addresses or packet contents.



(a) Inter-data-center traffic (b) Customer-facing traffic

Figure 1: Variability in traffic demands for inter-datacenter traffic and customer-facing traffic across different weeks.

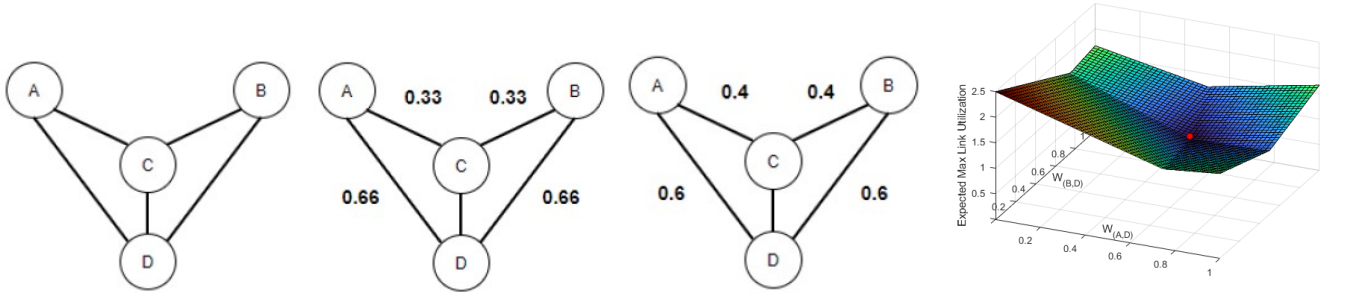
High variability in customer-facing traffic demands can accrue from different sources, e.g., (1) flash-crowds that may cause a surge in search requests, e-mail volume, etc., (2) congestion on the WAN's peering links with ISP networks, and (3) route changes and outages that cause traffic to ingress or egress the WAN at different sites. We have observed that customer-facing demands can exceed $100\times$ the average value for extended stretches of time. Thus, customer-facing traffic is harder to accurately predict than inter-datacenter traffic. See Figure 10(a)–Figure 10(b) in the appendix for differences in demand-prediction accuracy between the above discussed two traffic traces.

To summarize: for customer-facing traffic, which is a large and growing share of overall WAN traffic, not only is direct inference of traffic demands by the host OS infeasible, but accurate demand prediction also appears elusive. We seek a method that can achieve nearly optimal TE outcomes even for the unpredictable traffic demands.

2.2 Demand Prediction *vs.* Direct Optimization

We illustrate key insights underlying *DOTE* using the example in Figure 2(a). Each of nodes A and B wishes to send traffic to node D, and can do so either via its direct link to D or its 2-hop path to D through node C. All link capacities are 1. Every fixed time interval (say, 5 minutes), the TE system must determine, for each of the two source nodes, A and B, traffic splitting ratios specifying which fraction of its demand is forwarded along each of its assigned two paths to D. A and B's traffic demands for each time interval are drawn (i.i.d) at the beginning of each time interval from a *fixed* probability distribution: with probability $\frac{1}{2}$ node A's demand is $\frac{5}{3}$ and node B's demand is $\frac{5}{6}$ and with probability $\frac{1}{2}$ node B's demand is $\frac{5}{3}$ and node A's demand is $\frac{5}{6}$. The TE system has no a priori knowledge of the realization of the traffic demands; splitting ratios must be determined before actual traffic demands are revealed.

Demand-prediction-based TE and its shortcomings. A natural solution is training a predictor on empirical data containing past demands for A and B to predict the combination of demands closest (in expectation) to the realized combination of demands (*e.g.*, in terms of mean-squared-error), and then



(a) Network topology (b) Induced splitting ratios for a demand-predictor (c) Optimal splitting ratios (d) Expected MLU as a function of the splitting ratios

Figure 2: Simple WAN TE example

performing global optimization with respect to the predicted demands. In our simple example, this leads to the predicted demand-combination being $(\frac{5}{4},\frac{5}{4})$ and the induced splitting ratios presented in Figure 2(b). Under these splitting ratios, *regardless* of the realization of the demands, either link (A,D) or link (B,D) will carry more traffic than its capacity can accommodate. In the optimal solution shown in Figure 2(c), however, *regardless of the realized demands, no link carries more traffic than its capacity can support*.

Of course, instead of predicting a single demand-combination, one could have predicted a *probability distribution* over the traffic demands and optimized splitting ratios with respect to that. This entails two nontrivial challenges, which are significantly amplified for large WANs and real-world traffic: (1) We must impose a specific structure on the probability distribution to be predicted (e.g., Gaussian, bimodal), which might not be a good fit for actual WAN traffic. This is particularly true when there hidden correlations between demands (as in our example); (2) Optimizing an LP with respect to a distribution over *multiple* demand-combinations can be prohibitively time consuming for large WANs.

On *direct* optimization of traffic splitting ratios and why it might do better. An alternative approach, which avoids presuppositions regarding the traffic, and also LP optimizations, is training a decision model on past realizations of A and B's traffic demands to *directly* output traffic splitting ratios that are close to the global optimum. This approach can outperform the demand-prediction-based approach in scenarios where traffic is volatile and hard to predict but a certain configuration of splitting ratios performs well on most traffic realizations. Directly inferring the splitting ratios also obviates the need for solving an LP to optimize splitting ratios with respect to predicted traffic. As our evaluation results in §4 show, this significantly accelerates TE runtimes for large WANs. In our example, after sufficient training, the model is expected to learn the splitting ratios in Figure 2(c) (the unique global optimum). Indeed, *DOTE*, which is a manifestation of this approach, quickly converges to this global outcome.

Exploiting convexity/concavity for direct optimization of splitting ratios via gradient descent. A key insight is that for classical TE optimization objectives, the function mapping splitting ratios to *expected* performance scores satisfies desirable properties, namely, *convexity/concavity*. This facilitates utilizing elegant direct optimization methods, like (stochastic) gradient descent, circumventing explicit demand prediction.

To illustrate this, we consider the classical TE objective of minimizing maximum-link-utilization (MLU). We visualize in Figure 2(d) the impact of different choices of splitting ratios on MLU, i.e., the maximum ratio, across all network links, between the traffic traversing a link and the link capacity. x-axis values specify the fraction of A's traffic sent on the direct path (A,D). Since A only has two available paths, this value also uniquely determines the fraction of A's traffic sent on the indirect path (A,C,D). Similarly, y-axis values specify the fraction of B's traffic sent on (B,D) and so also on (B,C,D). z-axis values represent the *expected* MLU for different choices of splitting ratios for A (x-axis) and B (y-axis) for the underlying demand distribution described above. For instance, the scenario where A and B send all of their traffic on (A,D) and (B,D), respectively, is captured by $w_{(A,D)}=1$ (x-axis) and $w_{(B,D)}=1$ (y-axis), and the derived expected MLU is $\frac{5}{3}$ (z-axis). Indeed, in this scenario, regardless of which of the two demand combinations is realized, the traffic injected into either link (A,D) or link (B,D) will be $\frac{5}{3}$x its capacity. The *unique* global minimum for MLU, in which no link capacity is exceeded, is achieved for $w_{(A,D)}=0.6$ and $w_{(B,D)}=0.6$ (the red dot in Figure 2(d), which corresponds to the splitting ratios in Figure 2(c)).

As seen in Figure 2(d), the expected MLU exhibits a desirable structure—*convexity* in the traffic splitting ratios. This suggests the following procedure for converging to the optimum: start with *arbitrary* splitting ratios, and adapt the splitting ratios in the direction of the steepest slope of the (ex-

pected) MLU (i.e., the opposite direction of the *gradient* with respect to the splitting ratios) until converging to the global minimum. We show (in §3.3) that the convexity of the expected MLU extends to *any* network topology, *any* choice of network paths (tunnels), and *any* underlying demand distribution, and so, this elegant optimization procedure is guaranteed to converge to the global optimum in general.

2.3 TE as Stochastic Optimization

How to estimate the gradient of the *expected* MLU? Executing gradient descent on the *expected* MLU requires repeatedly evaluating the gradient for different traffic splitting configurations. However, *exact knowledge of the gradient is impossible without exact knowledge of the underlying demand distribution*. Once again, the specific structure of the TE setting gives rise to opportunities for effective optimization. We show how the gradient can be closely approximated from data samples of past realizations of the demands. Our approach builds on the following two observations that, while illustrated using our toy example, generalize to arbitrary network topologies, tunneling schemes, and distributions over traffic demands (see §3).

- **For any realized demand-combination, the MLU gradient *with respect to these specific demands* can be expressed *in closed form*.** Suppose that the realized demands in our simple example are $\frac{5}{3}$ for A and $\frac{5}{6}$ for B. The MLU as a function of A's splitting ratios, $w_{(A,D)}$ and $(1-w_{(A,D)})$, and B's splitting ratios, $w_{(B,D)}$ and $(1-w_{(B,D)})$, can be expressed as:

  $$\max\{\frac{5}{3}w_{(A,D)}, \frac{5}{3}(1-w_{(A,D)})+\frac{5}{6}(1-w_{(B,D)}), \frac{5}{6}w_{(B,D)}\}$$

  (i.e., the maximum load across the links (A,D), (C,D), and (B,D), respectively[2]). This representation of the MLU for the realized demands as a convex function of the splitting ratios enables deriving a closed form expression of the (sub)*gradient* of the MLU[3], as shall be discussed in §3.

- **Averaging over the MLU gradients *for past realized demands* closely approximates the gradient of the *expected* MLU.** Exact knowledge of the underlying probability distribution over demands is elusive in most real-world scenarios. Hence, the gradient of the *expected* MLU for a given configuration of splitting ratios cannot be precisely derived. However, this gradient can be well-approximated by averaging over the gradients *for realized demands* at those splitting ratios. In our example, deriving the expected MLU gradient for specific traffic splitting ratios for A and B can be achieved by sampling sufficiently many past realizations of A and B's demand-combinations, deriving the MLU gradient with respect to each such realized demand combination (at these splitting ratios), and averaging over these.

[2] Observe that the load on (A,C) and (B,C) is always dominated by the load on (C,D), and so we disregard these links.

[3] Note that even though this function is not differentiable for all inputs due to the maximum operator, the subgradient always exists and can be explicitly derived.

Why is reinforcement learning (RL) *not* a good fit? (Deep) RL methods have been applied to many networking domains, including routing [54]. Similarly to *DOTE*, RL approaches to TE also replace explicit demand prediction with end-to-end optimization, mapping recent traffic demands to TE configurations [54]. However, while RL can be applied to essentially any sequential decision making context, RL suffers from higher data-sample complexity, notorious sensitivity to noisy training, and a brittle optimization process that necessitates painstakingly sweeping hyperparameters [21]. A key observation underlying *DOTE* is that WAN TE exhibits a desirable structure that gives rise to opportunities for much simpler and more robust optimization, rendering RL an "overkill".

2.4 Harnessing Deep Learning

In our simple example, traffic demands were repeatedly drawn from the *same* probability distribution. Real-world traffic exhibits intricate temporal (hourly, diurnal, weekly), and other, patterns. To pick up on such regularities, the TE system could take into account the recent history of observed traffic demands (e.g., traffic demands from the last hour). However, there are infinitely many possible recent histories of traffic demands the TE system might observe. To address this, *DOTE* trains a deep neural network (DNN) to *approximate the optimal mapping from traffic histories to TE configurations*, exploiting the capability of DNNs to automatically identify complex patterns in large, high-dimensional data (§3.4). *DOTE* builds on recent developments in large-scale optimization, namely, the ADAM stochastic gradient descent optimizer [30], to accommodate efficient training on extensive empirical data (10s of thousands of traffic demand snapshots in our experiments).

3 Direct Optimization for TE (*DOTE*)

Below, we present our model for WAN TE with uncertain traffic demands, which extends the classical WAN TE model. We then delve into the the *DOTE* stochastic optimization framework, provide theoretical guarantees, and discuss how *DOTE* can be implemented in practice.

3.1 Modeling WAN TE

Network. The network is modeled as a capacitated graph $G=(V,E,c)$. V and E are the vertex and edge (link) sets,

respectively, and $c : E \to \mathbb{R}^+$ assigns a capacity to each edge.

Tunnels. Each source vertex s communicates with each destination vertex t via a set of network paths, or "*tunnels*", P_{st}.

Traffic demands. A *demand matrix* (DM) D is an $|V| \times |V|$ matrix whose (i, j)'th entry $D_{i,j}$ specifies the traffic demand between source i and destination j.

Optimization objective. To simplify exposition, we first describe *DOTE* for the case of one classical TE objective: minimizing maximum-link utilization (MLU) [9, 13, 17]. We discuss other optimization objectives (maximum network throughput and maximum-concurrent-flow) in §3.5.

TE configurations. We focus on how traffic should be split across a *given* set of tunnels so as to achieve the optimization objective. *DOTE* is compatible with any tunnel-selection method. We discuss an extension that incorporates data-driven tunnel selection in §5.

Given a network graph and demand matrix, a *TE configuration* $\mathcal{R}$ specifies for each source vertex s and destination vertex t how the $D_{s,t}$ traffic from s to t is split across the tunnels in P_{st}. Thus, a TE configuration specifies for each tunnel $p \in P_{st}$ a value x_p, where x_p is the fraction of the traffic demand from s to t forwarded along tunnel p (and so $\sum_{p \in P_{st}} x_p = 1$).

Given a demand matrix D and TE configuration $\mathcal{R}$, the total amount of flow traversing edge e is $f_e = \sum_{s,t \in V, p \in P_{st}, e \ni p} D_{s,t} \times x_p$. The objective is minimizing the maximum link utilization induced by $\mathcal{R}$ and D, $\max_{e \in E} \frac{f_e}{c(e)}$, which we will refer to as MLU and represent as $\mathcal{L}(\mathcal{R}, D)$. WAN operators seek to reduce the MLU to keep more headroom open for unplanned failures and traffic spikes. Typically, operators spend to increase link capacities when MLU exceeds a threshold value, and so reducing MLU can reduce CAPEX [14, 27].

In this work, we aim to select TE configurations without a priori knowledge of the traffic demands. To do so, we augment the above model as follows:

WAN TE under traffic uncertainty. Time is divided into consecutive intervals, called "epochs", of length δ_t. δ_t is determined by the network operator (e.g., at some large service providers [22, 24], δ_t is a few minutes). At the beginning of each epoch t, the TE configuration $\mathcal{R}^{(t)}$ for that epoch is decided based only on the history of *past* demand matrices and TE configurations. We also assume that the demand matrix is fixed within an epoch and can be approximately estimated after the fact.[4] Such periodic changes to TE configuration reflect the current practice in private WANs [22–24].

After selecting the TE configuration $\mathcal{R}^{(t)}$ for epoch t, the demand matrix D_t is revealed. To minimize MLU, the goal for direct optimization is to devise a *TE function* $\pi(D_{t-1}, \ldots D_1)$ that, for every $t > 0$, maps the history of DMs from the previous $t-1$ time epochs to a TE configuration $\mathcal{R}^{(t)}$ for the upcoming time epoch t so as to minimize $\frac{1}{T}\Sigma_{x=1}^{t} \mathcal{L}(\mathcal{R}^{(x)}, D_x)$, where T represents the length of time in which TE configurations are computed according to π.

To reason about WAN TE in the presence of traffic uncertainty, we assume that the demand matrix D_t at each epoch t is generated from some probability distribution. We also make the following two assumptions, which are fundamental to any data-driven approach to WAN TE. First, we assume that there is some sufficiently large $H > 0$ such that the finite window of H recent historical observations of DMs is sufficient for informing the decision of the next TE configuration. (Our empirical results in §4 suggest that $H = 12$ suffices for attaining high performance on our datasets.) Formally, we model the demand matrix D_t as generated according to an unknown H-Markov process with transition probabilities such that $P(D_t | D_{t-1}, \ldots, D_{t-H}) = P(D_t | D_{t-1}, \ldots, D_1)$. Second, we assume that the probability of observing a particular sequence of H DMs in the training data and during real-time system execution is the same. This formally translates to the Markov process being in a steady state. Let $P(D_{t-1}, \ldots, D_{t-H})$ denote the Markov process' stationary distribution, which determines the probability for any specific H-long recent history of DMs. The *expected* MLU for a TE configuration $\mathcal{R}$ at epoch t is therefore $\mathbb{E}_{D_t}[\mathcal{L}(\mathcal{R}, D_t)]$, where the expectation is with respect to the (unknown) probability distributions $P(D_{t-1}, \ldots, D_{t-H})$ and $P(D_t | D_{t-1}, \ldots, D_{t-H})$ defined above.

3.2 The *DOTE* TE Framework

DOTE leverages stochastic optimization to compute a TE function $\pi_\theta(D_{t-1}, \ldots, D_{t-H})$, parametrized by θ, which maps the H-long recent history of DMs to the TE configuration for the next time epoch, $\mathcal{R}^{(t)}$. If the TE function is sufficiently expressive, there should exist parameters that closely approximate the optimal TE function. As we shall discuss in §3.4, in *DOTE*, π_θ is realized by a deep neural network (DNN), and the parameters θ correspond to the DNN's link weights. We thus consider the optimization problem of seeking parameters θ for which the following expression is minimized: $\mathbb{E}[\mathcal{L}(\pi_\theta(D_{t-1}, \ldots, D_{t-H}), D_t)]$, where the expectation is with respect to choosing t uniformly at random from $\{1, \ldots, T\}$, and the probability distributions $P(D_{t-1}, \ldots, D_{t-H})$ and $P(D_t | D_{t-1}, \ldots, D_{t-H})$ defined above. Observe that by the linearity of expectation and the above equation, $\mathbb{E}[\mathcal{L}(\pi_\theta(D_{t-1}, \ldots, D_{t-H}), D_t)] = \frac{1}{T}\Sigma_{t=1}^{T} \mathbb{E}_{D_t}\left[\mathcal{L}(\mathcal{R}^{(t)}, D_t)\right]$, which is precisely our optimization objective in *DOTE*.

The training data for *DOTE* is a trace of historical DMs, consisting of N sequences of DMs of the form $\left\{D_t^i, D_{t-1}^i, \ldots, D_{t-H}^i\right\}$, where each sequence consists of $H+1$ DMs and captures a specific realization of a H-long history of DMs and the subsequent realized DM. We assume that

[4] For e.g., by sampling ipfix (or equivalent) data at each node in the WAN, as is done in production in SWAN [22] and B4 [24]. This data contains source and destination nodes and volume of bytes exchanged. Alternatively, traditional ISP backbones use network tomography on measured link usage data (see, e.g., [46, 58]).

these N observations of DM sequences are sampled i.i.d. from $t \in [1,\ldots,T]$, $P(D_{t-1},\ldots,D_{t-H})$ and $P(D_t|D_{t-1},\ldots,D_{t-H})$.[5]

DOTE executes *stochastic gradient descent* (SGD) [51] to optimize the parameters θ by sequentially sampling m-sized mini-batches of data, where each data point in the mini-batch is drawn from the data uniformly at random. For each mini-batch of sampled data points, the parameters θ are updated as follows:

$$\theta := \theta - \alpha \frac{1}{m} \sum_{i \text{ in batch}} \nabla_\theta \mathcal{L}(D^i_t, \pi_\theta(D^i_{t-1},\ldots,D^i_{t-H})),$$

where α is a step size parameter and $\nabla_\theta \mathcal{L}(D^i_t, \pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$ is the gradient of the loss function with respect to θ. Our realization of stochastic optimization in *DOTE* follows the ADAM [30] method, which incorporates momentum and an adaptive step size.

A closer look at *DOTE*'s parameter update step. Recall that our objective is to reach a performant TE configuration with respect to the *expected* loss (MLU). The success of *DOTE*'s SGD is thus crucially dependent on *DOTE*'s ability to well-approximate the gradient with respect to the expected loss. Unfortunately, in most real-world TE environments, exact knowledge of the underlying distribution over traffic demands is unattainable. To address this, *DOTE*'s parameter update step (see above) incorporates the expression $\frac{1}{m}\sum_{i \text{ in batch}} \nabla_\theta \mathcal{L}(D^i_t, \pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$. As discussed above, each sequence of $H+1$ demand matrices $\{D^i_t, D^i_{t-1},\ldots,D^i_{t-H}\}$ in the batch is assumed to be independently drawn from the underlying stationary distribution of the Markov process. Hence, $\frac{1}{m}\sum_{i \text{ in batch}} \nabla_\theta \mathcal{L}(D^i_t, \pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$ is an *unbiased* estimate of the gradient of the expected loss, and closely approximates the gradient of the expected loss for a large enough m. Approximating the gradient of the expected loss in this manner is termed Sample Average Approximation (SAA) in stochastic optimization literature [51]. Relying on unbiased stochastic gradients for SGD guarantees convergence to a *global* optimum with respect to the *expected* loss [49] when the loss function is concave (as in our context, see §3.3).

We are left with the challenge of deriving $\frac{1}{m}\sum_{i \text{ in batch}} \nabla_\theta \mathcal{L}(D^i_t, \pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$. An important technical observation is that each data point i in the batch, $\mathcal{L}(D^i_t, \pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$ is a composition of *differentiable* computations. *DOTE* capitalizes on this for calculating the gradient $\nabla_\theta \mathcal{L}(D^i_t, \pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$ in closed form via backpropagation. We revisit this point in §3.4.

[5] When the data is a long trace of historical DMs, the samples are not necessarily independent. However, we assume that the mixing time of the Markov process is fast enough such that correlations between the data samples are negligible. This is a common assumption in time series prediction.

3.3 Analytical Optimality Results

We prove that, for a *perfectly expressive* TE function, i.e., when the TE function can be *any* mapping from demand histories to TE configurations, and in the limit of infinite empirical data sampled from the underlying Markov process' stationary distribution, *DOTE* attains optimal performance. In practice, we relax both assumptions: in *DOTE*, we sample from a large, but *finite*, dataset of historical demands, and use a *parametric* model (specifically, a neural network) to map from the set of possible histories to valid TE configurations. Our theoretical result below, however, establishes that our approach is *fundamentally sound*, and so high performance in practice can be achieved by acquiring sufficient empirical data and employing a sufficiently expressive decision model (e.g., a deep enough neural network). Our empirical results in §4 corroborate this.

For the sake of analysis, we assume that the set of possible history realizations, which we denote by $\mathbf{H}$, is finite. Let $\pi : \mathbf{H} \to \mathbf{R}$ denote a mapping from history to TE configuration[6]. We consider an idealized stochastic gradient descent (SGD) algorithm that, at each iteration k samples *a single* data point $D_t, D_{t-1},\ldots,D_{t-H}$ from the probability distributions $P(D_{t-1},\ldots,D_{t-H})$ and $P(D_t|D_{t-1},\ldots,D_{t-H})$, and updates $\pi_{k+1} = Proj\{\pi_k - \eta v_k\}$, where $v_k \in \partial\mathcal{L}(\pi_k(D_{t-1},\ldots,D_{t-H}), D_t)$ denotes the subgradient of the objective function, and $Proj$ denotes a projection onto the simplex for each (s,d) pair. The final output after K iterations is $\bar{\pi} = \frac{1}{K}\sum_{k=1}^{K}\pi_k$. Let $\bar{\mathcal{L}}(\pi) = \mathbb{E}[\mathcal{L}(\pi(D_{t-1},\ldots,D_{t-H}), D_t)]$ denote the expected MLU of a TE function, and let $\pi^* \in \arg\min_\pi \bar{\mathcal{L}}(\pi)$ denote the optimal TE function. We prove the following theorem:

Theorem 1. *For any $\varepsilon > 0$, there exists $\eta > 0$ and finite K such that $\left|\mathbb{E}\left[\bar{\mathcal{L}}(\bar{\pi})\right] - \bar{\mathcal{L}}(\pi^*)\right| \le \varepsilon$, where the expectation is w.r.t. the sampling by the algorithm.*

The proof of Thoerem 1, which crucially relies on the convexity of the MLU objective, appears in Appendix B.

3.4 Scalability and Real-World Traffic

Direct TE optimization aims at computing a mapping from the history of recent traffic demands to a TE configuration that optimizes expected performance for the next demands. A key insight is that with real-world traffic, one may expect certain *patterns* in this mapping; for example, if two histories of traffic conditions are very similar, their corresponding optimal TE configurations should also be similar. However, measuring and explicitly quantifying such similarities is nontrivial. Our approach is to exploit deep neural networks, which have demonstrated remarkable success in identifying complex patterns in high dimensional data, for this task.

[6] Note that we dropped the subscript θ in π, as in our analysis we consider the space of all possible TE configurations, and not a specific parametrization.

DOTE employs a DNN to realize the TE function $\pi_\theta(D_{t-1},\ldots,D_{t-H})$. Specifically, *DOTE*'s DNN maps an input of H (12 in our experiments) most recent DMs into an output vector specifying the splitting ratios across tunnels for all source-destination pairs. In our implementation of *DOTE*, we use the popular Fully Connected DNN architecture. See Appendix E for a formal exposition of how the DNN's output and the realized DM are fed into the loss function to derive the induced MLU. Importantly, the sequence of steps for mapping the DNN output to the MLU value $\mathcal{L}(D^i_t,\pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$ involves only *differentiable* computations; the loss as a function of the TE configuration is a composition of a max and a linear function, and the neural network is differentiable by design. Hence, the gradient $\nabla_\theta\mathcal{L}(D^i_t,\pi_\theta(D^i_{t-1},\ldots,D^i_{t-H}))$ can be calculated in closed form via backpropagation. In our implementation, the Pytorch [41] auto-differentiation package is used to calculate the gradients.

3.5 On Maximum and Concurrent Flow

We next explain how *DOTE* extends to two other central TE objectives: maximizing network throughput [18, 22, 24, 25] (maximum multicommodity flow) and maximum concurrent-flow [11, 29, 48].

TE configurations for flow maximization. TE objectives that capture different notions of flow maximization require that the outputs of the TE mechanism satisfy strict capacity constraints. To address this, we revise our definition of TE configuration $\mathcal{R}$ from §3.1: for each source-destination pair $s,t\in V$, $\mathcal{R}$ now specifies (1) traffic splitting ratios x_p over the paths (tunnels) $p\in P_{st}$ (as in §3.1), (2) for each path (tunnel) $p\in P_{st}$, a "*cap*" $\omega_p\geq 0$. ω_p represents the maximum permissible flow between s to t along the path p (enforced via rate limiting). $\mathcal{R}$ must satisfy that no link capacity is exceeded (*regardless of the realized demands*), i.e., that for each link $e\in E$, $\Sigma_{s,t\in V,p\in P_{st},e\ni p}\omega_p\leq c(e)$. A TE configuration $\mathcal{R}$ and demand matrix D induce, for each tunnel p a flow $f_p(\mathcal{R},D)=\min\{x_p\times D_{s,t},\omega_p\}$. The total flow between s and t is thus $f_{st}(\mathcal{R},D)=\Sigma_{p\in P_{st}}f_p(\mathcal{R},D)$.

The maximum-multicommodity-flow and maximum-concurrent-flow objectives. In maximum-multicommodity-flow [18, 22, 24, 25], the performance objective $\mathcal{L}(\mathcal{R},D)$ is to compute, for a given demand matrix D, a TE configuration $\mathcal{R}$ that maximizes the expression $\mathcal{L}(\mathcal{R},D)=\Sigma_{s,t\in V}f_{st}(\mathcal{R},D)$ (the total network throughput). For a TE configuration $\mathcal{R}$ and demand matrix D, let $\alpha(\mathcal{R},D)$ denote the maximum value $\alpha\in[0,1]$ for which at least an α-fraction of each $D_{s,t}$ is routed *concurrently*, i.e., such that for all $s,t\in V$, $f_{st}(\mathcal{R},D)\geq\alpha D_{s,t}$. The goal in maximimum-concurrent-flow is to compute, for an input DM D, the TE configuration $\mathcal{R}$ for which $\mathcal{L}(\mathcal{R},D)=\alpha(\mathcal{R},D)$ is maximized. Relative to maximum-multicommodity-flow above, the maximum-concurrent-flow objective enhances fairness. Practical TE systems [22, 24] use a sequence of optimizations wherein they employ different objectives for different priority classes. For example, they may use maximum-multicommodity-flow or minimizing MLU for high priority traffic and maximum-concurrent-flow for scavenger-class traffic.

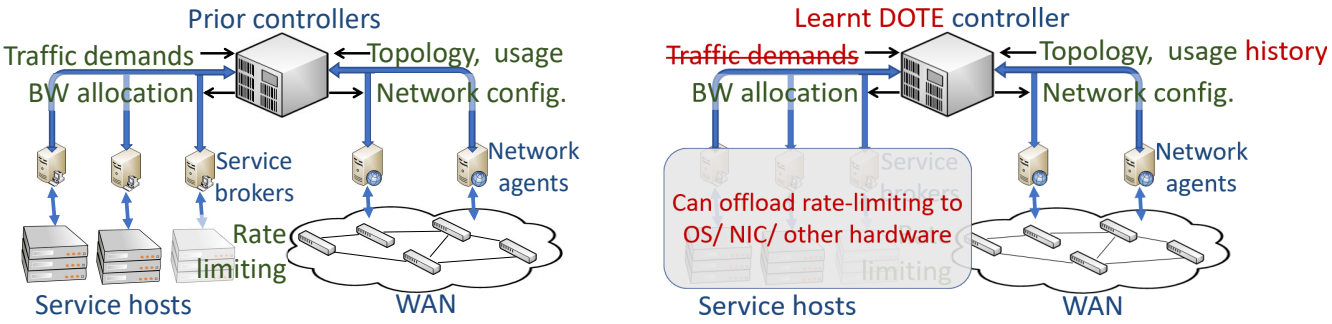
***DOTE* for maximum-multicommodity-flow and maximum-concurrent-flow.** Adapting *DOTE* to the above two flow-maximization objectives is accomplished along the lines described in §3.4. In particular, a DNN is again utilized to map the recent observations of DMs to the next TE configuration. Recall from the above discussion that the (revised) TE configuration consists of both traffic splitting ratios across tunnels and a "flow cap" for each tunnel. In our design, the DNN outputs $w_p\geq 0$ for each tunnel p. The w_p's are used to derive traffic splitting ratios and flow caps as follows. We set $\omega_p=\frac{w_p}{\gamma}$, where $\gamma=\max\left(\max_{e\in E}\frac{\Sigma_{p:e\in p}w_p}{c(e)},1\right)$. Observe that this guarantees that no link capacity can be exceeded even if each tunnel p carries its maximum permissible flow ω_p (i.e., that $\Sigma_{s,t\in V,p\in P^e_{st}}\omega_p\leq c(e)$). We then set the traffic split share on tunnel p to simply be its proportional weight: $x_p=\frac{\omega_p}{\Sigma_{q\in P_{st}}\omega_q}$. Since the objective is now *maximizing* a performance metric, *DOTE* now involves stochastic gradient *ascent*.

Optimality via stochastic quasi-concave optimization. In Appendix B, we prove the analogues of Theorem 1 for maximum-multicommodity-flow and for maximum-concurrent-flow, establishing *DOTE*'s optimality for these two objectives. Similarly to Theorem 1 (for MLU), this implies that with sufficient training data and a sufficiently expressive decision model, *DOTE* attains near-optimal TE configurations. Our evaluation results for maximum-multicommodity-flow and for maximum-concurrent-flow exemplify this (§4.3).

Our proofs for maximum-multicommodity-flow and for maximum-concurrent-flow are considerably more subtle than that of Theorem 1, as both objectives are not concave (the analogue of convexity for maximization problems). Instead, we show that the average maximum-multicommodity-flow / maximum-concurrent-flow score of a TE configuration over any set of DMs is *quasi*-concave. This result, which may be of independent interest, allows us to leverage the analytical arguments in [20] to show convergence of a suitable stochastic gradient ascent algorithm to the global optimum, and bound the number of required iterations.

3.6 Realizing *DOTE*

Figure 3 illustrates key differences between *DOTE* and prior software-defined WAN TE schemes. One key difference is the use of historical traffic demands and a learnt controller instead of running an optimization solver, leading to substantial decrease in deployment overheads and runtimes. In particular, bandwidth brokers are no longer needed to estimate application demands. Furthermore, rate allocations can, if necessary, be enforced by piggy-backing on novel traffic



(a) Architecture of prior SD-WAN TE schemes [22, 24].

(b) *DOTE* with differences shown in red.

Figure 3: Illustration of the key differences from previous SD-WAN TE schemes.

shaping techniques that are deployed in modern cloud servers at the OS-level as well as in NIC/FPGA offloads [1,12,43,47].

Training *DOTE*. Since *DOTE*'s decision model is trained *offline* on historical data, its operational model can be periodically replaced by a model trained in the background on more recent and up to date data, to gracefully adapt to *planned* changes in WAN topology (adding capacity, planned addition/removal of nodes or links) and to temporal drifts in traffic demand distributions. Our evaluation results (§4) indicate that *DOTE* produces high performance TE configurations even weeks after being deployed, and even if the network topology changes during this time (e.g., due to failures). This provides ample time for training substitute TE functions (a process that requires less than a day on large networks for our empirical datasets without code and hardware optimizations).

Handling network failures. Tunnelling protocols (e.g., MPLS) identify tunnels with failed nodes/links. A traditional approach in TE to rerouting traffic around failed tunnels is to let traffic sources redistribute traffic proportionally among their remaining tunnels [22, 24, 39, 52].[7] We incorporate this simple approach into *DOTE* and evaluate its effectiveness in §4), showing that it achieves high resiliency to failures. We discuss other possible approaches in §5.

4 Evaluation

Using actual traffic demands from three different production WANs (Abilene, GEANT, and Microsoft's WAN), we ask the following questions: (1) How does *DOTE* compare against an omniscient oracle with perfect knowledge of future demands? (2) How does *DOTE* compare with state-of-the-art prediction-based TE [4, 22, 24, 36, 40], demand-oblivious TE [8, 32], and RL-based TE [54]? (3) Can *DOTE* support different TE objectives (e.g., MLU [8], maximum-multicommodity-flow [4, 22, 24])? (4) How long does *DOTE* take to train and to apply online at each solver activation? (5) How does *DOTE* perform under network faults and drift in traffic patterns?

[7] Traffic split $(0.6, 0.3, 0.1)$ becomes $(0, 0.75, 0.25)$ if the first path goes down.

	#Nodes	#Edges	Length	Granularity
Abilene	11	14	4.5months	5 min.
GEANT	23	37	4 months	15 min.
PWAN	O(100)	O(100)	O(1) months	minutes
$PWAN_{DC}$	O(10)	O(10)	O(1) months	minutes
GtsCe	149	193	Synthetic	
Cogentco	197	243		
KDL	754	895		

Table 1: Datasets used to evaluate *DOTE*

4.1 Methodology

Datasets: Data-driven TE is best evaluated on *real-world* datasets; we use the production topology and the traffic demands from GEANT [53], Abilene [3], and PWAN, a private WAN at Microsoft. Traffic traces were collected at a few-minute granularity over several months. We also use three topologies (GtsCe, Cogentco and KDL) from Topology Zoo [31] with synthetic traffic (generated using the gravity model [8, 45]). Table 4 lists the topology sizes and traffic demands. Nodes in these WAN topologies are datacenters, edge sites, or peering points. Traffic on PWAN includes both traffic between datacenters and traffic to/from end users. To better understand how *DOTE* performs for each traffic class, we consider a subset–PWAN_{DC}–which only contains large datacenters and the traffic between them. For each WAN, we use the earlier 75% of demand matrices (DMs) for training and the later 25% DMs as the test set.

Tunnel choices are k-shortest-paths, edge-disjoint paths, and SMORE trees. More specifically, we use (1) Yen's algorithm for k-shortest-paths, with $k = 8$ per commodity (pair of nodes), (2) edge-disjoint shortest-paths where, for each commodity, we iteratively compute a shortest-path in the network and remove all links on that path from consideration until no more paths exist for that commodity, and (3) tunnels from SMORE [37] generated using Yates [36].

Comparables to *DOTE* include: (1) **Omniscient oracle**, which is an optimization with perfect knowledge of future demands and bounds the quality of *any* WAN TE scheme. (2) **Demand-prediction-based TE** methods [4, 22, 24, 36, 40], which are in production today [22, 24]. We consider a rich

WAN	Online Lat. (s)		Precomp. Lat. (s)		
	DOTE	LP	*DOTE*	COPE	Oblivious
Abilene	0.0005	0.02	1800	180	1
PWAN_{DC}	0.003	0.05	1200	7200	15
Geant	0.002	0.04	2400	10800	180
PWAN	0.2	1	36000	>345600	~ 86400

Table 2: Comparing the online latency (to compute a TE configuration for a demand matrix) as well as the precomputation latency (to train models, to compute demand-oblivious configurations, etc.) for various TE schemes. 8 shortest paths are used per demand across all WANs and TE schemes.

collection of possible predictors of future demands: linear regression, ridge regression, random forest, DNN models, and autoregressive models (§C). (3) **RL-based** WAN TE [54], which leverages a neural network of the same size as *DOTE*'s (see below). (4) **Demand-oblivious TE** [8], which optimizes the *worst-case performance* over *all* traffic demands. (5) **SMORE** [37], which picks source-rooted trees for worst-case demands but adapts splitting ratios over the chosen trees based on *predicted* future demands. (6) **COPE** [55], which enhances demand-oblivious schemes by also optimizing over a set of predicted traffic demands.

Metrics: Our TE quality metric is the ratio between the value obtained by the evaluated TE scheme and the performance obtained by the omniscient oracle, which has perfect information about future traffic demands. We consider three TE objectives: minimize maximum link utilization (MLU) [8, 14, 27], maximize multicommodity flow [4, 22, 24, 37] and maximize concurrent-flow [11, 29]. Note that this ratio is ≥ 1 for MLU (because lower max-link utilization is better) and ≤ 1 for the other metrics (because carrying more flow is better). We refer to the relative gap from 1 as the *optimality gap*. We also measure the runtimes (latency) of the evaluated TE schemes on the same physical machine.[8]

DNN architecture: Unless otherwise specified, results for *DOTE* use five fully connected NN layers with 128 neurons each and $ReLU(x)$ activation except for the output layer which uses $Sigmoid(x)$. For different TE objectives, *DOTE* uses a similar architecture with small changes. We chose this architecture because it empirically outperformed other investigated architectures.

Infrastructure and code: We ran our experiments in cloud VMs and made use of cloud ML training systems. To enable further research, we have released our code at [2].

Fault model: To examine TE behaviour under network faults, we randomly bring down a certain number of links (e.g., 1 to 20 while ensuring network is not partitioned), and compare the performance of *DOTE* (see *DOTE*'s failure-recovery scheme at the end of §3.6) and alternatives with an omniscient oracle with perfect knowledge of both future failures and future traffic demands.

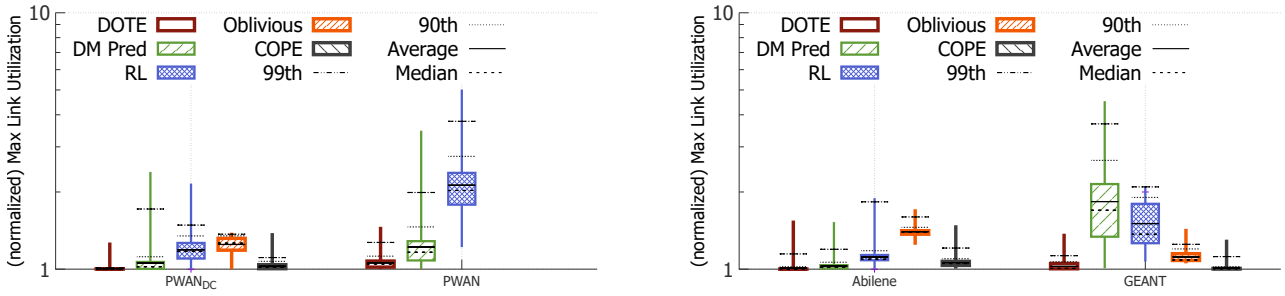
[8] VM with 8 vCPUs and 256GB RAM.

4.2 Comparing *DOTE* with Other TE Schemes

TE quality. Figure 4 compares *DOTE* with the other TE schemes described in §4.1, with the exception of SMORE (to be discussed in §4.3). The values plotted here are the maximum link utilization (MLU) normalized by that of the ominiscient oracle with perfect knowledge of future demands. The figure shows results on four different topologies. Each candlestick shows the distribution of MLUs achieved on the various demand matrices with the boxes ranging from 25th to 75th percentile and the whiskers going from minimum to maximum value. The figure also plots values achieved at various other percentile values. We note a few findings.

- First, optimizing for *predicted* demands can lead to poor TE quality (see results for GEANT and PWAN). Note that the y axis is in log scale. A value of $y = 2$ indicates that the link most utilized by the TE scheme is twice as utilized as the most utilized link in the optimal solution (produced by the oracle). Optimizing with respect to predicted demands performs well only on Abilene and PWAN_{DC}, where the traffic demands are predictable. These results are for a linear-regression-based predictor that outperforms all other considered predictors on our real-world traffic datasets (see Appendix C).
- Next, we observe that the RL-based TE scheme [54] has extremely poor TE quality even on Abilene. This could be due to the infamous training complexity of RL.
- Third, demand-oblivious TE [8] results in somewhat decent TE quality on GEANT but not on any of the other WANs. This could be because optimizing *worst-case* performance across *all* possible demands fails to take advantage of the specific characteristics of real-world traffic demands.
- Fourth, COPE [55], which explicitly accounts for historically observed demands, significantly outperforms demand-oblivious TE. The key issue with COPE is its extremely high runtimes. Our analysis (see Table 2 and discussion below) suggests that COPE's applicability does not extend beyond topologies with tens of nodes.
- Finally, note that *DOTE* achieves TE quality that is almost always significantly better than the alternatives' and nearly as good as the omniscient oracle's. The difference in TE quality is especially stark at the higher percentiles. Relative to the compared TE schemes, *DOTE* offers MLU up to 25% better at the median and 170% better at the 99th percentile.

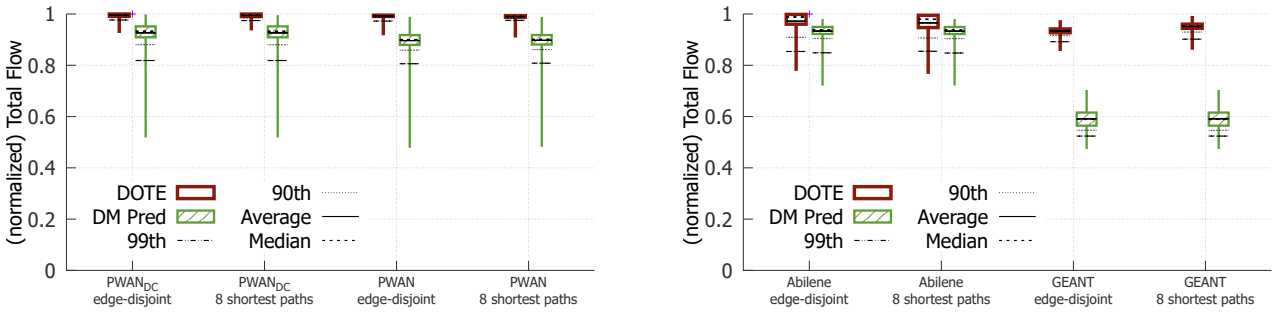
Runtimes. Table 2 presents a comparison of runtimes across TE schemes. The table presents the latency of applying each TE scheme to a new demand matrix and, wherever appropriate, the required precomputation time. Demand-oblivious schemes [8] and COPE [55] do not change the TE configuration online but involve very long precomputation latency and require very large memory. *DOTE* performs both precompu-



(a) $PWAN_{DC}$ & PWAN

(b) Abilene & GEANT

Figure 4: TE quality when aiming to minimize the maximum link utilization with 8 shortest paths per demand. Candlesticks depict results across hundreds of demands; the boxes are from the 25th to the 75th percentile, the whiskers range from min to max value, dashed lines capture other percentiles of interest. *DOTE* achieves much lower MLU compared to the alternatives.



(a) PWAN_{DC} & PWAN

(b) Abilene & GEANT

Figure 5: TE quality when aiming to maximize total flow with two different tunnel choices.

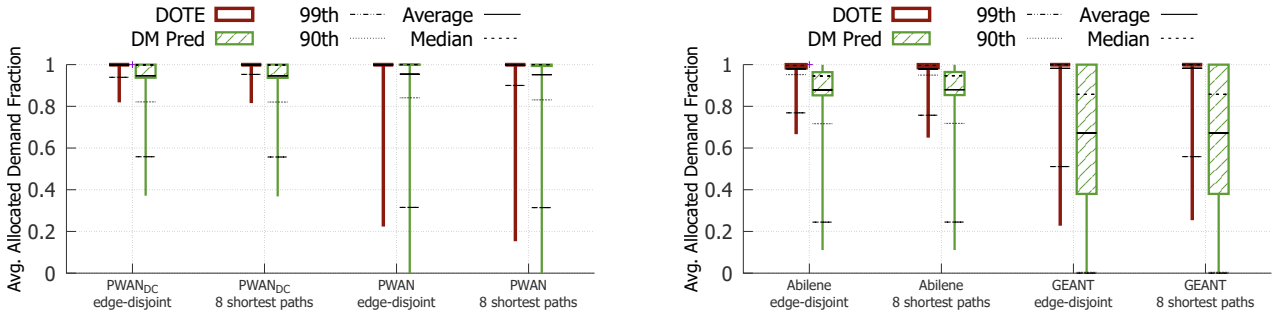
tation on historical demands (training the DNN) and online computation (invoking the DNN). SMORE's online computation involves solving an LP to optimize over predicted demand matrices and so its latency is roughly as high as the LP's latency in the table. To compute Racke's routing trees, SMORE requires several hours on the larger topologies.

The table shows that *DOTE*'s inference time is faster than the latency of using LPs to optimize over one (predicted) DM. The LP's latency is on par with results in recent studies [4, 40]. *DOTE*'s online computation is short because it is effectively a few matrix multiplications.[9] LP computation latency increases super-linearly with the network size and prior work notes that solver times can exceed several minutes on networks with thousands of nodes and edges [4, 40]; *DOTE*'s inference latency on large WANs, such as KDL (see Table 4), is still within a few seconds. *DOTE*'s training time is less than 12 hours for PWAN and can be accelerated using standard methods (e.g., by parallelization, SIMD and other model training enhancements).

COPE's precomputation latency is a few orders of magnitude higher than that of the demand-oblivious TE, which is, itself, a couple orders of magnitude higher than that of prediction-based TE. COPE also has much higher memory requirements (over 256GB on PWAN); in fact, on PWAN, COPE did not finish pre-computation even after four days on a 8-core VM with 256GB running Gurobi [19] vers. 9.1, and hence Figure 4 includes no results for COPE on PWAN. To understand COPE's runtime complexity better, we ran it on WAN topologies from Topology Zoo [31] that are larger than GEANT and PWAN_{DC} but smaller than PWAN. On Janet-Backbone which has 29 nodes and 45 edges, COPE ran for 1.5 hours and on SurfNet (50 nodes, 68 edges), COPE did not finish even in 10 hours. These results suggest that COPE is inapplicable to large WANs.[10]

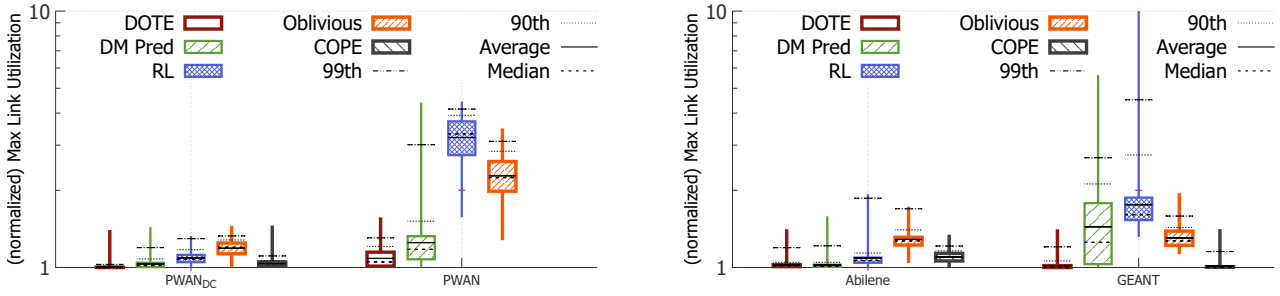
[9] Input is 12 demand matrices and output is splitting ratios or one double per tunnel per demand. On the large PWAN network, both the input and output are a few tens of MBs.

[10] Per Table 1 in [55], the previously published results on COPE are on much smaller topologies than considered here.



(a) PWAN_{DC} & PWAN (b) Abilene & GEANT

Figure 6: TE quality when aiming to maximize the concurrent flow for two different tunnel choices. For each demand matrix, we compute the fraction of demand satisfied for each source and destination, and sort these values into a vector. Across many hundreds of demand matrices, the candlesticks plot the average over all such allocation vectors. Note: allocating more flow is better. The box in each candlestick is the 25th and 75th percentile (fractional allocation) and the whiskers go from min to max value.



(a) PWAN_{DC} & PWAN (b) Abilene & GEANT

Figure 7: TE quality when aiming to minimize MLU with all possible edge-disjoint paths.

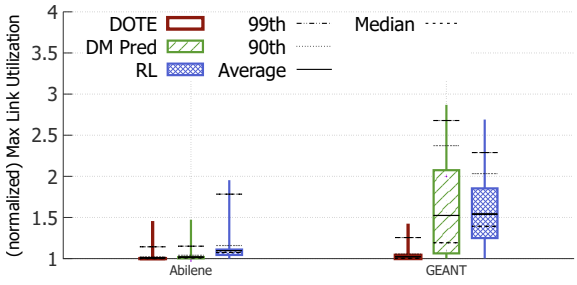


Figure 8: TE quality when aiming to minimize MLU with routing trees chosen by SMORE.

4.3 Generalizing to Other TE Objectives and Tunnel Choices

Here, we present results for two additional TE objectives – maximizing multi-commodity-flow and maximizing concurrent flow– as well as two other choices for tunnels.

Note that some of the compared alternatives to *DOTE*, namely, demand-oblivious TE [8] and COPE [55], do not readily apply to these TE objectives (as both build on results from oblivious routing theory that provide provable guarantees for MLU minimization), and it is not clear how to extend them to other objectives. Our evaluation of *DOTE* for these metrics is therefore restricted to benchmarking against the omniscient oracle and prediction-based TE.

Maximizing Total Flow: Figure 5 compares *DOTE* with prediction-based TE on all four WANs for two different tunnel choices when the TE objective is to carry as much total flow as possible while respecting capacity constraints. Observe that *DOTE* carries substantially more flow and closely approximates the TE quality of the omniscient oracle. As before, the gap between *DOTE* and prediction-based TE is larger on WANs where demands are less predictable (i.e., all WANs but Abilene) and at the higher percentiles. Generally, *DOTE* may be able to carry 10% to 20% more flow.

Maximizing Concurrent Flow: Figure 6 compares *DOTE* with the omniscient oracle and prediction-based TE when the TE objective is to maximize the minimum fraction of demand satisfied across all demands. Observe that *DOTE* fully allocates almost all of the demands (the upper candlesticks are at $y = 1$), whereas prediction-based TE allocates a smaller fraction of the demanded volume for many more demands.

Tunnel choice does not qualitatively change our results for TE performance; contrast Figure 7 and Figure 8 with Figure 4. Note that when using Racke's routing trees (as in SMORE) prediction-based TE coincides with SMORE.

4.4 Coping with Network Failures

Figure 9 shows how *DOTE* performs, in terms of MLU, when different numbers of (randomly chosen) links fail in the PWAN topology. As noted in §3.6, *DOTE* assumes that source nodes (or tunnel heads [44]) identify tunnels that fail and rebalance traffic proportionally among the surviving tunnels. The figure compares *DOTE* with two variants of prediction-based TE: DM Pred. which, similar to *DOTE*, has no a priori knowledge of future traffic demands or the link faults, and FA DM Pred. which is identical to DM Pred. except that it is fault-aware, i.e., knows the links that will fail. Our quality metric is still normalized MLU except that we now normalize based on an omniscient oracle that has perfect knowledge of *both* future traffic demands *and* the failures.

Our results show that *DOTE* outperforms both demand-prediction-based TE (DM Pred.) and demand-prediction-based TE with oracle access to future failures (FA DM Pred.) for many concurrent link failures with different tunnel choices. We interpret this result as indicating that the error in demand predictions weights more heavily on attaining a good TE objective than the confusion induced by these link failures. Our results on other topologies (Abilene, GEANT, and PWAN_{DC}) and for the maximum-multicommodity-flow objective show a similar trend (Figure 13 and Figure 14).

4.5 Robustness to Traffic Noise and Drift

Robustness to unexpected traffic changes. To assess *DOTE*'s robustness to noisy traffic, we evaluate *DOTE* on the GEANT, Cogentco, and GtsCe WANs [31], where each demand in the realized DM is independently multiplied by a factor chosen uniformly at random from $[1-\alpha, 1+\alpha]$ for $\alpha \in \{0.1, 0.25, 0.35\}$. Our results (see §D) show that under such traffic perturbations, the distance, in terms of MLU, from the omniscient oracle remains low across all evaluated WANs (e.g., 2%, 2.9%, and 3.8% for $\alpha = 0.1, 0.25, 0.35$ for GEANT with edge-disjoint tunnels).

Robustness to natural traffic drift. We investigate to what extent the quality of *DOTE*'s TE configurations deteriorates when *DOTE* is not frequently retrained. We quantify the distance from the omniscient oracle, in terms of both MLU and maximum-multicommodity-flow, of the average *weekly* value achieved by *DOTE* on the Abilene and GEANT WANs over 4 consecutive weeks (without retraining *DOTE*). See Table 3 and Table 4 in the Appendix. Our results show that while the distance from the optimum increases over time, in general, *DOTE* remains close to the optimum (within a few % on average) even weeks after the model is trained. This suggests that *DOTE* can provide high quality TE even if it was re-trained once every month. *DOTE*'s training time (see Table 2) allows for much more frequent retraining.

5 Limitations and Future Research

We believe that our investigation of direct optimization for WAN TE has but scratched the surface and outline below current limitations of our approach, as well as intriguing directions for future research.

Extending *DOTE* to support latency-sensitive traffic. To accommodate latency-sensitive traffic, the following strategy (similarly to [34]) could be employed: reserve shortest paths (tunnels) for such traffic and always schedule short/latency-sensitive traffic flows to these paths.

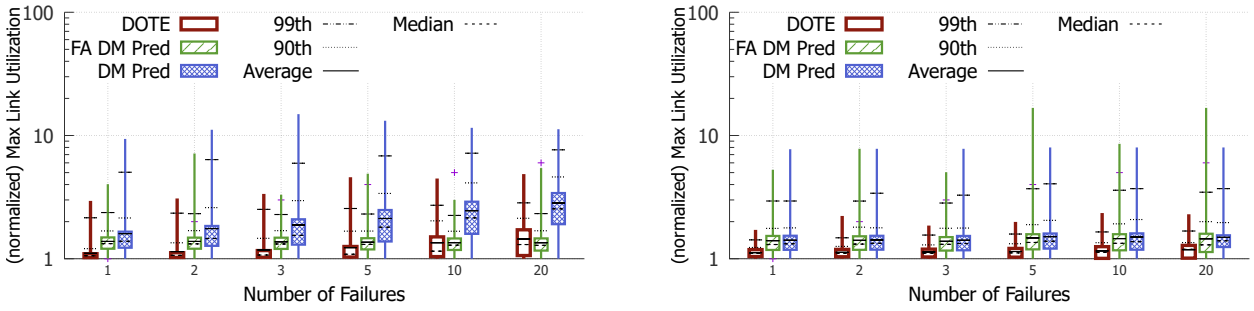
More expressive neural network architectures. Our realization of *DOTE* uses a relatively simple neural network that does not leverage knowledge of the WAN topology. Consequently, the neural network has to (implicitly) learn the network topology during training. Directly incorporating the WAN structure into *DOTE* using Graph Convolutional Networks [56] could potentially lead to faster training and/or better quality solutions.

Extending *DOTE* to incorporate data-driven tunnel selection. Our discussion of *DOTE* assumed an underlying tunnel-selection scheme. *DOTE* can be extended to support *data-driven* tunnel-selection by adding DNN output variables specifying a probability distribution over a finite set of candidate tunnels (e.g., shortest-path, edge disjoint, SMORE). At the beginning of each time epoch, the tunnels to be used in that time epoch would be chosen according to this probability distribution. *DOTE*'s optimality results extend to this setting. We defer a more thorough study of data-driven tunnel selection (e.g., not limited to a finite set of predetermined candidate tunnels) to future research.

Learning to contend with link failures. We described (§3.6) an approach for dealing with link failures in the data plane. An alternative is incorporating fault tolerance into the DNN training process by introducing random link failures.

6 Related Work

(WAN) TE. TE has been extensively studied [5, 7, 10, 11, 14, 16, 22, 24, 26–28, 37, 39, 57, 59] in a broad variety of settings, including legacy networks [13, 17], datacenter networks [6],



(a) 8SP

(b) edge-disjoint

Figure 9: Coping with different numbers of random link failures on PWAN; the candlesticks show the distribution over 1700 different randomly chosen failure cases.

and backbone networks [27]. SDN-controlled WAN TE has also received extensive attention [11, 22, 24, 35, 37–39, 59].

TE via oblivious routing, COPE, and SMORE. Oblivious routing optimizes *worst-case MLU* across *all* possible DMs [8, 9, 42]. Since oblivious routing does not exploit *any* information about past traffic demands, it naturally yields suboptimal solutions [8, 37]. COPE [55] optimizes *MLU* across a set of DMs spanned by previously observed DMs, while retaining a worst-case performance guarantee. Since COPE both extends oblivious routing *and* optimizes over *ranges* of demand matrices, its optimization phase is extremely time-consuming (§4.2). The key conceptual difference between *DOTE* and such "robust TE" schemes is in the goal of the pre-computation. Instead of emitting a single TE configuration that minimizes some cost function (specifically, MLU) over some predetermined set of DMs, DOTE's objective is to identify a mapping from a vector of DMs from the recent past to the next TE configuration. DOTE thus achieves higher flexibility by being able to emit different TE configurations on a case-by-case basis, and is also able to pick up on temporal patterns in traffic demands. SMORE [37] employs Racke's oblivious routing trees [42] to produce static tunnels that are robust to traffic uncertainty, with traffic splitting ratios still optimized with respect to the (inferred/predicted) future traffic demands. Thus, SMORE can be thought of as a instantiation of prediction-based TE.

Online TE [14, 15, 27], wherein traffic configurations (such as splitting ratios) adapt automatically and in short timescales to the observed demands is an enticing design point for TE, but is challenging to achieve. TexCP [27] requires WAN routers to offer novel explicit feedback, while MATE [14] relies on changes in end-to-end latency and hence takes much longer to react and converge and is also less stable [27]. Recently deployed TE schemes [22, 24] (see §2 and Figure 3) are simpler and easier to deploy because they replace such distributed, closed-loop, short-timescale control with centralized, open-loop and periodic adaptation. We view online TE as complementary to *DOTE*; *DOTE* could be used to *periodically* compute a TE configuration while online TE could be *continuously* used in between *DOTE* updates to tweak this TE configuration in response to changes in network conditions.

Reinforcement-learning-based TE. Demand-prediction-based and RL approaches to TE are contrasted in [54] in terms of MLU only on a small network (12 nodes and 32 edges) for *synthetic* traffic patterns and a model of hop-by-hop routing that does not capture routing along tunnels. Our theoretical and empirical results reveal that *DOTE*'s stochastic optimization scheme outperforms both demand-prediction-based and RL-based TE.

Some recent work on TE [4, 40] speeds up the multicommodity flow computations that underpin TE optimization by effectively breaking the large LPs into smaller problems that can be solved in parallel. However, these approaches still rely on predicted demand matrices (unlike *DOTE*). *DOTE* offers an alternate way to speed up TE: replacing the LP solver with invocations of a fairly small DNN. This has the potential to be innately more efficient.

7 Conclusion

We presented a new framework for WAN TE: data-driven end-to-end stochastic optimization using only historical information about traffic demands. Our theoretical and empirical results establish that this approach closely approximates the optimal TE configuration, significantly outperforming previously proposed TE schemes in terms of both solution quality and runtimes.

Acknowledgements: We thank our shepherd, Mojgan Ghasemi, and the NSDI reviewers, for their valuable feedback. We thank Umesh Krishnaswamy, Himanshu Raj and the SWAN team at Microsoft for their help and feedback. Yarin Perry and Michael Schapira were partially supported by BSF grant 2019798 and a grant from Microsoft. Aviv Tamar is funded by ERC grant 101041250.

References

[1] Google cloud armor: Rate limiting overview. https://bit.ly/3TnI1mO.

[2] *Github repo containing our code*. 2022. https://github.com/PredWanTE/DOTE.

[3] Abilene/Internet2. http://www.internet2.edu/.

[4] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 175–200, 2021.

[5] Ian F. Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in sdn-openflow networks. *Comput. Netw.*, 71:1–30, October 2014.

[6] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[7] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *SIGCOMM Comput. Commun. Rev.*, 44(4):503–514, August 2014.

[8] David Applegate and Edith Cohen. Making Intra-Domain Routing Robust to Changing and Uncertain Traffic Demands. In *SIGCOMM*, 2003.

[9] Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan, and Harald Racke. Optimal oblivious routing in polynomial time. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 383–388, 2003.

[10] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.

[11] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019*, pages 29–43, 2019.

[12] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *MICRO*, 2016.

[13] Marco Chiesa, Gábor Rétvári, and Michael Schapira. Lying your way to better traffic engineering. CoNEXT, 2016.

[14] A. Elwalid, C. Jin, S. Low, and I. Widjaja. Mate: Mpls adaptive traffic engineering. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1300–1309 vol.3, 2001.

[15] Simon Fischer, Nils Kammenhuber, and Anja Feldmann. Replex: Dynamic traffic engineering based on wardrop routing policies. In *Proceedings of the 2006 ACM CoNEXT Conference*, 2006.

[16] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM 2000. Nineteenth annual joint conference of the IEEE computer and communications societies. Proceedings. IEEE*, volume 2, pages 519–528. IEEE, 2000.

[17] Bernard Fortz and Mikkel Thorup. Increasing internet capacity using local search. *Computational Optimization and Applications*, 2004.

[18] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652, 2007.

[19] Zonghao Gu, Edward Rothberg, and Robert Bixby. Gurobi Optimizer Reference Manual, Version 5.0. *Gurobi Optimization Inc., Houston, USA*, 2012.

[20] Elad Hazan, Kfir Levy, and Shai Shalev-Shwartz. Beyond convexity: Stochastic quasi-convex optimization. *Advances in neural information processing systems*, 28, 2015.

[21] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[22] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. SIGCOMM, 2013.

[23] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B.,

Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. *SIGCOMM '18*, pages 74–87, 2018.

[24] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. SIGCOMM, 2013.

[25] William S. Jewell. *Multi-commodity Network Solutions*. 1966.

[26] Wenjie Jiang, Rui Zhang-Shen, Jennifer Rexford, and Mung Chiang. Cooperative content distribution and traffic engineering in an isp network. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 239–250. ACM, 2009.

[27] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *SIGCOMM*. ACM, 2005.

[28] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *SIGCOMM*, 2014.

[29] George Karakostas. Faster Approximation Schemes for Fractional Multicommodity Flow Problems. *ACM Trans. Algorithms*, 2008.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization learning. *arXiv preprint arXiv:1412.6980*, 2014.

[31] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 2011.

[32] M Kodialam, T V Lakshman, and S Sengupta. Traffic-oblivious routing in the hose model. *IEEE/ACM Transactions on Networking*, 19(3):774 – 787, 2011.

[33] Igor V Konnov. On convergence properties of a subgradient method. *Optimization Methods and Software*, 18(1):53–62, 2003.

[34] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BlastShield. Technical Report MSR-TR-2021-31, Microsoft Research, 2021.

[35] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Sigcomm '15*, 2015.

[36] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. Yates: Rapid prototyping for traffic engineering systems. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 11:1–11:7, New York, NY, USA, 2018. ACM.

[37] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, Renton, WA, 2018. USENIX Association.

[38] George Leopold. Building Express Backbone: Facebook's new long-haul network. http://code.facebook.com/posts/1782709872057497/, 2017.

[39] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 527–538, 2014.

[40] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[42] Harald Räcke. Minimizing congestion in general networks. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, 2002.

[43] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. Senic: Scalable nic for end-host rate limiting. In *NSDI*, 2014.

[44] E. Rosen, A. Viswanathan, and R. Callon. Multi-Protocol Label Switching Architecture. RFC 3031.

[45] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning. IMW, 2002.

[46] Matthew Roughan, Mikkel Thorup, and Yin Zhang. Performance of estimated traffic matrices in traffic engineering. In *SIGMETRICS*, 2003.

[47] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.

[48] Farhad Shahrokhi and David W. Matula. The maximum concurrent flow problem. *J. ACM*, 37:318–334, 1990.

[49] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[50] Shai Shalev-Shwartz, Ohad Shamir, Nathan Srebro, and Karthik Sridharan. Stochastic convex optimization. In *COLT*, volume 2, page 5, 2009.

[51] Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczynski. *Lectures on stochastic programming: modeling and theory*. SIAM, 2021.

[52] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. Network architecture for joint failure recovery and traffic engineering. In *Proceedings of the 2011 ACM SIGMETRICS Conference*, 2011.

[53] Steve Uhlig, Bruno Quoitin, Jean Lepropre, and Simon Balon. Providing public intradomain traffic matrices to the research community. *SIGCOMM Comput. Commun. Rev.*, 36(1):83–86, jan 2006.

[54] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, 2017.

[55] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. Cope: Traffic engineering in dynamic networks. In *SIGCOMM*, 2006.

[56] Zonghanu Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. arXiv preprint arXiv:1901.00596, 2019.

[57] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing deadlines for inter-data center transfers. *IEEE/ACM Transactions on Networking (TON)*, 25(1):579–595, 2017.

[58] Yin Zhang, M. Roughan, C. Lund, and D.L. Donoho. Estimating point-to-point and point-to-multipoint traffic matrices: an information-theoretic approach. *IEEE/ACM Transactions on Networking*, 13(5):947–960, 2005.

[59] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. ARROW: restoration-aware traffic engineering. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 560–579, 2021.

Appendix

A Predictability of WAN TE Traffic

Figure 10(a) plots the inter-data-center traffic demand between the pair of data centers with the highest average demand over the course of a week. Similarly, Figure 10(b) plots the normalized volume of customer-facing traffic for the pair of nodes with the highest average demand over the course of a week. Demands are shown at 5-minute granularity and are normalized by the peak demand. As shown in Figure 10(a), inter-data-center traffic demands exhibit very distinct diurnal and hourly patterns. Indeed, the figure also presents the predictions of a linear regression model trained on data from the 3 preceding weeks, which takes as input the traffic demands observed in the previous hour (at 5-minute granularity), and outputs the predicted traffic demand for the upcoming 5 minutes. In contrast, the predictions of a linear regressor for customer-facing traffic, as shown in Figure 10(b), are quite often far from the actual traffic demands.

B Analytical Results

B.1 Minimizing Max-Link Utilization

We next prove that, for an infinitely expressive TE function, i.e., when each history of DMs can be *independently* mapped to a TE configuration, and in the limit of infinite empirical data sampled from the underlying Markov process' stationary distribution, *DOTE* attains optimal performance. This establishes that our approach is *fundamentally sound*, and so high performance in practice can be achieved by acquiring sufficient empirical data and employing a sufficiently expressive decision model (e.g., a deep enough neural network).

For the sake of analysis, we make the following simplifying assumptions. We first assume that the set of possible history realizations, which we denote by $\mathbf{H}$, is finite. Let D_{max} denote an upper bound on the maximum traffic demand between a source-destination pair, c_{min} denote the minimum link capacity, and p_{max} denote the maximum number of tunnels interconnecting a source-destination pair. Note that any valid TE configuration specifies, for each source-destination pair, a point in the p_{max}-dimensional simplex (specifying its splitting ratios across at most p_{max} tunnels); let $\mathbf{R}$ denote the space of valid TE configurations. Let $\pi : \mathbf{H} \to \mathbf{R}$ denote a mapping from history to TE configuration. π can be represented as a vector with $|\mathbf{H}| \times n^2 \times (p_{max}-1)$ components.[11] Since each element in this vector is itself a vector in the p_{max}-dimensional simplex, we have that $\|\pi\| \leq \sqrt{|\mathbf{H}| n^2 (p_{max}-1)} \doteq B$, where $\|\cdot\|$ is the Euclidean norm. We make the following observation.

[11] Note that we dropped the subscript θ in π, as in our analysis we consider the space of all possible TE configurations, and not a specific parametrization.

Proposition 1. *The loss function $\mathcal{L}(\pi(D_{t-1},\ldots,D_{t-H}),D_t)$ is convex in π and ρ-Lipschitz, with $\rho = D_{max}/c_{min}$.*

Proof. f_e is, by definition, linear in the traffic splitting ratios and so in π. Since the max is a convex function, we have that $\mathcal{L}$ is convex in π. Similarly, since each component in $\frac{f_e}{c(e)}$ is D_{max}/c_{min}-Lipschitz, the maximum is also D_{max}/c_{min}-Lipschitz. □

We now consider an idealized stochastic gradient descent (SGD) algorithm where at each iteration k we sample $D_t, D_{t-1}, \ldots, D_{t-H}$ from the probability distributions $P(D_{t-1},\ldots,D_{t-H})$ and $P(D_t|D_{t-1},\ldots,D_{t-H})$, and update $\pi_{k+1} = Proj\{\pi_k - \eta v_k\}$, where $v_k \in \partial\mathcal{L}(\pi_k(D_{t-1},\ldots,D_{t-H}),D_t)$ denotes a subgradient of the objective function[12], and $Proj$ denotes a projection onto the simplex for each (s,d) pair. The final output after K iterations is $\bar{\pi} = \frac{1}{K}\sum_{k=1}^{K}\pi_k$.

The next theorem, based on Theorem 14.12 in [49], bounds the loss of this algorithm. Let $\bar{\mathcal{L}}(\pi) = \mathbb{E}[\mathcal{L}(\pi(D_{t-1},\ldots,D_{t-H}),D_t)]$ denote the expected loss of a TE function, and let $\pi^* \in \arg\min_\pi \bar{\mathcal{L}}(\pi)$ denote the optimal TE function.

Theorem 2. *For every $\varepsilon > 0$, if SGD is run for $K \geq \frac{B^2\rho^2}{\varepsilon^2}$ iterations with $\eta = \sqrt{\frac{B^2}{\rho^2 K}}$, then the output of SGD satisfies*

$$\mathbb{E}\left[\bar{\mathcal{L}}(\bar{\pi})\right] \leq \bar{\mathcal{L}}(\pi^*) + \varepsilon,$$

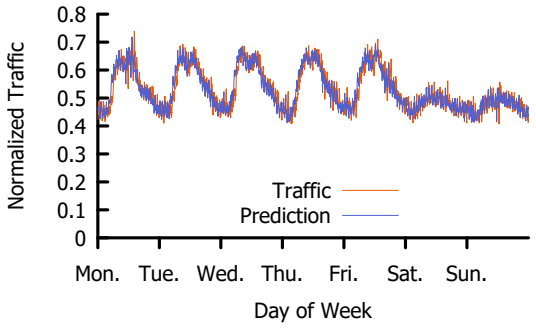
where the expectation is w.r.t. the sampling by the algorithm.

Theorem 2 shows that without function approximation (the TE function space spans *all* possible mappings from history to TE configuration), and with infinite data (the algorithm continuously samples from the true demand distribution), SGD converges to the optimal TE function with arbitrary precision. In practice, we relax both assumptions. In *DOTE* we sample from a large, but *finite*, dataset of historical demands, and use a *parametric* model (specifically, a neural network) to map from an *infinite* set of possible histories to valid TE configurations. Our empirical results show that, with enough data and a deep enough neural network, the approximate TE function *DOTE* learns is still very close to optimal.

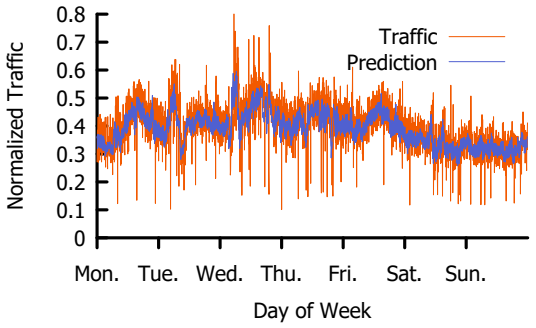
B.2 Maximum-Multicommodity-Flow and Maximum- Concurrent-Flow

We begin by stating a general convergence result for quasi-convex functions that satisfy certain assumptions. We then proceed to show that both maximum-multicommodity-flow and maximum-concurrent-flow indeed satisfy these assumptions, implying their convergence.

[12] The objective is not necessarily differentiable everywhere because of the max, but the subgradient exists for every π.



(a) Inter-data-center traffic



(b) Customer-facing traffic

Figure 10: Inter-data-center traffic and customer-facing traffic over the course of a week, along with the predictions of a linear regression model for the time-series.

B.2.1 General results

We begin by providing an analysis of stochastic quasi-convex optimization, under general assumptions. In the next section, we will show that maximum-multicommodity-flow and maximum- concurrent-flow are special cases of this setting.[13] Our analysis builds on two studies – the analysis of stochastic normalized subgradient of [20], which is for smooth and unconstrained problems, and the study of [33], which considered non-smooth quasi-convex optimization.

A quasi-convex function $f(x)$ satisfies that its level sets, $L(f;\alpha) = \{x|f(x) \leq \alpha\}$, are convex sets for all α.

We first define a normalized subgradient in the context of quasi-convex functions, following [33]. The normal cone to a convex set X at point x is defined by $N(X,x) = \{q \in \mathbb{R}^n | \langle q, y-x \rangle \leq 0 \quad \forall y \in X\}$. The set of subgradients at a point x are given by $N(L(f;f(x)),x)$. The set of normalized subgradients, $Q(f;x)$, at a point x, are given by $Q(f;x) = \mathrm{S}(0,1) \cap N(L(f;f(x)),x)$, where $\mathrm{S}(0,1)$ is the n-dimensional sphere of radius 1. These are directions of ascent – normalized vectors such that taking an infinitely small step in their direction is guaranteed to not decrease the function.

In the following, we consider a general stochastic optimization problem:

$$\min_{x \in X} \mathbb{E}_{D \sim P(D)} \left[f(x,D) \right], \tag{1}$$

where f is quasi-convex in x for every D.

We will further assume the following. Let $\mathbb{B}(z,r)$ denote the n-dimensional ball centered on z with radius r.

Assumption 1. *Set X is convex and bounded by $\mathbb{B}(0,\bar{B})$. The function f is bounded by B. It is also G-Lipschitz and quasi-convex in x for every D. Furthermore, $Q(\frac{1}{M}\sum_{i=1}^{M} f(x,D_i);x) \neq \emptyset$ for any $x \notin \arg\min_y \frac{1}{M}\sum_{i=1}^{M} f(y,D_i)$, and for every $D_1,\ldots,D_M$, we have that $\frac{1}{M}\sum_{i=1}^{M} f(x,D_i)$ is quasi-convex in x.*

[13] While we present results for quasi-convexity, the extension of these results to quasi-concave problems is immediate.

Note that the last requirement in Assumption 1 is not immediate, as the sum of quasi-convex functions is not necessarily quasi-convex.

The stochastic normalized subgradient method we consider works as follows [20]. At each iteration k we sample a minibatch $\{D_i\}_{i=1}^{b} \sim P(D)$ and define $f_k = \frac{1}{b}\sum_{i=1}^{b} f(x,D_i)$. We then update $x_{k+1} = Proj\{x_k - \eta v_k\}$, where $v_k \in Q(f_k;x_k)$ denotes a subgradient of the minibatch, and $Proj$ denotes a projection onto the set X. The final output after K iterations is $\bar{x}_K = \arg\min_{x_1,\ldots,x_K} f_k(x_k)$.

The analysis in [20] bounds the error of the normalized subgradient method, for smooth and unconstrained functions. We next adapt it to our setting.

The next definition adapts a central definition from [20] to our non-smooth setting.

Definition 1. *(SLQC) Let $x, x^* \in \mathbb{R}^n$, $\kappa, \varepsilon > 0$. We say that f is $(\varepsilon, \kappa, x^*)$-strictly-locally-quasi-convex (SLQC) in x if at least one of the following applies. (1) $f(x) - f(x^*) \leq \varepsilon$. (2) $Q(f;x) \neq \emptyset$ and for any $\Delta \in Q(f;x)$, and every $y \in \mathbb{B}(x^*, \frac{\varepsilon}{\kappa})$, it holds that $\langle \Delta, y-x \rangle \leq 0$.*

We next show that the Lipschitz and quasi-convex properties in Assumption 1 suffice to establish SLQC.

Lemma 1. *Let f satisfy Assumption 1. Fix D, and let $x^* \in \arg\min_{x \in X} f(x;D)$. Then f is (ε, G, x^*)-SLQC for all $x \in X$.*

Proof. Assume $f(x;D) - f(x^*;D) > \varepsilon$. Let Z denote the $f(x;D)$-level set of $f(x;D)$. Let ∂Z be the boundary of Z. By definition of the level set, for every $z \in \partial Z$, $f(z) - f(x^*) > \varepsilon$. From the Lipschitz property then, for every $z \in \partial Z$ we must have $\|z - x^*\| \geq \frac{\varepsilon}{G}$. Since Z is convex, we therefore have that $\mathbb{B}(x^*, \frac{\varepsilon}{G}) \subset Z$. From Assumption 1, $Q(f;x) \neq \emptyset$, and from the definition of $Q(f;x)$, we have that for every $y \in \mathbb{B}(x^*, \frac{\varepsilon}{G})$, if $\Delta \in Q(f;x)$ then $\langle \Delta, y-x \rangle \leq 0$. □

We next show that with high probability, the subgradient of each minibatch is a descent direction for the expected objective in (1).

Lemma 2. *Let Assumption 1 hold, and let* $x^* \in \arg\min_{x\in X} \mathbb{E}_{D\sim P(D)}[f(x,D)]$. *Assume that the minibatch size satisfies* $b = O\left(\frac{8nB^2\log(G\bar{B}/\delta)}{\varepsilon^2}\right)$. *Then, with probability at least* $1-\delta$, *we have that the minibatch average* $f_k = \frac{1}{b}\sum_{i=1}^{b} f(x,D_i)$ *is* $(\varepsilon, 2G, x^*)$*-SLQC in* x_k.

Proof. Let

$$\xi = \frac{1}{b}\sum_{i=1}^{b} f(x^*,D_i) - \mathbb{E}_{D\sim P(D)}[f(x^*,D)].$$

From Hoeffding's inequality, we have that

$$P(|\xi| \geq t) \leq 2\exp\left(-\frac{2bt^2}{B^2}\right).$$

Thus, if $b \geq \frac{B^2\log(2/\delta)}{2t^2}$ we have that with probability $1-\delta$, $|\xi| < t$.

Let $x_k^* \in \arg\min_{x\in X} \frac{1}{b}\sum_{i=1}^{b} f(x,D_i)$. Let

$$\xi' = \frac{1}{b}\sum_{i=1}^{b} f(x_k^*,D_i) - \mathbb{E}_{D\sim P(D)}[f(x_k^*,D)].$$

Then, using a covering number argument [50], we have that for $b \geq \frac{nB^2\log(G\bar{B}/\delta)}{2t^2}$, with probability $1-\delta$, $|\xi'| < t$. We have that

$$\frac{1}{b}\sum_{i=1}^{b} f(x_k^*,D_i) \leq \frac{1}{b}\sum_{i=1}^{b} f(x^*,D_i) \leq \mathbb{E}_{D\sim P(D)}[f(x^*,D)] + \xi,$$

and

$$\mathbb{E}_{D\sim P(D)}[f(x^*,D)] - \xi' \leq \mathbb{E}_{D\sim P(D)}[f(x_k^*,D)] - \xi' \leq \frac{1}{b}\sum_{i=1}^{b} f(x_k^*,D_i).$$

Therefore,

$$\frac{1}{b}\sum_{i=1}^{b} f(x^*,D_i) - \frac{1}{b}\sum_{i=1}^{b} f(x_k^*,D_i) \leq \xi + \xi'.$$

Now, similarly to the proof of Lemma 1, assume that $\frac{1}{b}\sum_{i=1}^{b} f(x_k,D_i) - \frac{1}{b}\sum_{i=1}^{b} f(x_k^*,D_i) > \varepsilon$. We choose $b = O\left(\frac{8nB^2\log(G\bar{B}/\delta)}{\varepsilon^2}\right)$ such that with probability $1-\delta$, $\xi+\xi' \leq \varepsilon/2$.

We therefore have:

$$\frac{1}{b}\sum_{i=1}^{b} f(x_k,D_i) - \frac{1}{b}\sum_{i=1}^{b} f(x^*,D_i) > \varepsilon - (\xi+\xi') \geq \frac{\varepsilon}{2}.$$

For simplicity, we denote $\bar{f}(x) = \frac{1}{b}\sum_{i=1}^{b} f(x,D_i)$. Note that $\bar{f}$ is quasi-convex, by Assumption 1. Let Z denote the $\bar{f}(x_k)$-level set of $\bar{f}(x)$. Let ∂Z be the boundary of Z. By definition of the level set, for every $z \in \partial Z$, $\bar{f}(z) - \bar{f}(x^*) > \varepsilon/2$. From the Lipschitz property then, for every $z \in \partial Z$ we must have $\|z-x^*\| \geq \frac{\varepsilon}{2G}$. Since Z is convex, we therefore have that $\mathbb{B}(x^*, \frac{\varepsilon}{2G}) \subset Z$. From Assumption 1, $Q(\bar{f};x) \neq \emptyset$, and from the definition of $Q(\bar{f};x)$, we have that for every $y \in \mathbb{B}(x^*, \frac{\varepsilon}{2G})$, if $\Delta \in Q(\bar{f};x)$ then $\langle \Delta, y-x\rangle \leq 0$. □

We are finally ready to present the converge result.

Theorem 3. *Let Assumption 1 hold. Suppose we run the stochastic normalized subgradient method for* $K \geq \frac{4G^2\|x_1-x^*\|^2}{\varepsilon^2}$ *iterations,* $\eta = \varepsilon/2G$, *and the minibatch size satisfies* $b = O\left(\frac{8nB^2\log(KG\bar{B}/\delta)}{\varepsilon^2}\right)$. *Then with probability* $1-2\delta$, *we have that* $f(\bar{x}_K) - f(x^*) \leq 3\varepsilon$.

Proof. This is a direct application of Theorem 5.1 of [20], where we used Lemma 2 to guarantee that at each iteration the minibatch is SLQC, as required in [20]. We note that by our Definition 1, the proof in [20] holds without change to the non-smooth setting. The projection onto the set X requires a straightforward modification to the proof of [20], where the first equality in their proof of Theorem 4.1 should be a $\leq$. The rest of the proofs remain unchanged. □

B.2.2 Results for Maximum-Multicommodity-Flow

We formally define the problem as follows.

for each tunnel T, let x_T denote the flow on that tunnel, and let $x_e = \sum_{T:e\in T} x_T$, for each edge e, denote the total flow on edge e. We define

$$\gamma = \max\left(\max_e \frac{x_e}{C_e}, 1\right),$$

and normalize the flows by γ, yielding normalized flows on a tunnel,

$$y_T = \frac{x_T}{\gamma},$$

and correspondingly, total normalized flows from source s to target t, $y_{s,t} = \sum_{T\in P_{st}} y_T$. Let $x = \{x_T\}$ denote our decision variables. Given a demand matrix D, the Max-MCF objective is

$$f_{MMCF}(x,D) = \sum_{s,t} \min(D_{s,t}, y_{s,t}).$$

We next show that f_{MMCF} is Lipschitz.

Lemma 3. *For any tunnel* T *and* $x \geq 0$, $\frac{x_T}{\gamma(x)} \leq C_{max}$.

Proof. Let $e \in T$, then by the definitions of $\gamma(x)$ and x_e, $\gamma(x) \geq \frac{x_e}{c_e} \geq \frac{x_T}{C_{max}}$. □

Lemma 4. $f_T(x) = \frac{x_T}{\gamma(x)}$ *is Lipschitz on* $\mathbb{R}^n_+$, *and its Lipschitz constant is at most* $K = 2\cdot\frac{C_{max}}{C_{min}}$.

Proof. Assume, without loss of generality, that $f(x) \geq f(y)$.
Case 1: $\gamma(y) = 1$

$$\begin{aligned}
&|f(x) - f(y)| \\
&= f(x) - f(y) \\
&= \frac{x_T}{\gamma(x)} - \frac{y_T}{\gamma(y)} \\
&= \frac{x_T}{\gamma(x)} - y_T \\
&\leq x_T - y_T \\
&\leq |x_T - y_T| \\
&\leq \|x - y\|_1 \\
&\leq 2 \cdot \frac{C_{max}}{C_{min}} \cdot \|x - y\|_1,
\end{aligned}$$

where the first inequality is since $\gamma(x) \geq 1$, and the third inequality is by the definition of $\|x\|_1$.
Case 2: $\gamma(y) = \frac{y_{e_0}}{C_{e_0}} > 1$, for some edge e_0.

$$\begin{aligned}
&|f(x) - f(y)| \\
&= f(x) - f(y) \\
&= \frac{x_T}{\gamma(x)} - \frac{y_T}{\gamma(y)} \\
&= \frac{x_T}{\gamma(x)} - \frac{y_T}{\gamma(x)} + \frac{y_T}{\gamma(x)} - \frac{y_T}{\gamma(y)} \\
&= \frac{1}{\gamma(x)} \cdot (x_T - y_T) + \frac{y_T}{\gamma(y)} \cdot \frac{1}{\gamma(x)} (\gamma(y) - \gamma(x)) \\
&\leq \frac{1}{\gamma(x)} \cdot (x_T - y_T) + \frac{y_T}{\gamma(y)} \cdot \frac{1}{\gamma(x)} \left(\frac{y_{e_0}}{C_{e_0}} - \frac{x_{e_0}}{C_{e_0}} \right) \\
&\leq \left| \frac{1}{\gamma(x)} \cdot (x_T - y_T) + \frac{y_T}{\gamma(y)} \cdot \frac{1}{\gamma(x)} \left(\frac{y_{e_0}}{C_{e_0}} - \frac{x_{e_0}}{C_{e_0}} \right) \right| \\
&\leq \frac{1}{\gamma(x)} \cdot |x_T - y_T| + \frac{y_T}{\gamma(y)} \cdot \frac{1}{\gamma(x)} \left| \frac{y_{e_0}}{C_{e_0}} - \frac{x_{e_0}}{C_{e_0}} \right| \\
&\leq |x_T - y_T| + \frac{C_{max}}{C_{min}} \cdot |y_{e_0} - x_{e_0}| \\
&\leq 2 \cdot \frac{C_{max}}{C_{min}} \cdot \|x - y\|_1,
\end{aligned}$$

where the first inequality is since $\gamma(y) = \frac{y_{e_0}}{C_{e_0}}$, $\gamma(x) \geq \frac{x_{e_0}}{C_{e_0}}$, $y_T \geq 0$, and $\gamma > 0$, the third inequality is since $|a+b| \leq |a| + |b|$, the fourth inequality is by Lemma 3 and since $\gamma(x) \geq 1$, and the last inequality is by the definitions of $\|x\|_1$, x_e and since $|a+b| \leq |a| + |b|$.

□

Proposition 2. *The function f_{MMCF} is Lipschitz, and its Lipschitz constant is at most $\sum_{s,t} \sum_{p \in P_{st}} 2 \cdot \frac{C_{max}}{C_{min}}$.*

Proof. By Lemma 4 and as a sum and minimum of Lipschitz functions. □

We next state two lemmas that we will use in our analysis.

Lemma 5. *For any $a, b \geq 0$, $c, d > 0$ and $\lambda \in [0,1]$, we have that $\min\left(\frac{a}{c}, \frac{b}{d}\right) \leq \frac{\lambda a + (1-\lambda)b}{\lambda c + (1-\lambda)d}$.*

Proof. Let $f(\lambda) = \frac{\lambda a + (1-\lambda)b}{\lambda c + (1-\lambda)d}$. Then,

$$\begin{aligned}
f'(\lambda) &= \frac{(a-b)(\lambda c + (1-\lambda)d) - (c-d)(\lambda a + (1-\lambda)b)}{(\lambda c + (1-\lambda)d)^2} \\
&= \frac{ad - bc}{(\lambda c + (1-\lambda)d)^2}.
\end{aligned}$$

Also, $f(0) = \frac{b}{d}, f(1) = \frac{a}{c}$, and $f'(\lambda)$ has a fixed sign for any $\lambda \in [0,1]$. Therefore, $f(\lambda) \geq \min\left(\frac{a}{c}, \frac{b}{d}\right)$. □

Lemma 6. *Let $x = \{x_T\}$, $x' = \{x'_T\}$, and $\lambda \in [0,1]$. Let $x'' = \{\lambda x_T + (1-\lambda)x'_T\}$, and let γ, γ' and γ'' be the respective normalization constants. Then $\gamma'' \leq \lambda\gamma + (1-\lambda)\gamma'$.*

Proof. We have that

$$\begin{aligned}
\gamma'' &= \max\left(\max_e \frac{x''_e}{C_e}, 1 \right) \\
&= \max\left(\max_e \frac{\lambda x_e + (1-\lambda)x'_e}{C_e}, 1 \right) \\
&\leq \max\left(\max_e \frac{\lambda x_e}{C_e}, \lambda \right) + \max\left(\max_e \frac{(1-\lambda)x'_e}{C_e}, 1-\lambda \right) \\
&= \lambda \max\left(\max_e \frac{x_e}{C_e}, 1 \right) + (1-\lambda) \max\left(\max_e \frac{x'_e}{C_e}, 1 \right) \\
&= \lambda\gamma + (1-\lambda)\gamma'.
\end{aligned}$$

□

We next show that Max-MCF satisfies Assumption 1.

Proposition 3. *The function f_{MMCF} is Lipschitz and bounded. Its maximum is obtained inside a convex set X. Furthermore, for every $D^1, \ldots, D^M$, we have that $\frac{1}{M}\sum_{i=1}^{M} f(x, D^i)$ is quasi-concave in x*

Proof. By definition, $x_T \geq 0$ for all T. Let $C_{max} = \max_e C_e$, and consider T-dimensional hypercube $X = [0, C_{max}]^T$. By definition, for every $x \geq 0$ that is outside X, there is an $x' \in X$ with an equivalent objective value. To see this, let γ the normalizing constant for x, and set $x' = x/\gamma$. Then,

$$x'_T = \frac{x_T}{\max\left(\max_e \frac{x_e}{C_e}, 1\right)} \leq \frac{x_T}{\max\left(\max_e \frac{x_e}{C_{max}}, 1\right)} \leq \frac{x_T}{\frac{x_T}{C_{max}}} = C_{max}.$$

But the normalizing factor for x' is 1, so x and x' have the same objective value.

Clearly, f_{MMCF} is bounded by $\sum_{s,t} \sum_{T: e \in T} C_{max}$.

The function is Lipschitz by proposition 2.

Let $\bar{f}_{MMCF}(x) = \frac{1}{M}\sum_{i=1}^{M} f_{MMCF}(x, D^i)$. We shall now show that for any $x, x' \in X$, and $\lambda \in [0,1]$, $\bar{f}_{MMCF}(\lambda x + (1-\lambda)x') \geq$

$\min\{\bar{f}_{MMCF}(x), \bar{f}_{MMCF}(x')\}$, proving that $\bar{f}_{MMCF}$ is quasi-concave. We denote by γ' and y' the respective normalization constant and normalized flows corresponding to x'. We also denote $x'' = \lambda x + (1-\lambda)x'$, and let γ'' and y'' denote its corresponding normalization constant and normalized flows, respectively.

$$
\begin{aligned}
&\min\Big(\sum_{i=1}^{M}\sum_{s,t}\min(D^i_{s,t}, y_{s,t}), \sum_{i=1}^{M}\sum_{s,t}\min(D^i_{s,t}, y'_{s,t})\Big)\\
&= \min\Big(\sum_{i=1}^{M}\sum_{s,t}\min(D^i_{s,t}, \sum_{T\in P_{st}}\frac{x_T}{\gamma}), \sum_{i=1}^{M}\sum_{s,t}\min(D^i_{s,t}, \sum_{T\in P_{st}}\frac{x'_T}{\gamma'})\Big)\\
&= \min\Big(\frac{1}{\gamma}\sum_{i=1}^{M}\sum_{s,t}\min(\gamma D^i_{s,t}, \sum_{T\in P_{st}} x_T), \frac{1}{\gamma'}\sum_{i=1}^{M}\sum_{s,t}\min(\gamma' D^i_{s,t}, \sum_{T\in P_{st}} x'_T)\Big)\\
&\le \frac{\lambda\sum_{i=1}^{M}\sum_{s,t}\min(\gamma D^i_{s,t}, \sum_{T\in P_{st}} x_T) + (1-\lambda)\sum_{i=1}^{M}\sum_{s,t}\min(\gamma' D^i_{s,t}, \sum_{T\in P_{st}} x'_T)}{\lambda\gamma + (1-\lambda)\gamma'}\\
&= \frac{\sum_{i=1}^{M}\sum_{s,t}\min(\lambda\gamma D^i_{s,t}, \lambda\sum_{T\in P_{st}} x_T) + \min((1-\lambda)\gamma' D^i_{s,t}, (1-\lambda)\sum_{T\in P_{st}} x'_T)}{\lambda\gamma + (1-\lambda)\gamma'}\\
&\le \frac{1}{\lambda\gamma + (1-\lambda)\gamma'}\sum_{i=1}^{M}\sum_{s,t}\min\Big(\lambda\gamma D^i_{s,t} + (1-\lambda)\gamma' D^i_{s,t},\\
&\quad \lambda\sum_{T\in P_{st}} x_T + (1-\lambda)\sum_{T\in P_{st}} x'_T\Big)\\
&= \sum_{i=1}^{M}\sum_{s,t}\min\Big(D^i_{s,t}, \frac{1}{\lambda\gamma + (1-\lambda)\gamma'}\sum_{T\in P_{st}}(\lambda x_T + (1-\lambda)x'_T)\Big)\\
&\le \sum_{i=1}^{M}\sum_{s,t}\min(D^i_{s,t}, \sum_{T\in P_{st}}\frac{x''_T}{\gamma''})\\
&= \sum_{i=1}^{M}\sum_{s,t}\min(D^i_{s,t}, y''_{s,t}),
\end{aligned}
$$

where the first inequality is by Lemma 5, the second inequality is since $\min(a,b)+\min(c,d) \le \min(a+c, b+d)$, and the third inequality is by Lemma 6. □

Lemma 7. *Let $x \notin \arg\max_y f(y)$, $x^* \in \arg\max_y f(y)$, and let γ, γ^* be the respective normalization constants. If $f(x+\lambda(x^*-x)) \ge \frac{\lambda\gamma^* f(x^*)+(1-\lambda)\gamma f(x)}{\lambda\gamma^*+(1-\lambda)\gamma}$ for any $\lambda \in [0,1]$, then $Q(f;x) \neq \emptyset$.*

Proof. The directional derivative of f along $x^* - x$ at x:

$$
\begin{aligned}
\nabla_{x^*-x} f(x) &= \lim_{h\to 0^+}\frac{f(x+h(x^*-x)) - f(x)}{h\|x^*-x\|}\\
&\ge \lim_{h\to 0^+}\frac{\frac{h\gamma^* f(x^*)+(1-h)\gamma f(x)}{h\gamma^*+(1-h)\gamma} - f(x)}{h\|x^*-x\|}\\
&= \lim_{h\to 0^+}\frac{\frac{h\gamma^*}{h\gamma^*+(1-h)\gamma}(f(x^*)-f(x))}{h\|x^*-x\|}\\
&\ge \lim_{h\to 0^+}\frac{\frac{\gamma^*}{\max(\gamma^*,\gamma)}(f(x^*)-f(x))}{\|x^*-x\|} > 0.
\end{aligned}
$$

Therefore, since $L(f; f(x))$ is convex, $-\frac{x^*-x}{\|x^*-x\|} \in Q(f;x)$. □

Lemma 8. *Let $x = \{x_T\}$, $x' = \{x'_T\}$, and $\lambda \in [0,1]$. Let $x'' = \{\lambda x_T + (1-\lambda)x'_T\}$, and let γ, γ' and γ'' be the respective normalization constants. Then, $\bar{f}_{MMCF}(x'') \ge \frac{\lambda\gamma\bar{f}_{MMCF}(x)+(1-\lambda)\gamma'\bar{f}_{MMCF}(x')}{\lambda\gamma+(1-\lambda)\gamma'}$.*

Proof.

$$
\begin{aligned}
\bar{f}_{MMCF}(x'') &= \sum_{i=1}^{M}\sum_{s,t}\min(D^i_{s,t}, \sum_{T\in P_{st}}\frac{x''_T}{\gamma''})\\
&\ge \sum_{i=1}^{M}\sum_{s,t}\min\Big(D^i_{s,t}, \frac{1}{\lambda\gamma+(1-\lambda)\gamma'}\sum_{T\in P_{st}}(\lambda x_T + (1-\lambda)x'_T)\Big)\\
&= \frac{\sum_{i=1}^{M}\sum_{s,t}\min\Big(\lambda\gamma D^i_{s,t} + (1-\lambda)\gamma' D^i_{s,t}, \lambda\gamma\sum_{T\in P_{st}}\frac{x_T}{\gamma} + (1-\lambda)\gamma'\sum_{T\in P_{st}}\frac{x'_T}{\gamma'}\Big)}{\lambda\gamma+(1-\lambda)\gamma'}\\
&\ge \frac{\sum_{i=1}^{M}\sum_{s,t}\Big(\min(\lambda\gamma D^i_{s,t}, \lambda\gamma\sum_{T\in P_{st}}\frac{x_T}{\gamma}) + \min((1-\lambda)\gamma' D^i_{s,t}, (1-\lambda)\gamma'\sum_{T\in P_{st}}\frac{x'_T}{\gamma'})\Big)}{\lambda\gamma+(1-\lambda)\gamma'}\\
&= \frac{\lambda\gamma\bar{f}_{MMCF}(x) + (1-\lambda)\gamma'\bar{f}_{MMCF}(x')}{\lambda\gamma+(1-\lambda)\gamma'},
\end{aligned}
$$

where the first inequality is by Lemma 6 and the second inequality is since $\min(a,b)+\min(c,d) \le \min(a+c, b+d)$. □

Proposition 4. *$Q(\bar{f}_{MMCF}, x) \neq \emptyset$ for any $x \notin \operatorname{argmax}_y \bar{f}_{MMCF}(y)$*

Proof. By Lemma 8 where $x = x^*, x' = x$, and by Lemma 7. □

Since Assumption 1 holds, Theorem 3 guarantees that the stochastic normalized subgradient method will converge to an optimal solution of the Max-MCF objective.

B.2.3 Results for Maximum-Concurrent-Flow

Given a demand matrix D, the Max-Concurrent-Flow objective is

$$f_{MCONC}(x, D) = \min\big(\{\frac{y_{s,t}}{D_{s,t}}\}_{s,t\in V, D_{s,t}>0} \cup \{1\}\big).$$

We assume that when $D_{s,t} \neq 0$, there is a minimal value ε for $D_{s,t}$, corresponding, e.g., to a single packet. We next show that Max-Concurrent-Flow satisfies Assumption 1.

Proposition 5. *The function f_{MCONC} is Lipschitz, and its Lipschitz constant is at most $\max_{s,t}\Big(\sum_{p\in P_{st}}\frac{2\cdot C_{max}}{\varepsilon\cdot C_{min}}\Big)$.*

Proof. By Lemma 4 and as a sum, minimum and multiplication by a constant of Lipschitz functions. □

Proposition 6. *The function f_{MCONC} is Lipschitz and bounded. Its maximum is obtained inside a convex set X. Furthermore, for every $D^1,\ldots,D^M$, we have that $\frac{1}{M}\sum_{i=1}^{M} f(x,D^i)$ is quasi-concave in x*

Proof. The claims in the beginning of proposition 3 hold for f_{MCONC}, and therefore its maximum is obtained inside a convex set.

Clearly, f_{MCONC} is bounded by 1.

The function is Lipschitz by proposition 5.

Let $\bar{f}_{MCONC}(x) = \frac{1}{M}\sum_{i=1}^{M} f_{MCONC}(x,D^i)$. We shall now show that for any $x,x' \in X$, and $\lambda \in [0,1]$, $\bar{f}_{MCONC}(\lambda x+(1-\lambda)x') \geq \min\{\bar{f}_{MCONC}(x),\bar{f}_{MCONC}(x')\}$, proving that $\bar{f}_{MCONC}$ is quasi-concave. We denote by γ' and y' the respective normalization constant and normalized flows corresponding to x'. We also denote $x'' = \lambda x+(1-\lambda)x'$, and let γ' and y'' denote its corresponding normalization constant and normalized flows, respectively.

$$
\begin{aligned}
&\min\Big(\sum_{i=1}^{M}\min\Big(\Big\{\frac{y_{s,t}}{D^i_{s,t}}\Big\}\cup\{1\}\Big),\sum_{i=1}^{M}\min\Big(\Big\{\frac{y'_{s,t}}{D^i_{s,t}}\Big\}\cup\{1\}\Big)\Big)\\
=&\min\Big(\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\frac{x_T}{\gamma}}{D^i_{s,t}}\Big\}\cup\{1\}\Big),\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\frac{x'_T}{\gamma'}}{D^i_{s,t}}\Big\}\cup\{1\}\Big)\Big)\\
=&\min\Big(\frac{1}{\gamma}\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}x_T}{D^i_{s,t}}\Big\}\cup\{\gamma\}\Big),\frac{1}{\gamma'}\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}x'_T}{D^i_{s,t}}\Big\}\cup\{\gamma'\}\Big)\Big)\\
\leq&\frac{\lambda\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}x_T}{D^i_{s,t}}\Big\}\cup\{\gamma\}\Big)+(1-\lambda)\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}x'_T}{D^i_{s,t}}\Big\}\cup\{\gamma'\}\Big)}{\lambda\gamma+(1-\lambda)\gamma'}\\
=&\frac{\sum_{i=1}^{M}\Big(\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\lambda x_T}{D^i_{s,t}}\Big\}\cup\{\lambda\gamma\}\Big)+\min\Big(\Big\{\frac{\sum_{T\in P_{st}}(1-\lambda)x'_T}{D^i_{s,t}}\Big\}\cup\{(1-\lambda)\gamma'\}\Big)\Big)}{\lambda\gamma+(1-\lambda)\gamma'}\\
\leq&\frac{\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\lambda x_T+\sum_{T\in P_{st}}(1-\lambda)x'_T}{D^i_{s,t}}\Big\}\cup\{\lambda\gamma+(1-\lambda)\gamma'\}\Big)}{\lambda\gamma+(1-\lambda)\gamma'}\\
=&\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\frac{\lambda x_T+(1-\lambda)x'_T}{\lambda\gamma+(1-\lambda)\gamma'}}{D^i_{s,t}}\Big\}\cup\{1\}\Big)\\
\leq&\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\frac{x''}{\gamma''}}{D^i_{s,t}}\Big\}\cup\{1\}\Big)\\
=&\sum_{i=1}^{M}\min\Big(\Big\{\frac{y''_{s,t}}{D^i_{s,t}}\Big\}\cup\{1\}\Big),
\end{aligned}
$$

where the first inequality is by Lemma 5, the second inequality is since $\min(a,b)+\min(c,d)\leq\min(a+c,b+d)$, and the third inequality is by Lemma 6. □

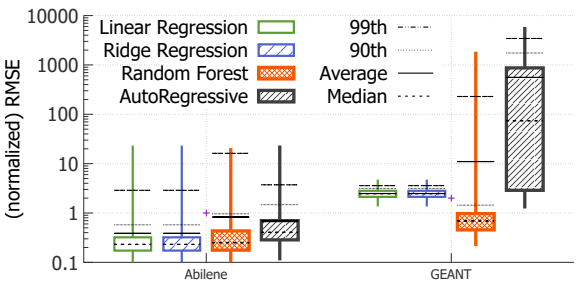


Figure 11: Accuracy of predicting demands; results from different prediction methods.

Lemma 9. *Let $x=\{x_T\}$, $x'=\{x'_T\}$, and $\lambda\in[0,1]$. Let $x''=\{\lambda x_T+(1-\lambda)x'_T\}$, and let γ, γ' and γ'' be the respective normalization constants. Then,* $\bar{f}_{MCONC}(x'') \geq \frac{\lambda\gamma\bar{f}_{MCONC}(x)+(1-\lambda)\gamma'\bar{f}_{MCONC}(x')}{\lambda\gamma+(1-\lambda)\gamma'}$.

Proof.

$$
\begin{aligned}
&\bar{f}_{MCONC}(x'') = \sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\frac{x''}{\gamma''}}{D^i_{s,t}}\Big\}\cup\{1\}\Big)\\
\geq&\sum_{i=1}^{M}\min\Big(\Big\{\frac{\sum_{T\in P_{st}}\frac{\lambda x_T+(1-\lambda)x'_T}{\lambda\gamma+(1-\lambda)\gamma'}}{D^i_{s,t}}\Big\}\cup\{1\}\Big)\\
=&\frac{\sum_{i=1}^{M}\min\Big(\Big\{\frac{\lambda\gamma\sum_{T\in P_{st}}\frac{x_T}{\gamma}+(1-\lambda)\gamma'\sum_{T\in P_{st}}\frac{x'_T}{\gamma'}}{D^i_{s,t}}\Big\}\cup\{\lambda\gamma+(1-\lambda)\gamma'\}\Big)}{\lambda\gamma+(1-\lambda)\gamma'}\\
\geq&\frac{\sum_{i=1}^{M}\Big(\min\Big(\Big\{\frac{\lambda\gamma\sum_{T\in P_{st}}\frac{x_T}{\gamma}}{D^i_{s,t}}\Big\}\cup\{\lambda\gamma\}\Big)+\min\Big(\Big\{\frac{(1-\lambda)\gamma'\sum_{T\in P_{st}}\frac{x'_T}{\gamma'}}{D^i_{s,t}}\Big\}\cup\{(1-\lambda)\gamma'\}\Big)\Big)}{\lambda\gamma+(1-\lambda)\gamma'}\\
=&\frac{\lambda\gamma\bar{f}_{MCONC}(x)+(1-\lambda)\gamma'\bar{f}_{MCONC}(x')}{\lambda\gamma+(1-\lambda)\gamma'},
\end{aligned}
$$

where the first inequality is by Lemma 6 and the second inequality is since $\min(a,b)+\min(c,d)\leq\min(a+c,b+d)$. □

Proposition 7. *$Q(\bar{f}_{MCONC},x)\neq\emptyset$ for any $x\notin argmax_y\bar{f}_{MCONC}(y)$*

Proof. By Lemma 9 where $x=x^*,x'=x$, and by Lemma 7. □

Since Assumption 1 holds, Theorem 3 guarantees that the stochastic normalized subgradient method will converge to an optimal solution of the Max-Concurrent-Flow objective.

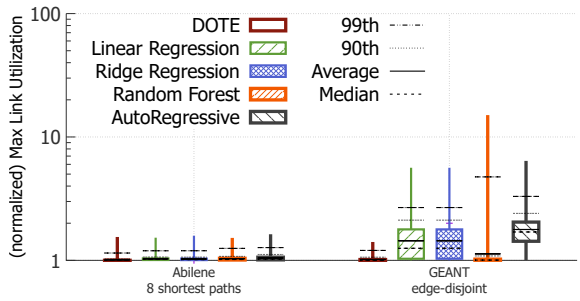


Figure 12: Impact of demand prediction accuracy on max-link-utilization.

C A Closer Look at Demand Prediction

Our results in §4 considered a demand-prediction-based scheme that utilizes linear regression. We next contrast linear regression with other prediction methods on our datasets. Specifically, we consider the following prediction methods: linear regression, ridge regressing, random forrest, and autoregressive model. With the exception of the autoregressive model, each of these schemes predicts the next traffic demand for each source-destination pair using only that specific pair's recently observed 12 most traffic demands, *i.e.*, the prediction for each pair is independent from the prediction for other pairs (as in SWAN [22]). The autoregressive model, in contrast, predicts the entire next DM from the 12 most recently observed DMs, to allow for detecting correlations between different pairs that might be conducive for prediction.

Figure 11 plots the accuracy of the different predictors, as quantified by the root-mean-squared-error, for the two publicly available WAN datasets. The accuracy is normalized by the average traffic demand for the dataset and presented in log-scale. Our results for *PWAN* and $PWAN_{DC}$ exhibit similar trends. As shown in the figure, linear regression and ridge regression achieve the best results on average on both WANs. We also considered a DNN-based predictor with a single hidden layer with 128 neurons and ReLU activation functions, but its performance was strictly dominated by linear regression on the test data (results omitted). Moreover, treating source-destination pairs individually attains better accuracy than that provided by the autoregressive model. We believe that this is because, on the one hand, the previous traffic demands for a single pair already contain a lot of valuable information and, on the other hand, the much larger input and output of the autoregressive model (entire DMs *vs.* single demands) makes effective learning more difficult.

Figure 12 plots the implications of choosing different predictors for TE performance, as quantified by the max-link-utilization, benchmarked against *DOTE*. Observe that *DOTE* outperforms all considered flavors of demand-based-prediction TE, and also that accuracy in demand prediction does not always translate to better TE performance, exemplifying the potential objective mismatch between the two, discussed in the Introduction.

D Robustness to Unexpected Traffic Changes

We consider the GEANT, Cogentco, and GtsCe network topologies with edge-disjoint tunnels. For Cogentco, and GtsCe we use the gravity model to generate demands for both train and test. To evaluate the implications of unexpected traffic changes, we add noise to the test set by multiplying each demand independently by a factor sampled uniformly at random from the range $[1-\alpha, 1+\alpha]$ for $\alpha \in \{0.1, 0.25, 0.35\}$.

Recall that for GEANT, *DOTE* generates TE configurations that are extremely close to the optimum (less than 2%). Our results show that even under random traffic perturbations, the distance from the omniscient oracle remains low; 2%, 2.9%, and 3.8% for $\alpha = 0.1, 0.25, 0.35$, respectively. For $\alpha = 0.35$, the distance from the omniscient oracle was 0.01% in the median, 13% in the 90th percentile, and no higher than 28% even in the 99th percentile.

For both Cogentco and GtsCe, *DOTE*'s trained model is roughly 0.5% from the omniscient oracle on the test demands are perturbed. This is because traffic is generated using the gravity model naturally does not reflect the intricate *temporal* patterns and complexity of real-world traffic. Even after perturbing the traffic in our experiments *DOTE* achieved near-optimal performance. Specifically, on Cogentco, the average distance from the omniscient oracle was 0.54%, 0.57%, and 0.6% for $\alpha = 0.1, 0.25, 0.35$, respectively. For $\alpha = 0.35$, the distance from the omniscient oracle was 0.56% in the median, 1% in the 90th percentile, and 1.4% in the 99th percentile. On GtsCe, the average distance from the omniscient oracle was 0.51%, 0.56%, and 0.61% for $\alpha = 0.1, 0.25, 0.35$ respectively. For $\alpha = 0.35$, the distance from the omniscient oracle was 0.57% in the median, 1% in the 90th percentile, and 1.4% in the 99th percentile.

	Tunnels	Week 1	Week 2	Week 3	Week 4
Abilene	8 SP	0.7	0.3	1.0	1.5
Abilene	edge-disjoint	2.1	2.4	2.4	2.0
GEANT	8 SP	1.4	2.7	2.9	3.1
GEANT	edge-disjoint	0.7	1.6	2.0	2.5

Table 3: Average weekly distance from the omniscient oracle achieved by *DOTE* for MLU across 4 consecutive weeks

	Tunnels	Week 1	Week 2	Week 3	Week 4
Abilene	8 SP	1.6	2.1	3.9	6.2
Abilene	edge-disjoint	1.1	1.4	3.1	5.5
GEANT	8 SP	4.9	4.7	5.0	4.8
GEANT	edge-disjoint	6.3	6.8	6.9	6.4

Table 4: Average weekly distance from the omniscient oracle achieved by *DOTE* for maximum-multicommodity-flow across 4 consecutive weeks

E Stochastic Optimization Loss Function Pseudocode

Function 1 Stochastic Optimization Loss Function Pseudocode

$G = (V, E, c)$ *// capacitated directed graph that models the WAN topology*
$U = \{(i, j) | i \in V, j \in V, i \neq j\}$ *// all pairs of nodes*
$T = \cup_{(s,t) \in U} P_{s,t}$ *// the set of all tunnels*
$A^{|U| \times |T|}$ *// specifies, for each pair of nodes $i \in U$ and tunnel $j \in T$ whether tunnel j interconnects the nodes in i*

$$A_{i,j} = \begin{cases} 1 & j \in P_i \\ 0 & \text{otherwise} \end{cases}$$

$B^{|T| \times |E|}$ *// specifies, for each tunnel i and edge j, whether tunnel i contains edge e*

$$B_{i,j} = \begin{cases} 1 & j \in i \\ 0 & \text{otherwise} \end{cases}$$

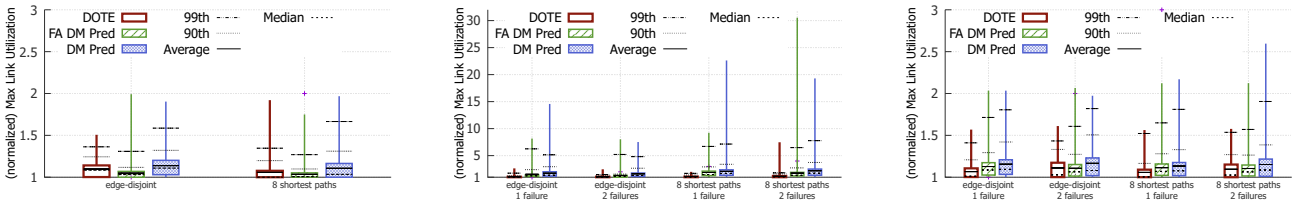
$C^{|E| \times 1}$ *// vector representing WAN link capacities*
$C_{i,1} = c(i)$

function LOSS(DNN_{output}, DM_{next})
  $DNN_{output}^{|T| \times 1}$ *// the output of the DNN*
  $DM_{next}^{|U| \times 1}$ *// the (actual) next demand matrix*

  // $\times$ and $/$ are element-wise operations
  // 1. Compute the splitting ratios
  $PathsSplit^{|T| \times 1} = DNN_{output} \times (A^T(1.0/A \times DNN_{output}))$
  // 2. Calculate the flow on each edge
  $FlowOnEdges^{|E| \times 1} = B^T((A^T \times DM_{next}) \times PathsSplit)$
  // 3. Compute the maximum-link-utilization
  $MaxLoad = max(FlowOnEdges/C)$
  return $MaxLoad$
end function

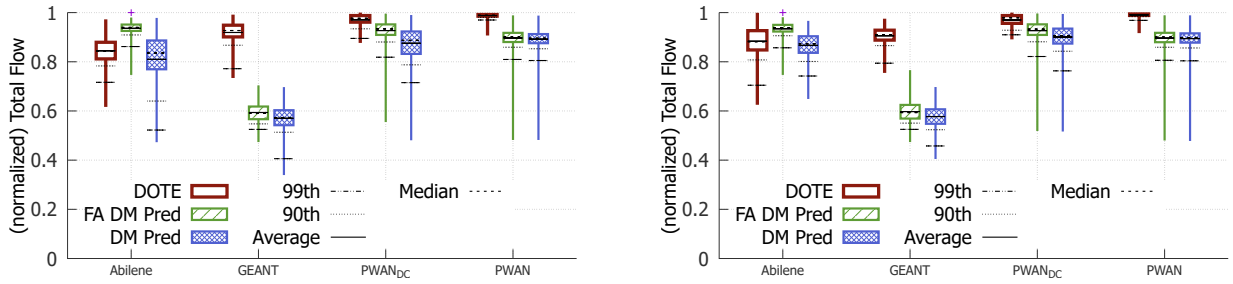
F Additional Failure Results

Analogous to Figure 9, Figure 13 shows the behavior under faults for the Abilene, GEANT and PWAN_{DC} topologies respectively. Figure 14 shows the results for maximum-multicommodity-flow.



(a) Abilene (b) GEANT (c) PWAN_{DC}

Figure 13: Understanding the behavior of *DOTE* under failures on different WAN datasets. The results are qualitatively similar to Figure 9.



(a) 8SP (b) edge-disjoint

Figure 14: Coping with a random link failure when aiming to maximize the total flow for two different tunnel choices.

Dashlet: Taming Swipe Uncertainty for Robust Short Video Streaming

Zhuqi Li, Yaxiong Xie, Ravi Netravali, Kyle Jamieson
Princeton University

Abstract

Short video streaming applications have recently gained substantial traction, but the non-linear video presentation they afford swiping users fundamentally changes the problem of maximizing user quality of experience in the face of the vagaries of network throughput and user swipe timing. This paper describes the design and implementation of Dashlet, a system tailored for high quality of experience in short video streaming applications. With the insights we glean from an in-the-wild TikTok performance study and a user study focused on swipe patterns, Dashlet proposes a novel out-of-order video chunk pre-buffering mechanism that leverages a simple, *non* machine learning-based model of users' swipe statistics to determine the pre-buffering order and bitrate. The net result is a system that outperforms TikTok by 28-101%, while also reducing by 30% the number of bytes wasted on downloaded video that is never watched.

1 Introduction

Short video streaming applications like TikTok and YouTube Shorts have rapidly risen in popularity, attracting billions of active users per month [31, 32, 41] and consistently topping popularity lists for mobile apps [33]. Unlike typical video streaming, the median duration of short videos is around 14 seconds [4]. During operation, these apps generate an ordered playlist of short videos (*e.g.*, based on a search or user-specific recommendations), and users watch them serially, with the ability to swipe from one to the next at any time. To provide an immersive experience and keep users engaged, short video streaming applications should minimize the video rebuffering time and maximize the video bitrate, which is modeled by quality-of-experience (QoE) [1, 3, 11, 13].

Although the aforementioned goals are consistent with those in traditional video streaming scenarios, existing ABR algorithms [2, 16, 22, 36, 40] are ill-suited for interactive, short videos. The reason is that predicting user swipes is difficult, and swipe times dictate both which video content will be viewed and when during a session. However, existing algorithms assume that the user will watch content sequentially to completion, and will hence buffer chunks (i.e., multi-second blocks of video) in that order. The deleterious effects, shown in Fig. 1, are twofold: **(1)** many chunks may be downloaded in the current video but never viewed if the user swipes before their playback, wasting resources and adding delays for the chunks that are required, and **(2)** users may swipe to the next video and incur significant rebuffering because that video's chunks have not been downloaded yet.

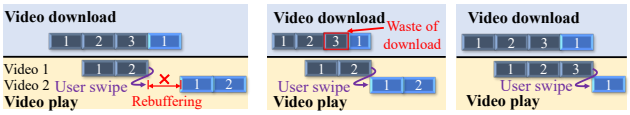


Figure 1: In short video apps, user swipes dictate the playing order of video chunks (and thus, the optimal chunk downloading order).

The fundamental challenge is that there are far too many possible chunk viewing sequences—the user may swipe at any position in each short video, and expects seamless (*i.e.*, no stalls) playback for both the current video, and the next one upon a swipe. The problem thus becomes how to find (at any time during playback) a buffering sequence of chunks in this large search space that maximizes QoE by simultaneously minimizing rebuffering time and wasted bandwidth.

To understand how commercial short streaming platforms attempt to address these challenges, we have conducted a detailed examination of TikTok in the wild (§2). Our key finding is that TikTok does download chunks out of order, but follows a generic algorithm that hedges against immediate rebuffering in the face of fast user swipes (it always pre-buffers the first chunk for the next five videos regardless of network conditions, user patterns, and/or video). This, however, entails substantial QoE penalties and wasted data consumption, as we will show via results from our own study of user swipe patterns across two distinct sets of users on a college campus and Amazon Mechanical Turk (§3). Specifically, we find substantial heterogeneity in the swipe patterns across users, with each warranting a different chunk downloading strategy.

A naïve solution would be to simply predict user swipes—if accurate, this would reduce the problem to a traditional streaming setting since chunk viewing sequences would be known a priori. However, predicting user behavior in interactive applications has consistently proven to be difficult [6, 21, 26]. Instead, we take a more fundamental approach that is rooted in an understanding of where swipe predictions are actually helpful (and actionable).

We present **Dashlet**, a new video streaming algorithm for short video applications (§4). The underlying insight behind Dashlet is that application playback constraints predetermine the relative priorities between many chunks that are candidates for buffering. More specifically, **(1)** later chunks in a video are only reachable via earlier ones, and **(2)** later videos are only reachable via swipes from earlier ones. To prioritize among the remaining chunks, *e.g.*, the next chunk in a given video *vs.* the first chunk in the next video, only coarse grained

information about swipe timings in videos is required. We show, via our user study, that although users tend to exhibit multimodal swipe patterns (complicating chunk prioritization) across videos, distributions from aggregating users' swipes *per video* provide a clear enough signal about which mode to expect. This information is readily available to current short video platforms, and our finding is spiritually aligned with past studies that highlight similarities in user engagement for certain video content [35, 43].

Building on this, Dashlet develops functions that characterize the expected rebuffering time for each potential chunk that could be downloaded, as a continuous function over both the expected download and playback times. These functions embed the aforementioned inter-chunk relationships, as well as rough swipe likelihoods at video start and end. Using these functions, Dashlet employs a greedy algorithm to determine the set of ordered chunks that should be downloaded in the current time horizon to minimize expected rebuffering delays for a given network estimate and across potential viewing sequences. This buffer sequence then feeds directly into a traditional ABR algorithm, which determine bitrates for those chunks that maximize overall QoE. Dashlet further improves upon existing short video systems by not prematurely binding bit rate decisions across entire short videos, and not letting the network idle at any point in time.

We have implemented Dashlet in the DASH framework [8], and compare with the TikTok mobile app with both a human subjects study and a trace-drive study[1]. Across these conditions, we find that Dashlet outperforms TikTok by 28-101% in QoE values, including 8-39% improvement on video bitrate, 1.6-8.9× reduction on rebuffering penalty, and 30% reduction on data wastage. Dashlet's QoE improvement varies with the network throughput, *i.e.*, 543.7%, 221.4%, and 36.6% over TikTok when the throughput is 2-4, 4-6, and 10-12 Mbps, respectively. The improvement diminishes with throughput approaching to 20 Mbps because both Dashlet and TikTok are getting closer to optimum. Further, Dashlet is tolerant to errors in swipe distributions: QoE degradations are only 10% with distribution errors of 50%. We will open source our datasets and implementation post publication.

2 A TikTok Case Study

We examine how TikTok, a state-of-the-art short video app, operates. We first describe its basic architecture (§2.1), before analyzing its operation and limitations (§2.2).

2.1 Short Video Streaming Primer

Unlike traditional streaming apps that divide video into chunks of equal time duration, TikTok splits each video into size-based chunks. For each supported bitrate, if the video file is smaller than 1 MB, TikTok treats the entire video as one chunk; else, the first chunk is the first MB, and the remaining video becomes the second. This chunking strategy enhances reliability, as TikTok pre-buffers first chunks (to cope with swipe uncertainty) so chunking in terms of bytes eliminates first-chunk size variance from variable bitrate encoding.

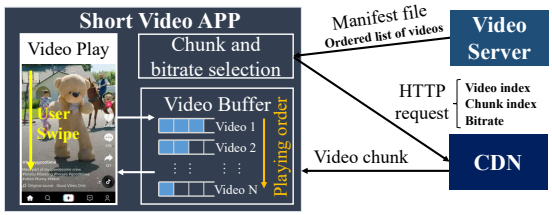


Figure 2: System architecture of TikTok and other short video apps.

Upon receiving a client session request either via a keyword search or category selection (*e.g.*, recommended videos), the server generates an ordered list of short videos to serve (Fig. 2). The server then ships a *manifest file* to the client which embeds the URL, as well as information about the number of *chunks* (multi-second blocks of video) and available bitrates, per video in the ordered list. The client operates much like a traditional streaming player (*e.g.*, DASH), maintaining a playback buffer for downloaded video and employing an adaptive bitrate (ABR) algorithm to determine what chunk to download next, when, and at what bitrate.

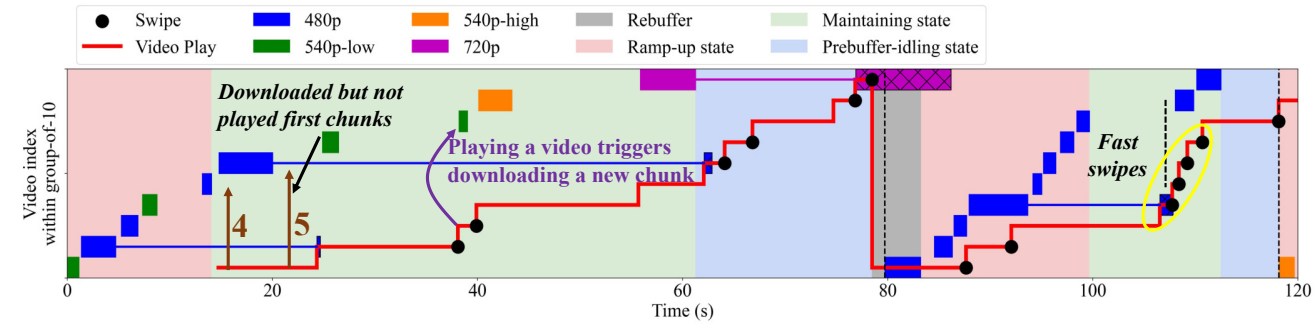
A key difference between traditional and short video streaming is that the client maintains one logical buffer per video in the server-provided manifest file, which contains information for an ordered group of 10 videos. The client requests a new manifest file after it downloads all the first chunks of the videos in the current manifest. Video playback operates sequentially within each logical buffer and across buffers (in the specified order); user swipes and video completion trigger the playback to move to the head of the buffer for the next video. To cope with such semantics, ABR algorithms for short videos have the ability to download chunks for any of the videos in the manifest file at any time.

TikTok provides four bitrate options for each video: 480p, 560p low, 560p high, and 720p, with bitrate adaptation occurring only at video-level (and not chunk-level) granularity. We hypothesize this is because the first 1 MB of a video encoded at different bitrates corresponds to different time durations, precluding seamless bitrate switches for the latter chunk, *i.e.*, content would be missed or repeated. As we will discuss, such constraints significantly limit TikTok from adapting to variations in network capacity during user sessions.

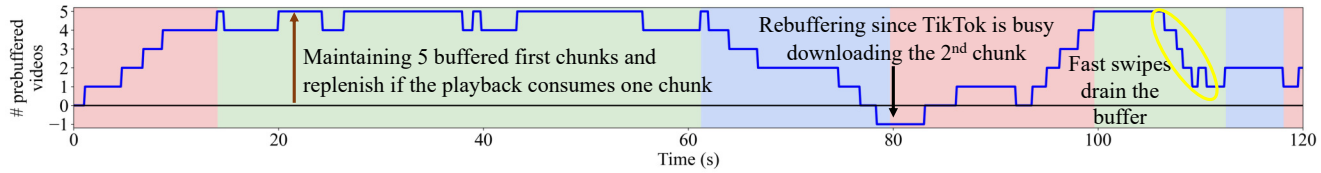
2.2 Analysis of TikTok

To study TikTok in a controlled and systematic manner, we perform our analysis over emulated networks using

[1] We release the code with the following url: https://github.com/PrincetonUniversity/Dashlet under the MIT Open Source License.



(a) Video chunk downloading and playing timeline: video index within a group-of-10 versus wall clock time. The *left and right edge* of the rectangular boxes respectively represent the downloading start and completion times of a chunk, while thin horizontal lines connect first and second chunks (if a second chunk exists), and box *color* indicates bitrate. The solid red line plots video playback.



(b) Client-side buffer occupancy as a function of time, the gap between playback and highest chunk downloaded in **(a)**.

Figure 3: An illustrative video downloading and video playing trace of TikTok, with associated video bitrate and buffer occupancy statistics.

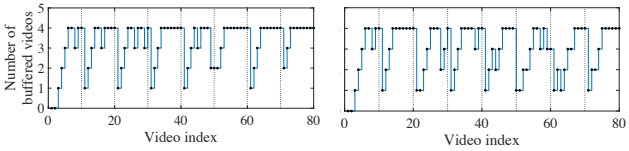
Mahimahi [23]. We log in to TikTok with a two-year old account and mirror its screen to a Linux desktop with scrcpy [28] and use the pyautogui tool [24] to replay aggregated user swipe traces that were collected from our user study (described in §3). During experiments, we use the mitmproxy [5] to collect and decrypt TikTok's network traffic. From the deciphered HTTP messages and headers, we are able to extract for each requested chunk, the video that it pertains to, its index in that video, the requested bitrate, and the download start/end time. Finally, we develop a screen analysis tool using pyautogui and opencv [17] to record duration of each rebuffering event (§5.1 further details our setup).

2.2.1 Chunk Download Control

TikTok's download control algorithm depends both on instantaneous network throughput and the client's internal buffer status: Fig. 3a illustrates its decisions (*i.e.*, order and timing of chunk downloads across videos, bitrates used each time) during a representative two-minute session. We plot client-side playback buffer occupancy in Fig. 3b, which shows the number of videos with at least one downloaded (but unplayed) chunk. We see that TikTok spends most of its time downloading the first chunk of videos, and downloads the second chunk when and only when the video starts to play, *e.g.*, the download of the second chunk of video two and the play-start of video two start simultaneously at $t = 22$ s.

Our analysis indicates that TikTok proceeds according to three discrete states, cycling among the three in order to handle one *group-of-ten* videos. At startup and the start of every group-of-ten, the **ramping-up state** continuously downloads first chunks to build up buffers. After accumulating five first chunks at $t = 18$ seconds, TikTok starts to play the buffered video and enters the **maintaining state**, where it aims to maintain a constant five buffered first chunks. Upon playing a new video (due to user swipe or reaching the end of a video), the client fetches one first chunk from the buffer, triggering TikTok to immediately initiate download of the first chunk of the next video in the manifest, as indicated by the additional download events corresponding to either swipes or video changes due to end of video in the green "maintaining state" regions of Fig. 3a. We see in Fig. 3b that as the downloading of each first chunk finishes, buffer levels return to five, the high water mark buffering level TikTok has chosen. The advantage of the maintaining state is resilience to quick user swipes: in the second group-of-ten of Fig. 3 ($t = 110$), the user swipes early in multiple consecutive videos, quickly draining the buffer, but TikTok experiences no rebuffering since its buffer contains the five first chunks.

Finally, after downloading all the first chunks of the 10 videos listed in the current manifest file, TikTok enters the **prebuffer-idling state**, where it stops initiating any new downloads of first chunks. Meanwhile, TikTok continues video playback, consuming video chunks in its buffer, so buffer occupancy decreases monotonically in this state, as shown in Fig. 3b. Our hypothetical explanation of this idle period is that TikTok is waiting to measure the user's reaction (swiping early means they might not be interested in the content) to the videos TikTok recommends in last round



(a) Net. throughput 10 Mbit/s. **(b)** Net. throughput 3 Mbit/s.

Figure 4: The number of downloaded videos inside the buffer when TikTok starts to download the first chunk of a new video via networks with capacity of **(a)** 10 Mbit/s and **(b)** 3 Mbit/s.

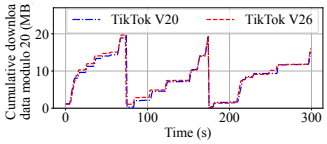


Figure 5: Cumulative downloaded data modulo by 20 MB for TikTok v20.9.1 and TikTok v26.3.3 when playing the same video sequence at the same swipe pace.

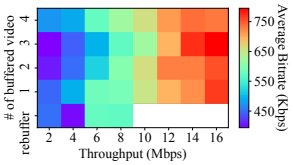


Figure 6: Impact of network throughput and client video buffer occupancy on TikTok's chosen video bitrate.

(manifest file), so it can assess its recommendation quality and adjust the subsequent round's recommendation before sending the next manifest file.

In contrast to the resilience of the maintaining state, TikTok becomes somewhat vulnerable in the prebuffer-idling state, where TikTok drains the buffer by itself. For example, TikTok experiences rebuffering in the middle of two video groups in Fig. 3. At that moment, TikTok has no buffered first chunk and at the same time spends a long time downloading the second chunk of the current video, leaving no time budget for downloading the first chunk of next video. In such a case, one user swipe results in rebuffering.

When the user starts to watch the ninth of the group-of-ten videos listed in a manifest, TikTok exits prebuffer-idle and begins afresh in the ramp-up state to download the videos listed in the next manifest file. The cycle through these three states repeats for each group-of-ten.

2.2.2 Network and Swipe Input Adaptation

We now investigate the effects of swipes, buffer occupancy, and the network on TikTok's bitrate and buffering choices. To measure the impact of the network on buffering strategy, we control network capacity to 10 and 3 Mbit/s using Mahimahi and plot the number of buffered first chunks at the moment TikTok initiates a download of the first chunks, in Fig. 4. Combing Fig. 4a and 4b, we see that TikTok adopts the same buffering strategy regardless of network capacity.

Next, we analyze the joint impact of network throughput and buffering status on Tiktok's bitrate decisions. We collect instantaneous network throughput and buffer status coupled with TikTok's bitrate decisions, for 5,300 videos, and plot the results in Fig. 6. In the figure, the x-axis is the network throughput of the one-second period before the downloading of that video, *i.e.*, the time period within which TikTok makes its decisions about the bitrate. The y-axis is the number of downloaded first chunks in the buffer. The color of a tile represents the average bitrate R of the video, which is given by $R = S/L$ where S is the size of the video in bits and L is the length of the video in seconds. Some tiles are not colored because the combination of the throughput and buffer status is not seen during our measurement, *e.g.*, when the throughput is 16 Mbit/s, we always observe four downloaded first chunks in the buffer. We observe that bitrate decisions correlate positively with network throughput, but observe no evidence for correlation with buffer status.

2.2.3 Buffering logic on different versions of TikTok

Our reverse engineering tool can only decipher complete TikTok telemetry information up to version v20.9.1. To investigate whether there are any updates in the buffering algorithm between v20.9.1 and the newest TikTok version (v26.3.3), we use scripts to watch the same videos on different versions, under the same network throughput and swipe pace. We record the number of bytes downloaded versus time with tcpdump. Fig. 5 shows an example trace for the two versions of TikTok. By correlating the downloading traces at different throughput and swipe speed, we infer that v20.9.1 and v26.3.3 use similar or identical buffering logic. In the rest of the paper, we only present v20.9.1 results.

2.2.4 Limitations of Current Short Video Streaming

Despite pre-buffering the beginnings of short videos, TikTok has a fundamentally static approach to coping with swipe uncertainty, with no evidence for adaptation across different videos or users. This approach is often too cautious or aggressive, manifesting in two particular ways:

Lack of swipe prediction. TikTok prioritizes the downloading of the first chunk, assuming that the user always swipes frequently, and delays the downloading of the second to the beginning of video playback. As we will show next however, there are indeed some users who swipe early when watching a video, but there also a significant number of users who watch most of many videos and swipe at the end or not at all. So, the urgency of downloading the second chunk varies with users and videos: a fixed rule cannot handle all cases.

Premature bitrate binding. TikTok groups the first MB of video data into the first chunk but selects the bitrate for both chunks according to the network conditions present during the first, prematurely binding the system into that bitrate for both. By design, there is often a large time lag between the downloading of the first and second chunks, as discussed above (the median gap between first and second chunk downloads is 25 s., with an interquartile range of 23 s.), resulting in a potential mismatch with network conditions

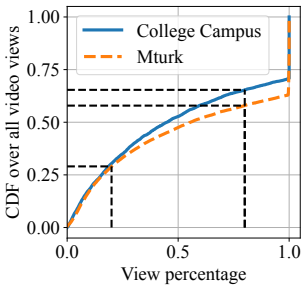


Figure 7: The distribution of average viewing percentage across all short video views.

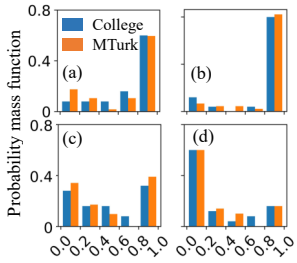


Figure 8: Distribution (across different users) of video viewing percentage for four sample videos *(a)–(d)*.

that change in the meantime.

Network Idling. As shown in Fig. 3a, TikTok has a prebuffer idling state. In the contrast, the buffer is not full and the bitrate of videos still has room to improve. This also calls for a better ABR algorithm to stream higher bitrate video by utilizing the idle time in a better way.

To understand the mismatch between TikTok's generic rule and the varying user swipe patterns, in the next section, we characterize the swipe patterns across real users and videos via two user studies.

3 Characterizing User Swipes

To better understand the nature of user interactions (*i.e.*, swipes) with short video applications, we conducted two IRB-approved user studies. In each study, we present users with a web-based short video streaming service that resembles the interface offered by TikTok We considered 500 popular short videos gathered by crawling the videos displayed on the TikTok landing page over time. The videos were randomly ordered per session, and each user watches 20 minutes of video with the ability to swipe freely (all swipes are recorded). Note that the number of videos watched by a given user depends on the number of swipes they performed.

For generality, we performed two versions of this study:

1. College campus: we recruit 25 student volunteers who collectively swipe 3,069 times during the study.

2. Amazon Mechanical Turk ("MTurk"): we recruit 258 different users. To ensure active user participation, we augment our web application to inject random interactivity tests that ask users to swipe within 10 seconds. Users who fail to swipe in time are entirely excluded from the study; users who do swipe continue the experience, but we exclude the forced swipe(s) from our final dataset. In total, we retain data from 133 workers, which covers 15,344 swipes.

Overall swipe distributions. Fig. 7 shows the distribution of swipe times across all video-user pairs in both studies. Users are most likely to swipe either soon after video playback begins or at the end of the video (manually or via auto-swiping once the video completes); this is consistent with prior studies on user swipe patterns [44]. For instance, 29% and 42% of swipes from MTurk users are within the first 20% or last 20% of videos, respectively. Swipes between these two endpoints occur, but far less often and with increasingly low likelihood as users watch more videos, *e.g.*, only 6% of swipes in the College Campus dataset are in the 60–80% of videos.

Swipe distributions per video. Fig. 8 shows swipe probabilities for four representative videos, aggregated across users who watched each one in the two studies. Different videos yield significantly different swipe distributions: over 60% of swipes in videos (a) and 80% of swipes in videos (d) come within the last few seconds (indicating low swipe probabilities for these). Video (c) exhibits the opposite pattern—60% of swipes in the first 20% of the video (indicating high swipe probabilities)—while swipes in (b) are more evenly distributed in time. Perhaps more importantly, we observe substantial stability in the swipe distributions per video across different user datasets: KL divergence values between the MTurk and College Campus datasets are 0.2 and 0.8 for the median and 95th percentile videos, respectively.

Conclusions. Despite general similarities in swipe patterns, users follow a few different modes of swiping (*e.g.*, swiping early in the chunk *vs.* not at all), each of which warrants a different buffering strategy to ensure high QoE. Fortunately, cross-user swipe data that is aggregated *per video* provides a relatively stable indicator as to how likely swipes are (and will be) in a given video, and (more coarsely) at what part of the video they will occur. We show in §4 how Dashlet leverages this coarse information – which is readily available at existing short video servers – to make robust buffering decisions that handle cross-user swipe traces.

4 Design

Dashlet leverages swipe distribution stability across videos (§3) to get a *coarse* sense of the likelihood of swipes at different video chunks. Coupling this information with constraints on inter-chunk viewing sequences intrinsic to short video applications, Dashlet models the *expected rebuffering time* for each potential chunk as a continuous function over the expected download and playback times (§4.1), then employs a greedy algorithm atop those functions to find a chunk buffering sequence that minimizes expected rebuffering delay over a time horizon for a given network throughput estimate, and across different user viewing sequences. Lastly, Dashlet feeds that buffering sequence into a bitrate selection algorithm (RobustMPC [40] in our implementation) to control video chunk bitrate and optimize overall QoE (§4.2).

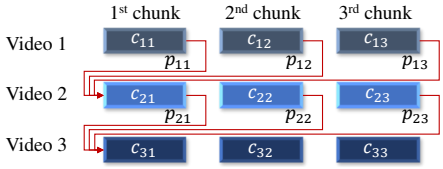


Figure 9: Short video streaming model: the player plays videos sequentially, switching to the first chunk of the next video after a swipe.

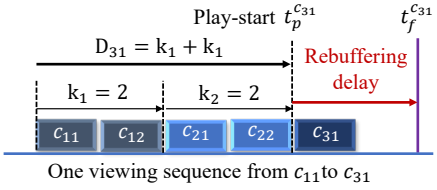


Figure 10: Chunk rebuffering delay depends on the order between the play start time t_p and the download finish time t_f.

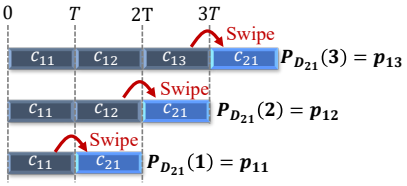


Figure 11: Three possible viewing sequences that start from chunk c_{11} and end at chunk c_{21}.

4.1 Forecasting Rebuffering Delay

Dashlet's expected rebuffering functions aim to quantify user-perceived delays across different chunk download times and viewing sequences. We begin by explaining the construction of these functions in a discrete setting where users can only swipe at chunk boundaries; we then extend the discussion to incorporate arbitrarily-timed user swipes.

System Model. Short video apps follow the flow shown in Fig. 9. Each video consists of multiple chunks of *chunk time* T. Within the ith video with N_i chunks, if the user does not swipe, the video player plays its chunks c_{ij} sequentially, where $j \in [0, N_i]$ is the chunk index. When playback reaches the end of the video or the user swipes, the player jumps to the first chunk of the next video. Since user swipe distributions vary across videos (§3), we denote the probability that the user swipes after watching chunk c_{ij} as p_{ij}. The list of the chunks the user watches is a *viewing sequence*

$$V_s = [c_{11}, \ldots, c_{1k_i}, c_{21}, \ldots, c_{K1}, \ldots, c_{Kk_L}] \tag{1}$$

where the user views the first k_i chunks of the ith video, assuming that the user watches L videos in total. Then the probability distribution of k_i is $P_{k_i} = \{p_{i1}, p_{i2}, \ldots, p_{iN_i}\}$. We define D_{ij}, the number of chunks that a user has watched prior to chunk c_{ij}:

$$D_{ij} = \sum_{l}^{i-1} k_l + (j-1). \tag{2}$$

By knowing the number of chunks that a user has watched before c_{ij}, the playback start time of c_{ij} then will be $D_{ij} \cdot T$. As shown in Fig. 10, the expected rebuffering delay for some chunk c depends on the relationship between the chunk's *play start* time t_p^c and *download finish* time t_f^c. There exists no rebuffering if the chunk downloading finishes before the play start time. Otherwise, rebuffering happens and the time difference between t_p^c and t_f^c tells us c's *rebuffering delay*:

$$T_c^{\text{rebuf}}(t_f^c, t_p^c) = \begin{cases} 0, & t_f^c < t_p^c \\ t_f^c - t_p^c, & t_f^c \geq t_p^c \end{cases} \tag{3}$$

The play start time of each chunk is determined by the viewing sequence V_s, as shown in Fig. 10. Since our goal is to schedule c's download to minimize rebuffering, we now formulate a reward function to meet this goal, parameterized on t_f^c and averaging over all possible viewing sequences (which are not under our control). The *expected rebuffering delay* of chunk c given that chunk's download finish time t_f^c, across all possible viewing sequences, is:

$$\mathbf{E}_c^{\text{rebuf}}(t_f^c) = \sum_{V_s \in \Phi} \Pr(V_s) \cdot T_c^{\text{rebuf}}(t_f^c, t_p^c(V_s)) \tag{4}$$

where probability $\Pr(V_s)$ represents how likely a specific viewing sequence V_s will appear based on user swipe distribution data, $t_p^c(V_s)$ is c's play start time in V_s, and Φ is the set of all possible viewing sequences.

To calculate the expected rebuffering delay for a specific chunk, we enumerate all possible viewing sequences that reach this chunk, as Eq. 4 shows. For each sequence, we compute how likely this sequence will appear based on user swipe distributions, and then determine the play start time of that specific chunk. Based on short video chunk playback constraints (§1, p.), we propose separate algorithms for calculating the expected rebuffering delay of a video's first chunk, and remaining chunks, respectively.

First chunk of a video. The number of possible viewing sequences between chunk one of video one (c_{11}) and chunk one of video i (c_{i1}) increases exponentially with i. On the other hand, the number of sequences from the first chunk of the previous to the first chunk of the current video is bounded by the number of chunks in the former. For example (see Fig. 9), there are three possible viewing sequences from chunk c_{21} to c_{31}. We therefore enumerate the viewing sequences in a recursive manner: deriving the viewing sequences that reach the first chunk of the ith video based on the viewing sequence of the first chunk of the $(i-1)^{\text{st}}$ video.

We start from the base case, viewing sequences from c_{11} to c_{21}. Fig. 11 lists all three possible viewing sequences that start from c_{11}: we see that random variable $D_{21} = k_1$ (*cf.* Eq. 2). Similarly, as shown in Fig. 10, $D_{31} = D_{21} + k_2$ (*cf.* Eq. 1). The distribution of D_{31} is then:

$$P_{D_{31}}[n_0] = \sum_{i=1}^{n_0-1} P_{D_{21}}[i] \cdot P_{k_2}[n_0 - i] \tag{5}$$

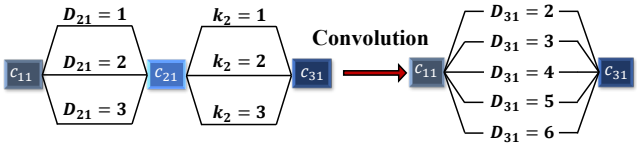


Figure 12: Convolution of the $P_{D_{21}}[\cdot]$ and $P_{K_2}[\cdot]$ provides us the probability distribution $P_{D_{31}}[\cdot]$.

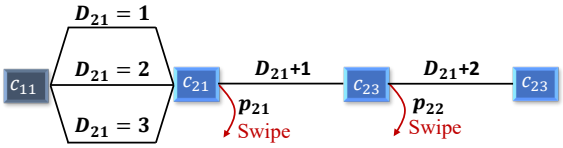


Figure 13: Starting from chunk c_{21}, the user must watch the second video continuously with swiping to reach chunk c_{23}.

where $P_{D_{31}}[n_0]$ means the probability of there are n_0 chunks before chunk c_{31} is viewed. This formula by definition is the operation of convolution between D_{21} and k_2, as shown in Fig. 12. Without losing generality, the number of chunks that user watches before chunk c_{i1}, D_{i1}, is $D_{(i-1)1}+k_{i-1}$. Therefore the distribution of D_{i1} is:

$$P_{D_{i1}} = P_{D_{(i-1)1}} * P_{k_{i-1}}. \tag{6}$$

With the knowledge of D_{i1}'s distribution for the first chunk of all the videos, we calculate the expected rebuffering delay of chunk c_{i1}, as the function of download finish time:

$$\mathbf{E}^{\text{rebuf}}_{c_{i1}}(t_f) = \sum P_{D_{ij}}[n] \cdot T^{\text{rebuf}}_{c_{i1}}(t_f, (n+1)T) \tag{7}$$

Remaining chunks in a video. There exists only one viewing sequence from the first to later chunks of the same video: Fig. 13 shows that c_{23} will be played when and only when the user watches the $i=2^{\text{nd}}$ video continuously without swiping. For non-first chunk c_{ij}, the number of chunks that user watched before it, D_{ij}, is the summation of D_{i1} and $j-1$ since the user has to watch the first $j-1$ chunks in video i before starting to watch it. Then the distribution of D_{ij} is that of D_{i1}, delayed by $j-1$ chunks. In addition, the user might swipe to the next video before watching c_{ij}:

$$P_{D_{ij}}[n_0] = P_{D_{i1}}[n_0-(j-1)] \times (1-\sum_{m=1}^{j-1} p_{im}). \tag{8}$$

With the distribution of D_{ij}, we follow the same procedure to calculate expected rebuffering time for remaining chunks in a video, according to Eq. 7.

Arbitrary user swipes. In reality, swipes do not only happen after a chunk finishes. If the continuously-valued viewing time for video i is κ_i, the PDF of κ_i is $f_{\kappa_i}(t_0)$. The *play start time* of c_{ij}, Δ_{ij}, is a random variable, with PDF $f_{\Delta_{ij}}(t)$. For the first chunk of video i, its playing start time $t_f^{c_{i1}}$ is also the summation of the playing start time of the previous video $t_f^{c_{(i-1)1}}$ and the time the user spends watching the previous video κ_{i-1}. Following a similar principle, we compute $f_{\Delta_{i1}}(t)$ for the first chunk of a video i as

$$f_{\Delta_{i1}}(t) = f_{\Delta_{(i-1)1}}(t) * f_{\kappa_{i-1}}(t). \tag{9}$$

For subsequent chunks c_{ij}, we also calculate the playing start distribution based on the first chunk

$$f_{\Delta_{ij}}(t) = f_{\Delta_{(i-1)i}}(t-(j-1)\cdot L)\cdot\left(1-\int_0^{(j-1)\cdot L} f_{\kappa_i}(x)\right)dx \tag{10}$$

Then the expected rebuffering function can be calculated similarly to Eq. 7:

$$\mathbf{E}^{rebf}_{c_{ij}}(x) = \int_{t=0}^{x} f_{\Delta_{ij}}(t) \times T^{rebf}_{c_{ij}}(x,t)dt \tag{11}$$

In the implementation, we approximate the continuous value swipe distribution with a discrete distribution with the time granularity of 0.1 seconds. The integral then can be approximated by the summation in the discrete distribution.

4.2 Determining Buffering Sequences

Given the preceding computation of expected rebuffering delay for each chunk, Dashlet's next task is to determine an order of chunks to download (*i.e.*, a buffering sequence) that minimizes expected rebuffering delay over a lookahead horizon. Prior schemes (*e.g.*, MPC [40]) can then be used to determine the bitrates for those chunks to optimize overall QoE for the horizon. However, unlike prior schemes, the horizon that we use is based on time (not chunks), since different user swipe patterns can translate into different numbers of viewed chunks. Using a horizon sized to a fixed number of chunks could result in optimization over very short viewing times (negating the effects of longer-term planning). Our current implementation uses a lookahead window of 25 seconds based on empirical observations, which is equivalent to the five chunks MPC uses. Chunk ordering relies primarily on whether the user swipes near the beginning of the video or not: *e.g.* if the user is highly likely to not swipe in c_{11}, the algorithm then needs to prioritize c_{12} over c_{21}.

4.2.1 Selecting the candidate chunk set

To determine the set of chunks to consider, we enforce a threshold on the minimum rebuffering penalty that each chunk is expected to incur at the end of the horizon if it is not included in the buffer sequence (Fig. 14(a)). Chunks whose rebuffering penalty falls below the threshold are deemed as unlikely to be viewed during the horizon (c_{32} in Fig. 14(a)), and thus low priority for inclusion in the buffer sequence.

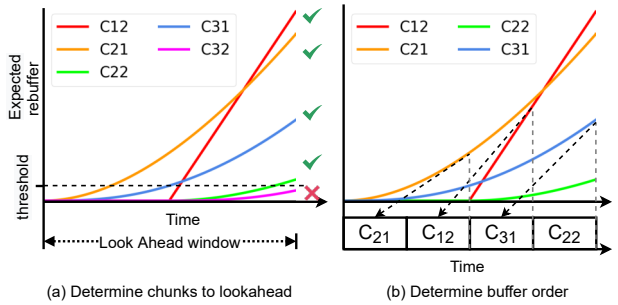


Figure 14: An example to illustrate Dashlet's algorithm.

Note that buffer sequences are constructed each time a chunk download completes, so an excluded chunk for one horizon may still be downloaded shortly (via inclusion in the next horizon's buffer sequence). We use an empirically-configured value of $1/\mu$ for threshold, which is the inverse of the rebuffering penalty weight in our target QoE function.

Using the set of chunks to consider, our final task is to order them in a manner that minimizes expected rebuffering penalties. We assign a bitrate to each chunk, and then use estimated network bandwidth to determine when it will complete donwloading (assuming some start time). This allows us to compute expected rebuffering time per chunk (§4.1). However, to bound computational complexity (since download decisions must be fast) we temporarily assume an equal bitrate per chunk that is set to the maximum bitrate, which ensures that all chunks in the list will complete downloading before the horizon completes. Although exclusion of per-chunk bitrate decisions here can result in suboptimal orderings, these effects are marginal (evidenced by Dashlet's closeness to the Optimal scheme in §5.2), as priorities between chunks (and potential per-chunk viewing times) are largely dictated by viewing constraints imposed by the application (§4.1). Thus, minor discrepancies in bitrates across chunks are unlikely to flip the priority order among them.

4.2.2 Priority-ordering the buffer sequence

To sort our list of chunks into a buffer sequence, we follow a greedy algorithm, whereby we partition the horizon into chunk-sized slots. For a given slot i, we select the chunk that will incur the largest additional rebuffering penalty if it were to be scheduled in slot $i+1$ rather than i. Fig. 14(b) shows this process for a scenario in which chunk c_{11} just completed downloading: c_{21} is assigned to slot 1 as its rebuffering penalty jumps the most between slots 1 and 2; c_{12} is next as it has the highest penalty for not going in slot 2, and so on. Finally, using the generated buffer sequence, Dashlet applies MPC's algorithm to determine the bitrate for each chunk in the buffer sequence in a way that optimizes the entire QoE (not just minimizing rebuffering) for the horizon according to the forecasted network throughput, *i.e.*, the harmonic mean over the observed throughputs in the last 5 chunk downloads. We describe the above algorithm with a pseudo code in §A.

4.3 Implementation

Dashlet's implementation includes one control module and multiple buffer modules. The control module schedules the chunk downloading and the buffer modules reuses the DASH.js playback management implementation to download video chunks. §B provides more implementation details.

5 Evaluation

We evaluate Dashlet across a variety of mobile network conditions, real user swipe traces, and videos. Our key findings:

- Dashlet outperforms TikTok by 28-101% in terms of average QoE, including 8-39% improvement on video bitrate, 1.6-8.9× reduction on rebuffering penalty, and 30% reduction on data wastage.
- Dashlet's QoE improvement varies with the network throughput, *i.e.*, 543.7%, 221.4%, and 36.6% over TikTok when the throughput is 2-4, 4-6, and 10-12 Mbps, respectively. The improvement diminishes with throughput approaching to 20 Mbps because both Dashlet and TikTok are getting closer to optimum.
- Dashlet tolerates errors in swipe distributions: with errors of 50%, Dashlet makes the correct buffering decisions 96.5% of the time, yielding an QoE reductions of only 10%. compared to cases with no distribution errors.

5.1 Methodology

Baselines. We compare Dashlet with the following systems:

- *TikTok.* We compare with TikTok App (version v.20.9.1).
- *Oracle.* We also run an 'oracle' baseline that serves as an upper bound for QoE. The oracle is the RobustMPC algorithm [40] running with perfect (a priori) knowledge of both the user swipe traces and network throughput in each experiment. With that information, the algorithm knows the upcoming video viewing sequence at all times, and can thus pick the buffer sequences (and bitrates) that directly optimize QoE for the current network conditions.

Overall setup. All video clients run on a rooted Pixel 2 phone (Android 10). The Oracle algorithm and Dashlet run in the Google Chrome browser (v. 97.0.4692.87), and contact a local desktop which houses the videos accessed in each experiment (described below). In contrast, TikTok runs as an unmodified, native Android app and contacts Akamai CDNs to fetch video content as it normally does. We checked the location of the CDN content server node and verified it was local to our area. All traffic to and from the phone passes over emulated mobile networks (which run atop WiFi connections with average speeds of ≈300 Mbps); to compensate for the discrepancy in video servers, we added 6 ms of round trip

delay to traffic for Dashlet and the Oracle algorithm, which reflects the maximum ping time we observed to the CDN used by TikTok.

Evaluation metrics. Short videos share similar goals of traditional video streaming [22, 40]: maximizing video bitrate, minimizing rebuffering delays, and avoiding frequent bitrate fluctuations, so we adopt a widely used QoE metric:

$$QoE = R_{bitrate} - \mu \cdot P_{rebuffer} - \eta \cdot P_{smooth} \quad (12)$$

where $R_{bitrate}$ is the average video bitrate, $P_{rebuffer}$ is the cumulative penalty for rebuffering (i.e., stalled playback), and P_{smooth} is the penalty for frequent bitrate switching across adjacent chunks. We use the same values for μ and η as prior work [40], i.e., $\mu = 3000$ and $\eta = 1$.

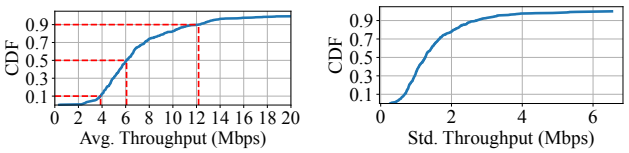
Human subjects study for QoE. We conduct a small-scale human study, where we recruit ten participants[2]. We ask them to log in their own accounts to use TikTok under emulated mobile networks (the videos are recommended by TikTok)[3]. We randomly choose three network traces with average throughput of 4 ± 0.1, 6 ± 0.1, 12 ± 0.1 Mbps respectively. We record the content, quality, order of videos TikTok streams to each user, and swipe timestamps. For the evaluation of Dashlet, we first download every video that users have watched in TikTok experiments and collect per-video user swipe distributions with Amazon Mturk. We then stream the same videos in the same order as TikTok using Dashlet, under the same emulated network, during which we replay the user swipes recorded in the TikTok experiment. Take the analogy to machine learning: the "training set" we use for Dashlet is collected by MTurk, and the testing set is real users' swipe. To quantify performance, we record the quality of every video chunk and the rebuffering event to calculate the QoE for both TikTok and Dashlet.

Human subjects study for users' satisfaction. We let the same group of participants use TikTok and Dashlet for in total 30 minutes and ask them to rate the video quality and smoothness after they finish. Each participant used both TikTok and Dashlet for three five-minute sessions under three different network traces. Notice that the videos played in TikTok and Dashlet are different in the study since the users would behave differently if shown the same video once more, e.g., users tend to swipe fast when they are already familiar with the content in a video. For Dashlet, the swipe distribution is pre-collected with MTurk before the study.

Trace-driven study. We run a trace-driven study to scale up the evaluation under different user swipe speed and network traces. We use a script to automatically swipe in TikTok

[2]Among the ten participants, three of them are new users, three of them are occasional users, and four of them are daily users.

[3]The only action that users perform in the study is to swipe to the next video based on their watching experiences



(a) Average. **(b)** Standard deviation.

Figure 15: Throughput distribution for our network dataset.

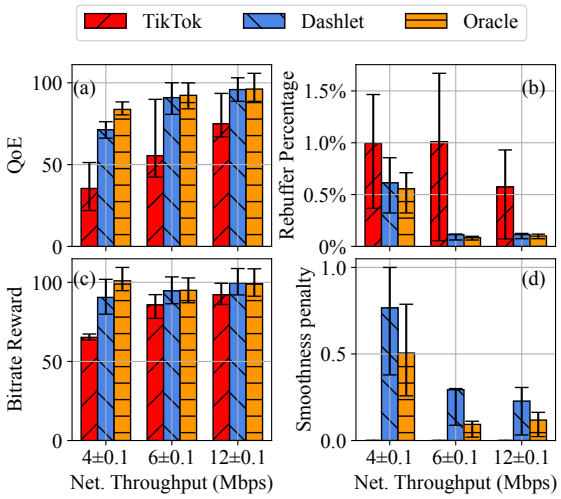


Figure 16: End-to-end result for human subjects study. (a) QoE (b) Rebuffer percentage. (c) Bitrate reward (d) Smoothness penalty

based on the distribution shown in Fig. 7. In order to enforce the same playing sequence (i.e., ordered list) of videos across the considered systems, we exploit the fact that the order in which videos are streamed with TikTok for a given keyword search remains unchanged on the order of many days. We use that same order across all systems and across experiments with different network and swipe traces. Each experiment considers 10 minutes of viewing time to match the average session time for TikTok users [34]. Similar to our human subjects study, we replay the same traces recorded from TikTok experiments to evaluate Dashlet and Oracle. The swipe distribution used for Dashlet in replay is collected from another batch of user study via Amazon Mturk.

Network conditions. We consider the combination of two sets of mobile network traces: (1) the FCC LTE dataset [9] that is widely used in prior work [22, 40], and (2) a WiFi trace dataset that we collected in January 2022 in a shopping mall. Fig. 15 shows the average and standard deviation of throughput traces in the combined dataset.

5.2 End-to-End performance

Human subjects study. Fig. 16 shows the end-to-end result for human subjects study, including QoE, rebuffering

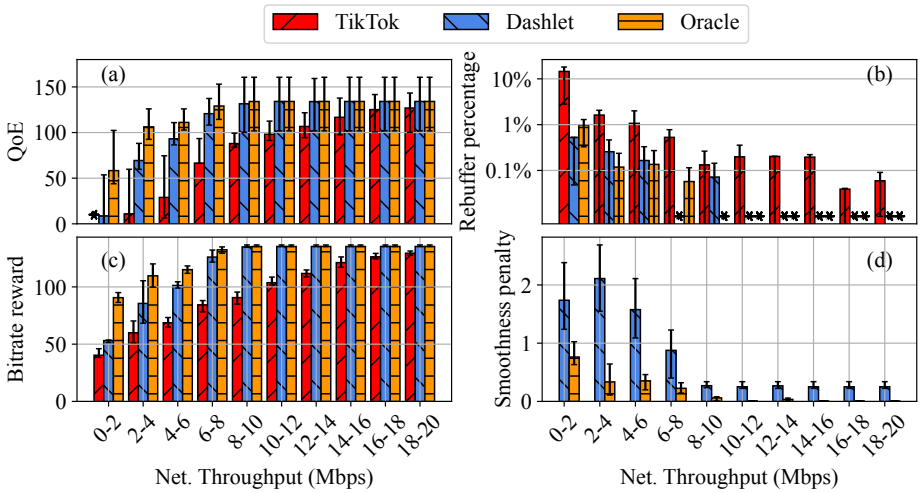


Figure 17: End-to-end result for trace-driven study. (a) QoE. * is the outlier data point with average QoE at -389 due to rebuffer (b) Rebuffer percentage (of total time). Note log ordinate axis; * denotes zero rebuffering. (c) Bitrate reward. (d) Smoothness penalty

Network throughput (Mbps)	**4 ± 0.1**	**6 ± 0.1**	**12± 0.1**
TikTok quality	3.1 ± 0.83	3.2 ± 0.87	4.0 ± 0.89
Dashlet quality	3.6 ± 0.80	3.9 ± 0.70	4.1 ± 0.94
TikTok stall	2.8 ± 1.08	3.0 ± 0.77	4.2 ± 0.99
Dashlet stall	3.5 ± 1.02	3.9 ± 0.94	4.3 ± 0.90

Table 1: User survey for TikTok and Dashlet. Each participant is asked to score 1 (worst) to 5 (best) in terms of video quality (resolution) and stall (rebuffer) under three different network throughput. The sample questionnaire is shown in §D. Table summarizes the average and standard deviation of the score.

Network throughput (Mbps)	**4 ± 0.1**	**6 ± 0.1**	**12± 0.1**
QoE	-363.2	-287.9	-133.5
Rebuffer percentage	28.0%	24.8%	14.3%
Bitrate reward	77.2	96.6	97.8
Smoothness Penalty	0.38	0.12	0.02

Table 2: End-to-end result for MPC.

percentage, bitrate reward and smoothness penalty. There are two key takeaways from these results. First, Dashlet consistently outperforms TikTok across different network throughput. Dashlet improve the average QoE over TikTok by 101%, 64%, 28% on 4 Mbps, 6 Mbps, 12 Mbps respectively. When break down the QoE into the components, Dashlet reduces the rebuffering by 1.6-8.9x compared with TikTok and improve the QoE by 8% - 39% with the cost of marginal smoothness penalty. Second, Dashlet can reach the close-to-optimal performance starting from 6 Mbps. While TikTok does not achieve that even at 12 Mbps.

We also run experiments on MPC [40], a state of art traditional video streaming algorithm, on the same setup mentioned above. As a traditional video streaming algorithm, MPC only prebuffers chunks for the current video. Table 2 summarizes the end-to-end result. Compared with Dashlet, MPC incurs a much higher rebuffering as it experiences rebuffer delay every time the user swipes to a new video., leading a significant lower QoE compared with Dashlet.

We also perform a experiment to understand the participants' satisfaction of the service provided by TikTok and Dashlet. We let the participants watch videos using TikTok and Dashlet for five minutes, after which we conduct a user survey by asking the participant to report their satisfaction scores in terms of video quality (resolution) and stall (rebuffer) conditions for both TikTok and Dashlet. Table 1 shows the users' satisfaction towards the video resolution and rebuffer for both TikTok and Dashlet on the human subjects study. From the figure, we can see that Dashlet improves the users satisfaction on both video resolution and rebuffering.

Trace-driven study. Fig. 17 shows the result for trace-driven study. Key results are: (1) Dashlet's QoE improvement varies with the network throughput, *i.e.*, 543.7%, 221.4%, and 36.6% over TikTok when the throughput is 2-4, 4-6, and 10-12 Mbps, respectively. The improvement diminishes with throughput approaching to 20 Mbps. (2) Dashlet can reach the optimal QoE at a much lower network throughput than TikTok, *i.e.* Dashlet reaches the optimal at throughput 8-10 Mbps. While TikTok is close to the optimal at the throughput 18-20 Mbps. (3) Dashlet consistently incurs a lower rebuffering compared with TikTok.

5.3 Ablation study

We further perform an ablation study to understand the contribution of five design components (detailed in Table 3). *Idle:* TikTok has a prebuffer idle state as described in §2.2.1 while Dashlet does not. *Chunking:* TikTok splits the video into one or two chunks while Dashlet splits the video into

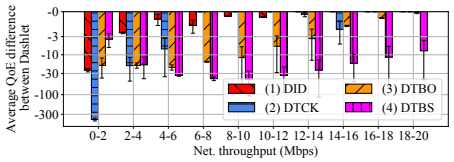


Figure 18: The contribution to End-to-End QoE for different design components. The y-axis in log scale is the QoE difference between the corresponding ablation study systems and Dashlet.

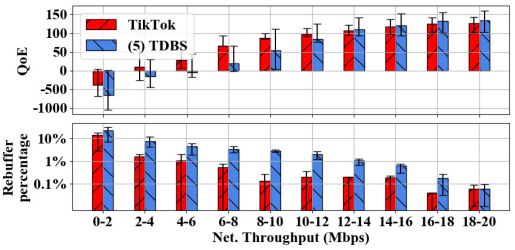


Figure 19: Ablation study for bitrate choice.

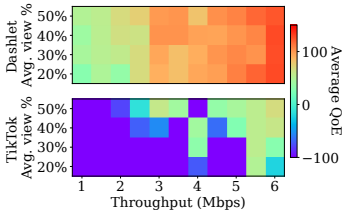


Figure 20: Average QoE of Dashlet and TikTok under different average viewing percentage (based on swipe patterns) and network throughput.

System Name	Idle	Chunking	Fix bitrate	Buffer order	Bitrate selection
(1) Dashlet+Prebuffer idle (DID)	T	D	D	D	D
(2) Dashlet+TikTok Chunking (DTCK)	D	T	T	D	D
(3) Dashlet+TikTok buffer order (DTBO)	D	D	D	T	D
(4) Dashlet+TikTok bitrate (DTBS)	D	D	D	D	T
(5) TikTok+Dashlet bitrate (TDBS)	T	T	T	T	D

Table 3: Setup for ablation study. We summarize the difference in design components between Dashlet and TikTok and evaluate the impact of corresponding design components. **"T"** and **"D"** denote TikTok's and Dashlet's design components respectively.

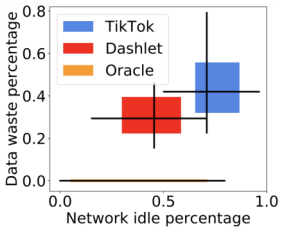


Figure 21: Data wastage and network idle time. Boxes span 25-75th percentiles. Black lines span min/max, and intersect at the median for both properties.

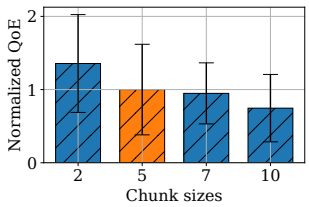


Figure 22: Dashlet's chunk duration's impact on QoE (normalized for 5-second chunk). Bars list averages, with error bars for one st. dev. in each direction.

various number of equal-length chunks. Notice that Tiktok's chunking also leads to a *fix bitrate* for chunks in the same video. *Buffer order:* whether the system follows TikTok or Dashlet's buffer order. *Bitrate selection.* whether the system follows TikTok or Dashlet's bitrate selection. We implement the TikTok's logic for the corresponding design components according to our TikTok analysis in §2.2. For the bitrate selection, we use a lookup table to record the TikTok's bitrate decision under different network throughput and buffer level. Our implementation for TikTok's bitrate choice will then make the decision according to the look-up table.

We first investigate the performance drop when replacing Dashlet's design components with the corresponding TikTok's design component. Fig. 18 shows the QoE difference compared to Dashlet. Dashlet+Prebuffer idle (DID) curve shows that having a prebuffer idle state will have a significant negative impact at low throughput (*e.g.* 0-2 Mbps). But when the impact diminishes as the network throughput is above 4 Mbps. Similarly, TikTok's chunking also has a significant negative impact at the low network throughput. The low network throughput forces TikTok to choose a low video bitrate, but consequently increases the first chunk duration, making TikTok more vulnerable to rebuffering when swipe happens. TikTok's buffering order (DTBO) selection has significant negative impact on QoE until the throughput reaches 14 Mbps. The bitrate selection (DTBS) have the most significant impact on the QoE. Its impact to the QoE dominants as the network throughput reaches 4-6 Mbps. By digging deep in the reason, we find TikTok is very conservative in choosing high bitrate. We show the detail in §C. One natural next question arises is that could we just simply increase the request bitrate to improve the QoE. To answer this question, we consider another ablation study case **TDBS**, which includes TikTok's design for all other components but keeps the high bitrate choices as Dashlet. Fig. 19 shows the comparison between TDBS and TikTok. with the higher bitrate choices, TDBS performs worse than TikTok when the network throughput is less than 12 Mbps. The key reason behind is that TDBS has a higher rebuffer percentage compared with TikTok. The takeaway is that TikTok's low bitrate is a result of adaptation to avoid rebuffering. Simply increase the downloading bitrate could lead to a worse QoE.

5.4 Micro Benchmarks

Impact of Swipe and Network Speeds on QoE. Patterns in network throughput and user swipes largely influence the performance of short video streaming algorithms. To understand the effect of each, we report Dashlet's and TikTok's results for different network throughputs and swipe rates. As shown in Fig. 20, the major factor that affects QoE with Dashlet is the network throughput. Importantly, swipe speed does not have a significant impact on Dashlet's performance, validating its robustness under different swipe patterns. In contrast, both network throughput and swipe speed have a large impact on TikTok's QoE.

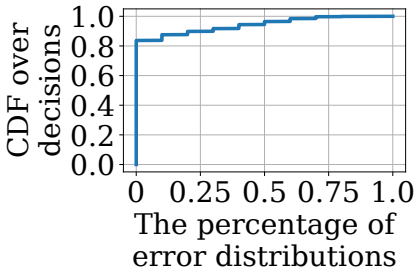


Figure 23: Dashlet's tolerance to swipe distribution errors.

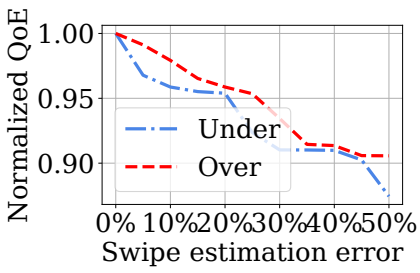


Figure 24: Impact of swipe estimation errors on Dashlet.

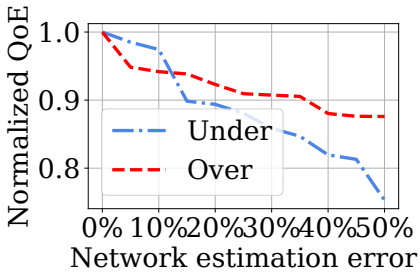


Figure 25: Impact of network estimation errors on Dashlet.

Network Idle and Data Waste. To dig deeper into Dashlet's performance gain over TikTok, we investigate network idling and data wastage for both systems. Fig. 21 shows our results; note that the Oracle algorithm does not incur any data wastage since it has perfect knowledge of user swipe times. As shown, median data wastage and idle time for Dashlet are 29.4% and 45.5%, respectively, which are 30.0% and 35.9% lower than those with TikTok. These improvements enable Dashlet to stream video at higher bitrates than TikTok while keeping rebuffering delays low.

The impact of chunk size on Dashlet's QoE. Unlike TikTok which breaks up videos by bytes, Dashlet (by default) breaks up videos into 5-second chunks. We evaluated the impact that chunk sizes have on Dashlet's performance by considering the following chunk sizes (based on prior work [42]): {2, 5, 7, 10} seconds. Note that we did not modify chunk sizes for TikTok as we could not alter its video servers. As shown in Fig. 22, Dashlet's performance decreases as chunk sizes grow, *e.g.*, average QoE drops by 35.4% as chunk sizes grow from 5 to 10 seconds. The reason is that data wastage grows with larger chunk sizes: a user swipe at 1 second into a chunk will result in more wasted bytes with a larger chunk size.

Decision Stability with Swipe Prediction Errors Dashlet determines buffer sequences by leveraging (coarse information from) users' swipe distributions for each video. Thus, a natural question is how robust are Dashlet's decisions to errors in those distributions, i.e., does it make the right decisions even with different degrees of errors?

Recall that there are three inputs to Dashlet's algorithm at any time: the swipe distribution for each considered video, the estimated network throughput, and the client-side player's current buffer state. The algorithm uses this information and returns a buffer sequence of chunks to download, with the first chunk in the ordered list indicating the action to perform immediately, i.e., the chunk to download now. To answer the above question, we profiled the above inputs throughout our experiments, and then compared the actions selected by Dashlet with those that it would select if the input swipe distribution involved errors. In particular, we considered 10 versions of each video's distribution by (roughly) modeling its original distribution as an exponential one, and then altering the corresponding λ value to change the average swipe time by $1\pm\{0\text{-}50\%\}$ (in 10% increments).

Fig. 23 shows our results. As shown, 83.7% of Dashlet's decisions are unchanged across all of the considered distribution errors. The values remain relatively stable as errors grow – *e.g.*, 96.5% of Dashlet's decisions are unchanged with errors of 50% – but begin to grow after 82%. These results illustrate that Dashlet only relies on coarse information from swipe distributions (*e.g.*, about whether a user is likely to swipe early or late in the video); it is for this reason that decisions are varied only when errors are very high (and even the coarse information that Dashlet uses has changed).

QoE sensitivity with Swipe and Network Errors Building on the previous results, we now analyze how errors in swipe distribution affect the QoE that Dashlet delivers. We ran Dashlet on all videos and the network traces using same faulty distributions from above. Fig. 24 shows the results, breaking them down in terms of scenarios with over estimation of swipe times (longer average viewing time than the correct distribution, i.e., later swipes) and under estimation (shorter average viewing time). As shown, Dashlet is quite tolerant to such errors, delivering 87% and 91% of its full QoE (with no errors) when the traces are over- and under-estimating swipe times by 50%.

We perform a similar analysis to evaluate Dashlet in the presence of network prediction errors. Specifically, we replace the network predictor in RobustMPC [40] with one that reads in the actual instantaneous throughput from the current Mahimahi trace, and multiplies that value by between $1\pm\{0\text{-}50\%\}$. Overall, as per Fig. 25, we find that Dashlet's QoE drops to 88% and 76% of its values without network errors when the network estimate is over- or under-estimating by 50%. These results highlight that Dashlet is more robust to errors in swipe distributions than network forecasts.

6 Related work

Traditional adaptive video streaming Traditional video streaming services deliver video content from the CDN to the user with adaptive bitrate system with the objective of

maximizing the quality of experience for users [20]. Research effort has been made to improve the quality of experience from different perspectives, including streaming algorithm [18, 19, 22, 29, 40], video codec [7, 10], network prediction [30, 36], protocol design [14, 45], and video super resolution [38, 39]. But all these optimization is for the same video streaming model: the video download sequence is the same as the video playing sequence. Dashlet also uses QoE as the optimization goal but tackles a different problem as in the short video streaming the video download sequence is the same as the video playing sequence due to users' swipes.

Streaming new form of video There are also rising interest on 360 degree video [12, 26] and volumetric video streaming [25]. These systems need to handle the uncertainty from the users' head position or location. Dashlet's design also models the uncertainty from the user swipe patterns. But Dashlet targets on a different problem compared with 360 degree or volumetric video streaming. Some existing works [15, 27] also try to apply reinforcement learning algorithms from traditional video streaming [22] to short video streaming. However, these works do not factor in the impact of user swipes on buffering decisions as Dashlet does.

7 Discussion

TikTok version. Our reverse engineering tool can only deciphers the HTTPS messages transmitted by TikTok with version up to v20.9.1. As a result, we cannot conduct our case study (§2) using the up-to-date TikTok v26.3.3, which adopts a different encryption method as V20.9.1. We leave the task of deciphering the HTTPS messages and thus studying the streaming algorithm of the newest version of TikTok as our future work.

Backward swipes, fast-forwarding, and pause. Our current model only allows forward swipes, *i.e.*, swipes to watch next videos. The newest version of TikTok also allows backward swipes where the user swipes to watch the previous video and fast-forwarding, where the user speeds up the playback of the current video: we will study these in future work. In addition, our model does not consider the video pause. The pausing of videos will make it easier for the system since it gives the player more time to download videos. For Dashlet's design, we focus on a harder problem, which assumes no pause in the video.

Diminished gain at higher network speeds. We observe a diminished improvement for Dashlet over TikTok at higher network speeds. At higher network speeds, mistakes made by TikTok are masked by higher network throughput. As network speed increases, TikTok can pick up the highest bitrate but still have enough time to react to users' wipes. In our evaluation, we use the bitrates that TikTok's CDN offers, which are capped at 720P video quality. We expect the gap between Dashlet and TikTok will widen if higher bitrate videos are used to evaluate both systems, which we expect will happen in the future.

Generalization of Dashlet design. The Dashlet design does not rely on the design of TikTok but only relies on a sequence of videos that are played in chunks. Therefore, it should be able to generalize to other platforms like YouTube Shorts and Instagram Reels.

Energy implication to smartphones. Dashlet could potentially reduce the energy consumption for short video applications. The energy cost includes both the cost to run the algorithm and the cost to download the video content. Dashlet uses a simple non-machine learning algorithm, which causes minimal extra energy overhead. For the cost to download the video content, Dashlet has less energy overhead since its waster download is much less than TikTok.

Evaluation generalizability. We have conducted a small scale human study to compare the performance of Dashlet and TikTok, where ten participants log into their own account to watch TikTok video on a emulated mobile network, repeating the experiments using Dashlet. The personality of the recruited user may lead to biased results, for example, a patient user may tend to not swipe or swipe at the end of the video, leaving larger time window for TikTok to download the second chunk. A larger scale human study that involves more diverse users is needed to eliminate this potential bias.

We conduct our evaluation under emulated mobile networks, but prior work [36, 37] has pointed out that the emulation based evaluation of network applications and congestion control schemes may not always be indicative of real-world performance. For example, although we compensate the average round trip delay to the CDN server in the emulated environment, the variation in the round trip delay might potentially impact the results. While we note that Dashlet does not input network measurements into an ML model, we acknowledge that large-scale evaluation in the wild may be required to verify the full generalizability of our results.

8 Conclusion

In this paper, we design and implement Dashlet with the insight provided by measurement for a commercial short video app and a user study on general user swipe pattern. Dashlet's algorithm strategically determines the buffer order with the input from a coarse-grained swipe distribution. Evaluation result shows Dashlet significantly improves video quality and reduces rebuffering compared with the baseline system.

9 Acknowledgement

We thank the anonymous reviewers and our shepherd, Sanjay Rao for their insightful comments. This material is supported in part by NSF CNS grants 2152313, 2153449, 2147909, 2140552, and 2151630, Sloan Research Fellowship, as well as a grant from the Princeton School of Engineering and Applied Science.

References

[1] 24 Important TikTok Stats Marketers Need to Know in 2022. https://blog.hootsuite.com/tiktok-stats/.

[2] AKHTAR, Z., NAM, Y. S., GOVINDAN, R., RAO, S., CHEN, J., KATZ-BASSETT, E., RIBEIRO, B., ZHAN, J., AND ZHANG, H. Oboe: Auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 44–58.

[3] CHEN, F., LI, P., ZENG, D., AND GUO, S. Edge-assisted short video sharing with guaranteed quality-of-experience. *IEEE Transactions on Cloud Computing* (2021), 1–1.

[4] CHEN, Z., HE, Q., MAO, Z., CHUNG, H.-M., AND MAHARJAN, S. A study on the characteristics of douyin short videos and implications for edge caching. In *Proceedings of the ACM Turing Celebration Conference-China* (2019), pp. 1–6.

[5] CORTESI, A., HILS, M., KRIECHBAUMER, T., AND CONTRIBUTORS. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 8.0].

[6] DASARI, M., BHATTACHARYA, A., VARGAS, S., SAHU, P., BALASUBRAMANIAN, A., AND DAS, S. R. Streaming 360-degree videos using super-resolution. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications* (2020), IEEE Press, p. 1977–1986.

[7] DASARI, M., KAHATAPITIYA, K., DAS, S. R., BALASUBRAMANIAN, A., AND SAMARAS, D. Swift: Adaptive video streaming with layered neural codecs. In *USENIX NSDI* (2022).

[8] DASH Industry Forum. https://reference.dashif.org/dash.js/latest/samples/index.html.

[9] Federal Communications Commission. 2016. Raw Data - Measuring Broadband America. (2016). https://www.fcc.gov/reports-research/reports/.

[10] FOULADI, S., EMMONS, J., ORBAY, E., WU, C., WAHBY, R. S., AND WINSTEIN, K. Salsify:{Low-Latency} network video through tighter integration between a video codec and a transport protocol. In *USENIX NSDI* (2018).

[11] GAO, X., LE CALLET, P., LI, J., LI, Z., LU, W., AND YANG, J. *QoEVMA'20: 1st Workshop on Quality of Experience (QoE) in Visual Multimedia Applications*. Association for Computing Machinery, New York, NY, USA, 2020, p. 4771–4772.

[12] GUAN, Y., ZHENG, C., ZHANG, X., GUO, Z., AND JIANG, J. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *ACM SIGCOMM* (2019).

[13] GUO, J., AND ZHANG, G. *A Video-Quality Driven Strategy in Short Video Streaming*. Association for Computing Machinery, New York, NY, USA, 2021, p. 221–228.

[14] HAN, B., QIAN, F., JI, L., AND GOPALAKRISHNAN, V. Mp-dash: Adaptive video streaming over preference-aware multipath. In *CoNEXT* (2016).

[15] HE, J., HU, M., ZHOU, Y., AND WU, D. Liveclip: towards intelligent mobile short-form video streaming with deep reinforcement learning. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video* (2020).

[16] HUANG, T.-Y., JOHARI, R., MCKEOWN, N., TRUNNELL, M., AND WATSON, M. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, Association for Computing Machinery, p. 187–198.

[17] ITSEEZ. *The OpenCV Reference Manual*, 4.5.1 ed., 2022.

[18] JIANG, J., SEKAR, V., AND ZHANG, H. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *ACM CoNEXT* (2012).

[19] KIM, J., JUNG, Y., YEO, H., YE, J., AND HAN, D. Neural-enhanced live streaming: Improving live video ingest via online learning. In *ACM SIGCOMM* (2020).

[20] KRISHNAN, S. S., AND SITARAMAN, R. K. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking* (2013).

[21] LYMBEROPOULOS, D., RIVA, O., STRAUSS, K., MITTAL, A., AND NTOULAS, A. Pocketweb: Instant web browsing for mobile devices. In *ASPLOS 2012 (Architectural Support for Programming Languages and Operating Systems)* (March 2012), ACM.

[22] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM* (2017).

[23] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate {Record-and-Replay} for {HTTP}. In *USENIX ATC* (2015).

[24] PyAutoGUI - PyPI. https://pypi.org/project/PyAutoGUI/.

[25] QIAN, F., HAN, B., PAIR, J., AND GOPALAKRISHNAN, V. Toward practical volumetric video streaming on commodity smartphones. In *ACM HotMobile* (2019).

[26] QIAN, F., HAN, B., XIAO, Q., AND GOPALAKRISHNAN, V. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *ACM MobiCom* (2018).

[27] RAN, D., ZHANG, Y., ZHANG, W., AND BIAN, K. Ssr: Joint optimization of recommendation and adaptive bitrate streaming for short-form video feed. In *IEEE MSN* (2020).

[28] Genymobile/scrcpy: Display and control your Android device. https://github.com/Genymobile/scrcpy.

[29] SPITERI, K., URGAONKAR, R., AND SITARAMAN, R. K. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking 28*, 4 (2020), 1698–1711.

[30] SUN, Y., YIN, X., JIANG, J., SEKAR, V., LIN, F., WANG, N., LIU, T., AND SINOPOLI, B. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *ACM SIGCOMM* (2016).

[31] TikTok: Thanks a billion! https://newsroom.tiktok.com/en-us/1-billion-people-on-tiktok.

[32] TikTok Revenue and Usage Statistics (2022). https://www.businessofapps.com/data/tik-tok-statistics/.

[33] https://influencermarketinghub.com/tiktok-stats/. https://influencermarketinghub.com/tiktok-stats/.

[34] TikTok User Statistics (2022). https://backlinko.com/tiktok-users.

[35] WANG, S. W., YANG, S., LI, H., ZHANG, X., ZHOU, C., XU, C., QIAN, F., WANG, N., AND XU, Z. Salientvr: Saliency-driven mobile 360-degree video streaming with gaze information. In *Proceedings of the 28th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2022), MobiCom '22, Association for Computing Machinery.

[36] YAN, F. Y., AYERS, H., ZHU, C., FOULADI, S., HONG, J., ZHANG, K., LEVIS, P., AND WINSTEIN, K. *Learning in Situ: A Randomized Experiment in Video Streaming*. USENIX Association, USA, 2020, p. 495–512.

[37] YAN, F. Y., MA, J., HILL, G. D., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon: the training ground for internet congestion-control research. In *USENIX ATC* (2018).

[38] YEO, H., CHONG, C. J., JUNG, Y., YE, J., AND HAN, D. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In *ACM MobiCom* (2020).

[39] YEO, H., JUNG, Y., KIM, J., SHIN, J., AND HAN, D. Neural adaptive content-aware internet video delivery. In *USENIX OSDI* (2018).

[40] YIN, X., JINDAL, A., SEKAR, V., AND SINOPOLI, B. A control-theoretic approach for dynamic adaptive video streaming over http. In *ACM SIGCOMM* (2015).

[41] YouTube Shorts Video-Making App Now Receiving 3.5 Billion Daily Views. https://www.latestly.com/technology/youtube-shorts-video-making-app-now-receiving-3-5-bil html.

[42] ZHANG, T., REN, F., CHENG, W., LUO, X., SHU, R., AND LIU, X. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in dash. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications* (2017), IEEE, pp. 1–9.

[43] ZHANG, X., OU, Y., SEN, S., AND JIANG, J. SENSEI: Aligning video streaming quality with dynamic user sensitivity. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 303–320.

[44] ZHANG, Y., LIU, Y., GUO, L., AND LEE, J. Y. Measurement of a large-scale short-video service over mobile and wireless networks. *IEEE Transactions on Mobile Computing* (2022).

[45] ZHENG, Z., MA, Y., LIU, Y., YANG, F., LI, Z., ZHANG, Y., ZHANG, J., SHI, W., CHEN, W., LI, D., ET AL. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *ACM SIGCOMM* (2021).

A Dashlet Pseudocode

Algorithm 1: Dashlet's ABR algorithm

Input : 1) Buffer status $r_1 \dots r_n$
2) Video bitrate $B = \{b_{11} - b_{1k} \dots b_{n1} - b_{nk}\}$
3)The probability distribution of play start time for each chunk $\hat{G} = \hat{g}_{11}(t) \dots \hat{g}_{nc_n}(t)$
4) network throughput estimation T
5) The look-ahead time length F
6) Chunk length L

Output :The location and bitrate to buffer next

foreach $i, j \in \{1 \dots n\}$ **do**
  if $\int_0^F (F-t)\hat{g}_{c_{ij}}(t)dt > 1/\mu$ ***and*** $j > r_i$ **then**
    $candidateList.append(c_{ij})$;
  //Add the chunk to the candidate list if there is significant penalty for not downloading it
$targetBitrate = F \times T$ / $len(candidateList)$ / L;
do
  c_{min} = `minRebufferCost` ($targetBitrate$, $bufferOrder$, $candidateList$);
  $bufferOrder.append(c_{min})$;
  $candidateList.remove(c_{min})$;
while $len(candidateList) > 0$;
//use greedy algorithm to put chunks from *candidateList* into *bufferOrder*
$bitrateList$ = `getMaxBitrate` ($bufferOrder$, B, T);
//Enumerate all the bitrate combination for chunks in *bufferOrder* to maximize the QoE
Return $bufferOrder[0]$, $bitrateList[0]$

B Dashlet Implementation Further Detail

Dashlet makes no change to the CDN/server side so our system can be easily deployed client side. Dashlet includes one control module and multiple buffer modules. Each buffer module manages the video playback of one short video, including downloading chunks, tracking playback progress, and reporting buffer status. We reuse the DASH.js playback management for the buffer modules. The control module manages scheduling across short videos, collecting estimated throughput and buffer length from each buffer module. With the collected data, control module runs Dashlet's algorithm to schedule the video buffering. Based on the algorithm's output, it assigns the quota to the buffer module that is assigned to download the next video chunk. The quota includes the target video bitrate and the target download finish time. Once the buffer module receives the quota, it sends an HTTP request with target bitrate to the CDN to download s the corresponding video chunk. A call back function is set to report the status to control module in case the download time exceeds the target download finish time. The control module schedules the video buffering when the call back function for target download time is triggered, the chunk download finishes, or the user swipes. Similar to Pensieve [22], we also use an ABR server to run Dashlet's algorithm on the same machine as the client. The control module communicates with the ABR server using XMLHttpRequests locally.

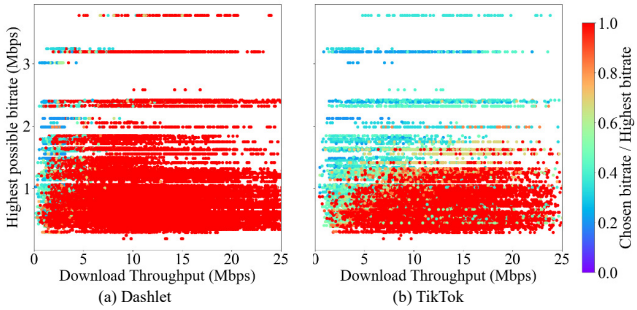


Figure 26: Bitrate choice made by Dashlet and TikTok. The x-axis is the network throughput and the y-axis is the highest available bitrate to choose. The color is the ratio between the chosen bitrate and the highest available bitrate. The red color means the highest available bitrate is chosen.

1. How will you rate the quality of the video you watched on **TikTok**
   a. Excellent, all the videos are high quality (5)
   b. Good, the video quality is generally good (4)
   c. OK, there are some blur videos but do not affect my watching experience (3)
   d. Need improve, there are a lot of blur videos and affect my watching experience (2)
   e. Bad, the videos are not clear at all (1)

2. How will you rate the quality of the video you watched on **Dashlet**
   a. Excellent, all the videos are high quality (5)
   b. Good, the video quality is generally good (4)
   c. OK, there are some blur videos but do not affect my watching experience (3)
   d. Need improve, there are a lot of blur videos and affect my watching experience (2)
   e. Bad, the videos are not clear at all (1)

3. How will you rate the smoothness of the video you watched on **TikTok**
   a. Excellent, I did not experience any rebuffer (5)
   b. Good, the video plays smoothly except only a few short rebuffer (4)
   c. OK, the video stalls sometime, but it does not affect my watching experience (3)
   d. Need improve, the video stalls a lot, it affects my watching experience (2)
   e. Bad, the video has lots of stall and can hardly be played (1)

4. How will you rate the smoothness of the video you watched on **Dashlet**
   a. Excellent, I did not experience any rebuffer (5)
   b. Good, the video plays smoothly except only a few short rebuffer (4)
   c. OK, the video stalls sometime, but it does not affect my watching experience (3)
   d. Need improve, the video stalls a lot, it affects my watching experience (2)
   e. Bad, the video has lots of stall and can hardly be played (1)

Figure 27: Questionnaire for user survey.

C TikTok is conservative in video bitrate selection.

We show every bitrate that TikTok and Dashlet has selected in the section with Fig. 26. We can conclude from the figure that TikTok limits its bitrate even if the network throughput is high enough. This then leads to a significant negative impact on the QoE.

D Questionnaire sample.

We show the sample of the questionnaire we used in the user survey in Figure 27.

CellDAM: User-Space, Rootless Detection and Mitigation for 5G Data Plane

Zhaowei Tan, Jinghao Zhao, Boyan Ding, Songwu Lu
University of California, Los Angeles

Abstract

Despite all deployed security fences in 5G, attacks against its data plane are still feasible. A smart attacker can fabricate data packets or intelligently forge/drop/modify data-plane signaling messages between the 5G infrastructure and the device to inflict damage. In this work, we propose CellDAM, a new solution that is used at the device without any infrastructure upgrades or standard changes. CellDAM exploits the key finding that such data-plane attacks by the adversary would trigger unexpected data signaling operations. It thus detects all known and even currently unreported attacks via verifying data signaling correctness with novel state-dependent model checking. CellDAM could work with or without firmware access at the device using inference on low-level 5G signaling and configurations. It mitigates the damage upon detection by inducing frequency band switches at the device via the existing handover procedure. The prototype and empirical evaluation in our testbed confirm the viability of CellDAM.

1 Introduction

The current 5G and its legacy 4G cellular networks provide anywhere, anytime Internet access for billions of users. Security is an important design goal for 5G systems. Multiple security fences are thus deployed or enhanced [7], e.g., device authentication, enforced data encryption and integrity check. They aim to defend against recent attacks against the control plane [23, 41], as well as the data plane [47, 48, 60].

The current security fences are mostly *proactive protection* on data packets. However, this is insufficient for 5G data plane. Certain categories of attacks still cannot be protected. For example, a smart attacker can selectively drop a few data-plane signaling to incur cascading effect. Moreover, proactive protection is sometimes of high cost, such as protecting low-layer, cleartext, data signaling messages. Furthermore, proactive protection could be turned off with certain attacks [43], or unavailable to legacy devices in developing countries [1, 52].

In this paper, we explore a *reactive* solution approach called CellDAM towards 5G security. Instead of proactively preventing attacks, CellDAM complements the existing efforts by detecting whether a potential attack is underway and mitigating its damage. In addition to identifying the above-mentioned attacks that cannot be handled by proactive solutions, it offers two more benefits. First, the solution needs no standard change or hardware upgrade, thus being immediately deployable. Second, by verifying the correct operations, a reactive solution can find any attacks that do not follow the standard procedure, including both known and unreported ones.

A well-known challenge for data-plane detection is the high data throughput by 5G. It is thus considered impractical to inspect data packets at Gbps on a mobile device without consuming excessive processing or energy resources. CellDAM addresses the issue with a novel approach. We do not check each data packet directly. Instead, we inspect the data-plane signaling messages, which are standardized to facilitate data delivery but incur 1-2 orders of magnitude less overhead compared with monitoring all data packets. Our approach can detect attacks against both signaling messages and data packets. This is because every data delivery must exchange signaling messages for resource grant over the licensed 5G wireless channels. Therefore, *undesired data-plane signaling operations* might be triggered at the device by data-plane attacks.

We "verify what is right" when inspecting data-plane signaling. We thus model signaling operations for 5G data delivery; any operation that deviates from the standardized model is considered as a potential attack. It turns out that, a single-protocol, static checking scheme cannot detect all attacks. We thus devise a novel cross-layer, state-dependent model checking to validate data-plane signaling operations. At each state, we perform context-dependent validation to spot unexpected messages. Our experiments show that we have discovered three unreported attacks in addition to the known ones.

CellDAM is designed to work with various levels of privilege. Note that it requires access to signaling messages for inspection. The messages can be easily obtained with firmware access. However, for a user application, low-level 5G signaling cannot be accessed without root privilege. To overcome

this limitation, CellDAM utilizes SecHub, a separate companion node placed near the device. It uses a new inference technique to capture the signaling messages of interest from/to the protected device, but filter out all others.

Upon detection of an attack, CellDAM mitigates impact by triggering a 5G-standard handover procedure. This switches the device to a new cell. The attacker cannot track the device due to the encrypted handover messages and the dense deployment of 5G cells. Meanwhile, the handover procedure only incurs a disruption that lasts less than 100ms. With firmware access or network assistance, CellDAM could trigger the procedure using standardized commands. If they are unavailable, CellDAM leverages the SecHub to impact the channel measurement results to trigger a handover procedure on the victim device, but does not affect other devices.

We show that, CellDAM offers a practical solution to detecting and mitigating data-plane attacks with high accuracy. We prototype a device-side, user-level solution in C++ and Python based on open-source srsRAN [51]. SecHub implements components to infer the parameters, detect by verifying data-plane signaling, and mitigate with frequency band switching; all components do not require root privilege on the protected device. It achieves the detection accuracy of 0.989∼1.0, recall of 0.705∼1.0, and the F1 score of 0.823∼1.0. It can detect known and new attacks within 28ms on average. For mitigation, the handover can be triggered at 100% success rate with a latency of 1.85s on average. This procedure only incurs an average of 72.3ms disruption on applications. SecHub can successfully find the protected COTS devices with the accuracy of 97.2%∼100% under various traffic types.

2 Background

2.1 5G Primer

5G Architecture 5G system has 3 major components (Figure 1): User Equipment (UE), Base stations (gNB), and 5G Core network (5GC). The UE is a 5G user device. The gNB powers up the 5G network and provides wireless access in its coverage area for UE. It also forwards data to and from 5GC. 5GC includes the control plane for authentication and mobility support, and the data plane for data packet delivery.

5G Protocol Stack The 5G protocol stack consists of multiple protocols for both control-plane and data-plane functions. Non-access Stratum (NAS) and Radio Resource Control (RRC) are in charge of control-plane signaling. NAS facilitates control-plane signaling message exchange between the UE and 5GC. RRC carries the control messages and data-plane parameters for setup, power management, and handover behavior between the UE and the gNB. The data-plane enables IP packet delivery. We describe 5 protocols involved in data-plane operations: Service Data Adaption Protocol (SDAP), Packet Data Convergence (PDCP), Radio Link Control (RLC), Medium Access Control (MAC), and Physical Layer (PHY). SDAP manages the quality of service for data delivery. PDCP takes charge of encryption and integrity protection for control-plane and data-plane packet. RLC performs data concatenation and reorganization to ensure reliable, in-order data transfer. MAC controls radio access in licensed bands, and PHY performs wireless signal processing.

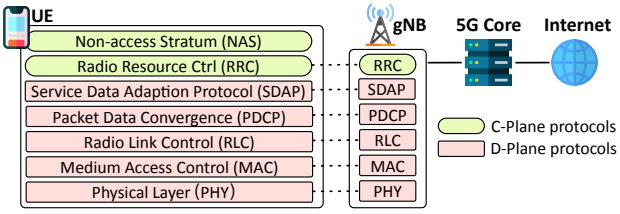


Figure 1: Architecture of 5G and its protocols.

Data-Plane Signaling In addition to RRC and NAS signaling for the control plane operations, 5G data-plane also has signaling between UE and gNB to facilitate packet delivery. The data-plane signaling exhibits in multiple forms. Various flags at the MAC layer (grant assignment DCI, scheduling request, HARQ acknowledgements, etc.), MAC control elements [3] that convey power control, time alignment or buffer status, and RLC control [5] for reliable transfer are all instances of data-plane signaling.

2.2 Protecting Data-Plane in 5G

Mutual Authentication Mutual authentication is a critical security measure in 5G, inherited from 4G with little change. The UE and network perform a secure Authentication and Key Agreement (AKA) procedure during connection for authentication and session key set-up. The session key is used to generate keystream for every packet with several parameters such as sequence number.

Protection on Data Packets The keystream is generated to encrypt data packets without key reuse for both control-plane and data-plane packets [7]. The sender also updates another keystream to generate an integrity code attached to the message for integrity check at the receiver. While integrity protection for control messages is enforced in 4G, it was optional for data plane due to high overhead. The vulnerability allows attacks that fabricate data packets [48, 49]. 5G aims to enforce integrity protection on all data packets. Although its usage is still optional, both 5G UE and gNB should support integrity protection at the full speed starting from release 16 [7]. With the increasing capacity of the hardware, it is expected such integrity protection will be mandatory.

3 A Case for Detection and Mitigation for 5G Data-Plane Attacks

3.1 Threat Model

In this paper, we consider an adversary who seeks to incur various damages on the target victim 5G device on its *data*

plane. The attacker has the following capabilities: 1) Connect to the same cell as the victim device, which is feasible with fingerprinting [32] or social engineering [36]; 2) Eavesdrop on, transmit data, or send noises on physical channels; 3) The adversary may exploit fake base station (FBS) [48, 49] or overshadowing attacks [60]. Although FBS is mitigated in 5G [4], it is still possible to launch certain variants of FBS, such as relay FBS as man-in-the-middle. We do not limit the message that can be forged by the adversary, which could be user packets or signaling messages.

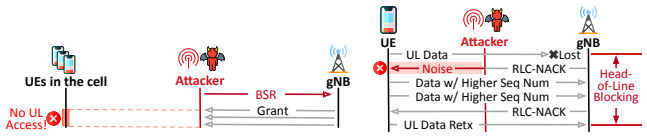
We do *not* consider attacks that threaten control-plane, such as an IMSI catcher. We also do *not* consider an insider attack, where the adversary can steal security keys stored in SIM/networks or even compromise a 5G base station. Protecting such attacks is beyond the scope of this work.

3.2 Proactive Protection is Insufficient

State-of-the-art: Per-message proactive protection The authenticity of 5G user packets is protected by end-to-end (such as TLS [29]) and 5G-specific integrity protection (as introduced in §2.2). However, data-plane attacks are still possible despite the protection.

Issue 1: Not every message could be protected at low cost Unlike end-to-end packets, the small-sized data-plane signaling messages are not protected by proactive solutions [2], due to high overhead or impracticability being in the below-PDCP layer. Fabricating these messages can incur serious damages as shown in recent studies [54]. An attacker only needs to forge a few messages to incur damages consistently. One example attack is shown in Figure 2(a). An attacker forges a Buffer Status Report (BSR) that requests uplink grant from the gNB. The uplink grant in a time period will be assigned in accordance with the request. A malicious BSR could contain a large request, wasting licensed resource and blocking uplink delivery of any UE in the cell for hundreds of milliseconds. The attacker is capable of repeatedly forging such messages, which can continuously drain the resource. We note that, gNB cannot distinguish this attack message from a legitimate one.

Issue 2: Certain attacks cannot be proactively protected The attacker can also intelligently corrupt/drop certain messages to incur serious damages. Such attacks *cannot* be protected by any integrity check. One example attack is shown in Figure 2(b). In this attack, the adversary corrupts an RLC control message NACK, which is used to request retransmission for certain packets. When this NACK is lost, the retransmission is delayed. Due to the RLC mechanism, all subsequent data packets will be blocked. The effect will last until the next RLC message, which could be hundreds of milliseconds based on common RLC configurations. Moreover, the attacker can repeatedly send the message to cause persistent damages. The attacker does not require fake base stations or channel jamming. Instead, a lightweight attacker only needs to send a single signaling every hundreds of milliseconds.



(a) Fabricate data-plane signaling. (b) Selective corruption.

Figure 2: Example data-plane attacks that cannot be protected with current proactive approaches.

Issue 3: Proactive protection could be turned off Forging user data packets is still possible despite the data packet integrity check in 5G. The usage of integrity protection is still negotiation-based [7]. Hence, the attacker can disable integrity protection by leveraging certain vulnerabilities, such as those from firmware [43]. Furthermore, legacy 4G devices or 5G devices on earlier versions, which are still a considerable number [42, 52, 56], do not implement mandatory integrity check [1]. For these devices, known attacks can incur serious damage when end-to-end protection is not used. An attacker can manipulate DNS requests to a malicious Web server [48]. The malicious server could then send any forged content to the victim. Beyond Web, the attacker could forge arbitrary encrypted data through a man-in-the-middle [49].

Insight: Reactive detection and mitigation can complement proactive protection Given all the existing threats which cannot be protected by proactive approaches, we shall also develop proper *reactive* solutions to complement the proactive protection. They should include both detection and mitigation. The detection methods will spot any suspicious activities, while the mitigation will help the victim recover from the damages. They can complement existing solutions, while not requiring any 3GPP framework change.

4 Overview of `CellDAM`

In this section, we present the design goal and challenges of designing a reactive protection.

4.1 Security Goals

Following our insights, we seek to *detect and mitigate* 5G data-plane attacks *without standard changes*. As discussed in §3.2, this solution approach offers an immediate remedy for certain attacks that are not protected by current proactive approaches. We argue that, such detection is essential on the device side. First, it is more scalable compared to network-centric solutions. Second, some attacks are only detectable on the device side. Take the signaling attack (Issue 1) in §3.2 as an example. In this attack, the UE will receive uplink grant that it did not request. However, the network cannot distinguish whether the request is malicious or legitimate. Therefore, device-side detection is required to detect and mitigate the attacks.

Meanwhile, our reactive detection and mitigation scheme should achieve the following goals:

Verify what is right for attack detection We design and implement approaches that verify whether the runtime, data-plane operations follow the correct procedures stipulated by the standards, and treat any undesired behaviors, rather than specific attacks, as suspicious. This ensures the detection of a category of attacks that yield improper data-plane operations. Even if an attack has not been reported yet, it can be detected by our approach as long as it triggers undesired behavior. We admit that certain attacks that adhere to the standardized approach might not be detected by our approach. Our solution prioritizes soundness over completeness.

Detection and mitigation need to be practical We aim to design a reactive method without hardware update or extra privilege. It can thus benefit legacy devices.

•***No infrastructure upgrade*** We will design our solution *on the device side*. Unlike network-centric protection approaches [24, 44, 46, 57], our device-side scheme needs no expensive infrastructure or hardware upgrades. Moreover, as we discussed earlier, certain attacks can only be observed by the device.

•***1-2 orders of magnitude lower overhead compared to verifying each data packet*** It is nontrivial to monitor the 5G data plane. Data throughput in 5G is expected to reach a few Gbps. The traffic volume of data packets is several orders of magnitude higher than the control-plane signaling messages. We aim to design a lightweight security solution that can work under heavy data traffic. The overhead should be 1-2 orders of magnitude smaller compared to monitoring the entire traffic.

Applicable to different defense models The solution should work given different levels of privilege. We consider three defense models in this work: (1) Defense with firmware access; (2) A user-space application that can communicate with the operator; and (3) A user-space application without any privilege. All three models have their own usage scenarios. For (1), a device vendor implements the solution to enhance the security of its 5G devices. For (2), we consider an operator who tries to protect its users with additional security requirements. However, the operator cannot directly access the firmware. For (3), we consider a normal user who tries to protect the device. All three models complement each other under different scenarios to protect 5G data plane.

For the Defense model (1), the solution could be implemented inside the firmware, as it has access to all cellular-specific info and OS-level privilege. For both (2) and (3), mobile OS only provides control-plane basic info [10], such as connection state. The application cannot access device-side cellular-specific info unless extra privilege (e.g., Diag port) is granted for tools like MobileInsight [37] or QXDM [45]. However, such extra privilege (e.g., root access) exposes new vulnerabilities [15, 21, 22, 50] and is unavailable to most devices due to technical or legal concerns [34]. Our solution aims to function despite the limitations.

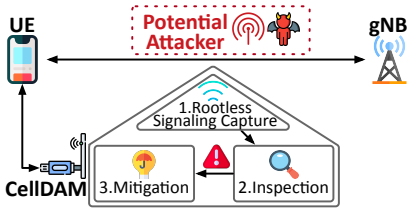


Figure 3: Envisioned procedure of `CellDAM`.

State-of-the-Art: Existing *reactive* approaches cannot achieve these goals To the best of our knowledge, all previous works on 5G/4G attack detection focus on undesired behavior on the control plane [26, 27]. This is insufficient, as recent studies [54, 60] show that an advanced adversary can bypass control plane and directly attack the data plane.

4.2 Solution Overview

We design `CellDAM`, a 5G data-plane inspection scheme without root privilege, as illustrated in Figure 3. It satisfies all design goals. `CellDAM` first captures all data-plane signaling messages from/to the protected UE. It could do so in a separated node called `SecHub` and bypass the requirement of in-device extra privilege. `CellDAM` then inspects the *lightweight* data-plane signaling messages to spot undesired behavior. Finally, `CellDAM` uses `SecHub` and in-device operation to mitigate the attack damage via handover-based cell switching.

Inspecting lightweight signaling messages with state-dependent checking (§5) We first show that, inspecting data-plane signaling offers an effective way to detect data-plane attacks, while it has magnitude lower overhead due to its low volume. We further propose a *cross-layer, state-dependent* model checking for attack detection. If the device spots an incoming signaling message that is undesired in the current state, it detects a potential attack.

Rootless signaling inference (§6) Signaling verification requires access to the messages. For the defense model with firmware access, such privilege is granted. To serve the majority of rootless devices, `CellDAM` incorporates a separate, companion node named `SecHub` for the protected device. The goal is to share a consistent view of the physical channels. The node can continuously capture over-the-air signaling messages by inferring the device ID and traffic pattern. It needs no extra privilege on the protected device.

Device-side reaction without infrastructure upgrade (§7) Once a suspicious operation is detected, we activate the mitigation module that switches to other, potentially attack-free 5G channels. This could be done by leveraging the standardized 5G handover procedure and dense cell deployment. Handover incurs small disruptions in the applications, while being resilient to attacks. `CellDAM` can initiate such a procedure via two approaches. It can directly create a standard-compliant message for handover. It can also leverage `SecHub` to affect the device to trigger a handover, without affecting other user devices.

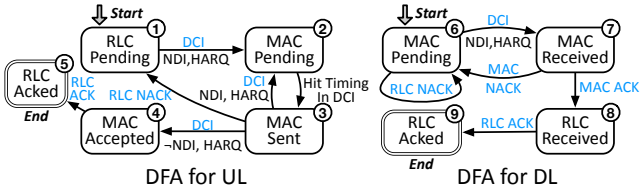


Figure 4: DFA for tracking states with data-plane signaling. See Appendix C for an extended version.

5 Cross-layer, State-dependent Checking on Data Plane Signaling

We now present how CellDAM inspects the data-plane signaling messages to detect undesired behavior. We follow our guideline of "verifying what is right" to model the device-side protocols and detect any undesired behavior. To this end, we devise validation schemes using the standardized data-plane delivery procedure to check each incoming/outgoing data-plane signaling message.

The solution has two highlighted components. First, instead of verifying data packets of large throughput, CellDAM inspects lightweight, yet critical data-plane signaling messages for attack detection. Second, we present a state-dependent model checking method to find suspicious behaviors.

5.1 Monitor and Inspect Data Plane Signaling

A straightforward solution can verify each data packet. This can be done by applying Deep Packet Inspection (DPI) or detecting anomalies in the wireless signal for each data packet. However, they cannot detect all attacks, such as the attacks that target signaling messages. Meanwhile, this will incur huge overhead, especially given the large 5G throughput.

Why inspect data-plane signaling We show that, monitoring data-plane signaling offers an effective alternative method for detecting data-plane attacks, including attacks on both data-plane signaling and data packets. First, delivering each data has a standardized sequence of signaling messages due to 5G infrastructure-controlled data access. For any type of attack, the attacker needs to tamper with the signaling messages. Therefore, this solution approach will cover a wide range of attacks. Second, the frequency of data-plane signaling is smaller than data packets due to 5G aggregation scheduling.

Data-plane attacks might manipulate data-plane signaling in standardized data delivery For data delivery, each device must follow a standardized approach, as 5G uses gNB to mandate the radio resources. Therefore, we can use the signaling messages to model the state of each packet delivery. To model and track the state, we use Deterministic Finite Automata (DFA), a common technique for state tracking in attack detection [19, 26]. We study 3GPP standards across all 5G-specific data-plane sublayers and manually create DFA based on *mandated, standardized data delivery procedure*. We form cross-layer DFA for each RLC data packet with its necessary state transition at the MAC layer. We do not include PDCP, as it does not maintain state or buffer packets.

The constructed DFA is shown in Figure 4. For uplink, data transmission follows a scheduling-based feedback loop. The device first sends requests (Buffer Status Reports or BSR) to ask for resource grants until the packet is delivered by MAC. The MAC fast retransmission is notified by a new DCI with the same HARQ ID and NDI. The packet is considered delivered when the RLC ACK is received. For downlink, the data transmission follows the same procedure but without the request-grant loop, as the gNB initiates the transmission. The device sends the MAC feedback of ACK/NACK in PUCCH.

Inspecting data-plane signaling is lightweight Compared with data-plane packets, inspecting data-plane signaling is of much lower overhead. First, the size of each signaling message is much smaller than the actual data packet. The signaling messages (such as DCI, RLC Control) are at most several bytes long. Some PHY messages are merely 1-bit indicators. Second, 5G data delivery will transmit multiple IP data packets in a single data block (aggregated and segmented by the 5G RLC protocol). Therefore, for signaling messages that facilitate data delivery (e.g., DCI), only one such message is needed for the large block.

We validate the hypothesis that control traffic is significantly lower than data traffic by comparing their traffic volume. We show results from operational traces in a commercial network and in our SDR-based testbed. Our testbed runs standard-compliant srsRAN 5G [51] and the details will be shown in §10. Since our testbed does not support features such as MIMO or carrier aggregation for higher throughput, we also collect traces from commercial operators. As the current open-source 5G decoding tools (e.g., MobileInsight 5.0 [37, 38]) have not supported 5G data plane, we collect and analyze 4G data plane as a reference, considering that data-plane signaling design remains largely unchanged in the current 5G NSA [3,5][1]. As shown in Figure 5, the processing of data-plane signaling is 1∼2 orders of magnitude lower than that of data packets.

Therefore, detecting attacks with data-plane signaling is of much smaller overhead than monitoring the entire data-plane packets. Prior work [20] has already shown an SDR-based system is capable of monitoring DCI messages for *all devices in a cell*. It is thus feasible for CellDAM to capture all data-plane signaling *for a single device* while performing inspections in real-time (detailed in §9). This is important considering 5G's high data rate.

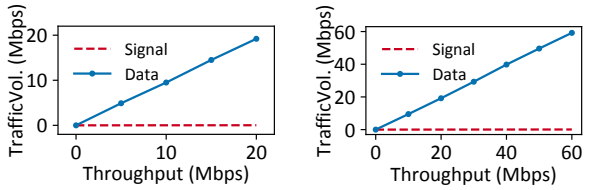
5.2 Cross-layer, State-Dependent Checking

Stateless checking cannot detect certain attacks The detector could perform certain checks within each protocol

[1] One major difference is that 5G cancels PHICH which is used for uplink data retransmission. Therefore, we do not include it for calculation.

Check	State	Message	Validation Details
c_1	All	Any Message	The data-plane signaling shall be in the accepted list for each state. (Appendix C)
c_2	s_3, s_6	RLC NACK	It shall not be received after RLC timer and MAC retransmission timer expire or after receiving an RLC ACK with higher sequence number.
c_3	s_4, s_8	RLC ACK	An RLC ACK shall not be received before the packet is acknowledged in MAC.
c_4	s_7	MAC ACK/NACK	The ACK/NACK in PUCCH should be delivered at correct timing after DCI; If not, this is an indicator that the previous grant/data is received during DRX OFF.
c_5	s_1	DCI for UL Grant	There should not be large "free" grant when no request is sent or no data in buffer.

Table 1: Validation checks performed by CellDAM based on state and message.



(a) Under Testbed (5G) (b) Under Commercial (LTE)

Figure 5: The comparison of traffic volume per second for data packets vs. data-plane signaling messages.

regarding whether the incoming signaling message conforms to the standards. However, this solution will fail to spot certain attacks against 5G data plane.

We showcase a concrete example. When an RLC signaling message ACK is received, it is possible if MAC layer has accepted this message. Otherwise, this message is impossible and a potential attack is detected. Therefore, the checking must rely on the current state across different protocols.

Cross-layer, state-dependent model checking with DFA Therefore, we propose a state-dependent checking for 5G data plane security. Instead of proposing a few checks statically in each protocol, we leverage the cross-layer DFAs we built for data delivery, whose inputs for transition are *data-plane signaling messages* or their derived events. If the next captured signaling message m passes all validation checks, the DFA moves to a new state; Otherwise, a potential attack is detected and we initiate the mitigation procedure.

Formally, we maintain n deterministic finite state machines $M = \{M_1, M_2, ..., M_n\}$. For each DFA $M_i, i = 1, 2, ..., n$, we denote it as a 5-tuple $(S_i, S_i^0, S_i^1, \Sigma, T_i)$, where S_i is a finite set of states with $S_i^0 \in S_i$ being the initial state and $S_i^1 \subseteq S_i$ being the accepted states, Σ is a finite set of input messages[2], $T_i : S_i \times \Sigma \rightarrow S_i$ is a transition function mapping the pairs of a state and a received message to the next state.

We build validation checks $V = \{V_1, V_2, ..., V_k\}$. Each V_i is associated with a DFA M_j, a state $S \in S_j$, and a message $m \in \Sigma_j$. Every time the DFA M_j with state S inspects a new message m, CellDAM runs the corresponding check(s). They map the signaling message m to 0 (fail) or 1 (succeed) given the current context, which is derived from past records or other DFAs. A potential attack is identified if one of the validation checks fails; Otherwise, M_j accepts message m and updates its state accordingly.

[2]To make the problem tractable, we only consider the discussed data-plane signaling messages. We prioritize soundness over completeness.

Based on the state, we perform a few validations on each incoming/outgoing signaling message. We show the list of validations in Table 1. First, all states will have a list of accepted messages. CellDAM checks whether the next message is in the list. The detailed list for this check is shown in Appendix C. For c_2 to c_3, we are checking whether the RLC operations are consistent with the MAC layer for both uplink and downlink. For example, upon receiving RLC NACK, we check whether an early RLC ACK that has already acknowledged a packet is received. For c_4, we use the indicator of no ACK/NACK to detect a possible forged message received in DRX OFF. For c_5, CellDAM detects abnormal grants from gNB without any request. Note that a gNB can freely grant the device with small grants. However, they are usually 100-200 Bytes long. Any larger grant incurs a waste of resources. Therefore, it could be the outcome of a forged BSR or grant signal. If all checks pass, the DFAs are switched to the new state.

6 Access Signaling Messages for Detection

With CellDAM's checking techniques, an end device can inspect signaling messages for attack detection. We now discuss how to access them under three defense models in §4.

With direct firmware access For defense solution that has firmware access, it could directly capture the signaling messages. In fact, these messages are already processed by firmware to realize the functionalities.

No direct firmware access If CellDAM is deployed on the application layer, it cannot directly access the signaling messages. Although messages are available in tools such as MobileInsight [37] or QXDM [45], using these tools and accessing the low-level messages require root, which exposes new vulnerabilities [15, 21, 22, 50] and is unavailable to most users [31]. Even the application is allowed to communicate with the network (i.e., defense model 2), it cannot access the *device-side* signaling for detection.

Idea: Infer signaling messages without root privilege in a separate node To address the concern, we design and deploy a separate gadget (e.g., extended hardware with wireless capability), SecHub. The node is placed close to the protected device within a meter; it passively receives and decodes the data-plane signaling over the air. It can also communicate with the user-space security manager application at the protected device. The device and SecHub connect each other,

either with wire (a gadget that is attached to the device via USB) or wirelessly (i.e., Bluetooth).

However, it is not trivial to infer such info in the user space with SecHub. We address this issue in the next section.

Challenge: Unknown configurations Although 5G data-plane signaling messages are not encrypted, SecHub must identify which ones belong to the protected devices. Several configurations are needed: (1) The carrier frequencies; (2) The physical cell ID (PCI) that indicates the physical-layer identity of the cell; (3) The Cell Radio Network Temporary Identifier (C-RNTI) that distinguishes the target device from other devices connected to the same cell.

Directly infer the parameters will not work Unfortunately, not all these configurations could be acquired from the protected device without root access. Android provides APIs for applications to obtain the current band and PCI [10]. In contrast, C-RNTI can only be extracted from the victim device with root privilege. Without C-RNTI, SecHub cannot recognize which traffic is for the protected device. Therefore, we need a solution that can recognize which configurations are assigned to the protected device over the air.

Idea: Use high-layer traffic pattern that is visible to SecHub Since the user space has no access to lower-layer C-RNTI, we must identify the configurations with higher layer features. We note that, data traffic pattern is *visible to both device and SecHub*. Therefore, it is an ideal "channel" to insert fingerprint and notify the identity to SecHub.

Henceforth, we generate specific traffic patterns on the target device, with SecHub being aware of the pre-agreed pattern in advance. It can thus observe the channel and identify the target device's C-RNTI by analyzing low-layer signaling. Our approach takes three concrete steps.

Step 1: Traffic Pattern Coordination First, SecHub randomly generates a traffic interval and sends it to the target device through the wired or wireless channel. This interval is used as the fingerprint for the target device. The traffic interval triggers a unique pattern for the data-plane signalings for fast inference. SecHub leverages this shared traffic pattern to recognize the target device in later analysis.

Step 2: Trace Collection Second, the device creates traffic (e.g., small UDP packets) with the acquired interval. The traffic generation is performed by the application and does not require root privilege. gNB will assign grants for the device to deliver data. At the same time, SecHub monitors all the subcarriers in the target cell and tries to decode the C-RNTI from all grants with all possible positions in the band. This is necessary as grants do not always locate on the same subcarrier in the band (for reducing inter-cell interference). SecHub records the decoded C-RNTIs with corresponding time slots for inference.

Step 3: C-RNTI Inference Finally, SecHub aggregates the grants for each C-RNTI decoded from the collected trace. SecHub first ranks the C-RNTIs by grant numbers and filters the top 10% C-RNTIs as the candidates. The time intervals between consecutive grants are calculated and compared with the negotiated interval for all the candidates. The grants for the fingerprinting traffic will show the same interval. Although there may be background traffic from the target device, the grants triggered by the fingerprinting traffic still show the periodic pattern and could be filtered from the device's grant traces. By ranking the ratio of matched intervals with the total interval number, the top C-RNTI will be selected as the target's C-RNTI. To ensure robustness, SecHub performs the procedures twice and checks if the inferred C-RNTI values match. If the candidates do not match, SecHub will perform the inference again until a candidate is selected.

Combining with the frequency and PCI of the device from OS API and the C-RNTI inferred from the collaborated traffic fingerprinting, the SecHub could successfully camp on the cell and capture the downlink/uplink messages. The entire procedure does not require root access at the device.

Tracking configuration change We also note that, the C-RNTI configuration could be updated within an encrypted message in both static and mobility scenarios. CellDAM incorporates a new solution to prevent such change and enable continuous tracking. We detail this solution in Appendix D.

7 Device-Centric Mitigation

Although 5G standard updates could fundamentally defend against forgery attacks, they require months or even years to be deployed in practice. To this end, we design device-centric mitigation to provide a quick remedy for existing devices. We leverage the existing, dense 5G cell deployment to help the victim dodge the attacker.

7.1 "Quick Dodge" with Handover

Band switching to avoid the attacker on a specific cell We observe that, the attacker must camp on the cell that serves the victim device and forge messages in the current band to launch the attack, regardless of the attack methodology (§3.2). Therefore, the victim could quickly escape from attackers by switching to another frequency band (i.e., a different 5G cell).

With the insight, CellDAM thus aims to switch the band (i.e., cell) that serves the victim device to avoid the attacker.

Use handover procedure to trigger band switching In CellDAM, we leverage the 5G handover procedure to realize band switching. In 5G, a UE measures the signal quality by metrics of Reference Signals Received Power (RSRP) and Reference Signal Received Quality (RSRQ). When the experienced signal quality of the serving cell is worse than the thresholds configured by gNB, the UE sends reports to gNB, which makes the handover decision and sends a handover command to the device.

Although rare, it is possible that there are no other cells available. In such cases, CellDAM could opt to generate a warning instead of triggering the handover.

Why is band switching via handover effective against attackers? First, the attacker could not control or know which cell the device is switching to, as the handover command is encrypted. Given that, the attacker needs to enumerate all nearby cells and use fingerprinting on each to find whether the device is in the cell. It takes prolonged time and effort for an attacker. This could take minutes, before which the device might already move to a new cell.

Application is resilient to handover Handover incurs little overhead on applications. UE does not go through the slow cell search or connection setup procedure. Instead, it only has to update its PHY parameters for the new cell. Therefore, the disruption to the applications is minimum.

How to trigger a handover? To initiate a handover procedure, the most straightforward way is by requesting the gNB to send a handover command to the device. This is possible if CellDAM can communicate with the operator (Defense model 2). On the other hand, the device can send a 5G measurement report to the base station. It indicates that the measurement result from another cell is better than the current one by an offset configured by gNB (i.e., a measurement event in 5G). This will subsequently trigger a handover procedure. Defense model 1 could take this approach. However, for Defense model 3, neither approach is available. In the next section, we describe how it could still initiate a handover.

7.2 Trigger Handover with SecHub

When the handover-related messages cannot be directly created, CellDAM takes a different path by affecting the measurement result. If the measured RSRQ on the serving band is bad, a legitimate report will be triggered by the device. Although either bad RSRQ or RSRP can result in handover, we focus on RSRQ, because RSRP is measured based on the reference signal power and is hard to be affected by SecHub.

Solution: Precise reference signal downgrade We design adaptive signal degradation to ensure low-overhead band switching. Instead of the entire channel, SecHub only copes with the reference signal in 5G. The device measures the reference signal to monitor the signal quality regardless of PHY techniques (e.g., MIMO, carrier aggregation, dual connectivity, etc). The reference signal only exists in specific subcarriers and time slots. SecHub calculates the positions of the reference signal based on the current physical band according to the 5G standards [6]. By morphing the reference signals only, SecHub downgrades the victim's measurements of the current frequency band without much overhead.

The solution should not affect other devices. SecHub adaptively controls its power upon triggering the handover. More details on the power control are shown in Appendix B.

8 Security Analysis

8.1 Attacks Covered by CellDAM

In this section, we discuss what attacks can be detected by CellDAM. With our design, CellDAM can detect multiple attacks that target both data plane packets and signaling message, as shown in Table 2. The details for each attack and how CellDAM detects it are elaborated in Appendix A.

Attacks against data packets We consider three attack actions that target data packets: injection, deletion, and manipulation. For injection, the attacker attempts to insert a new data packet. For deletion, the attacker tries to remove a packet from being received by the device or the network. For manipulation, the attacker seeks to change certain bits in a data packet. Any bit in the IP packet (application, transport, or IP headers) could be changed.

We first show that, both injection and manipulation attacks can be detected by CellDAM. We note that, neither attack can be directly launched over the air. Flipping data bits over the air will fail the CRC check, while directly injecting a new packet will fail to be decrypted due to mandatory encryption. Therefore, the possible methodology for injection or manipulation is the Man-in-the-Middle (MitM) approach. The adversary intercepts the packet, flips the bits, re-encodes it, and injects the altered packet.

This could be detected by CellDAM. as a MitM will incur undesired signaling messages. There are two possible ways to launch MitM. For the scheme using relay FBS (A1 in Table 2), an attacker cannot directly learn the critical data-plane configurations, which are transmitted over the *encrypted* RRC messages [60]. Consequently, the forgery could be sent in an impossible context, e.g., in the time slots when the device is in its Discontinuous Reception (DRX) OFF state (i.e., sleep mode). This behavior will be detected by CellDAM. It applies to both uplink and downlink forgery, as the attacker needs to send DCI to the victim device for notification of both uplink and downlink scheduling. For the attack that corrupts the transmission and injects retransmission (A2 in Table 2), it needs to forge DCI and data packets. The next DCI from the attacker could reach the device before the acknowledgment of the forged data, thus incurring an undesired behavior.

Unlike manipulation and injection, deleting data packets cannot be detected by CellDAM. The attacker can send noises and corrupt the data packets. CellDAM cannot distinguish it from a corruption caused by environmental noises. There is no readily available scheme to defend it without changing the 5G PHY; changing PHY is beyond the scope of this work.

Attacks against data plane signaling We show that, all the detection, manipulation, and deletion of signaling messages can be detected by CellDAM. This is relatively straightforward compared with data packet forgery. Since the forged or missing signaling is not anticipated, some messages will be received in wrong or unexpected context. Three examples

#	Attack	Target Message	New?	CellDAM	Undesired Behavior	Check
A1	DL Data Manipulation w/ Relay FBS	Data packet	Adapted from [48, 49]	✓	The forged packet received during DRX OFF.	c_4
A2	DL Data Forgery w/ Retransmission	Data packet	Inspired by [54, 60]	✓	Forged DCI for forged data received in wrong context.	c_1
A3	Packet Delivery Blocking	RLC Control NACK	Adapted from [54]	✓	The RLC Control is not received when the timer expires.	c_1, c_2
A4	Prolonged Packet Delivery	DRX Command	Adapted from [54]	✓	Message received with pending transmission; DCI during DRX OFF.	c_1, c_4
A5	Radio Resource Draining	Buffer Status Report	Adapted from [54]	✓	Grant is received when no data is pending.	c_5
A6	Break Reliable Transfer	RLC Control ACK	Yes	✓	RLC packet is NACKed after being already ACKed.	c_3
A7	Data Collision	DCI UL Grant	Yes	✓	Data sent in unauthorized resource blocks will not be acknowledged.	c_5
A8	Delayed Transfer	MAC ACK	Yes	✓	Sender MAC falsely regards ACK and triggers RLC retransmission.	c_1

Table 2: List of known (A1–A5) and unreported (A6–A8) data-plane attacks and how they trigger undesired messages. See Appendix A for details of each attack.

are listed in Table 2 (A3-A5). They are adapted from those reported in 4G or Cellular IoT and include all three types of attack. Each attack violates a certain context in packet delivery and fails to pass all checks. For A3, the attack corrupts an RLC NACK signaling. This incurs head-of-line blocking, stopping the delivery for more than 100ms. The attack can be detected by the device, as the device observes no RLC NACK after it requests one in the uplink packet. For A4, the attacker injects a DRX command signaling message, which forces the device into DRX OFF even in the presence of new data. The device thus receives a DCI during DRX OFF, detected by its lack of ACK/NACK. For A5, the device manipulates the amount in the BSR request to drain the wireless resource and block access. The device will observe unsolicited grant without pending UL data.

Unreported attacks CellDAM also detects unreported data-plane attacks, since it *verifies what is right* and detects any potential attack that breaks the delivery procedure. For each DFA state, signaling message, and validation, we check if a forged message that fails the validation can be from the adversary to incur damages. Consequently, we illustrate three unreported attacks and how CellDAM can detect them in Table 2 (A6-A8) with details in Appendix A.

8.2 Attacks against CellDAM

We next consider an attacker who is aware of the existence of CellDAM and tries to break it under our threat model. We specifically focus on the security of SecHub. This is because, if the inference or mitigation is implemented within the firmware, it is considered difficult to break it without an insider attacker. This is beyond the scope of our threat model.

Attacker attempts to break CellDAM's inference We first consider an attacker who tricks SecHub and prevents CellDAM from recognizing the traffic from the protected device. With CellDAM design, this is not possible. The traffic is generated from the in-device application with the pre-defined pattern. The application is also secure as it needs no root or other privileges. Therefore, the pattern is unknown to the attacker, who cannot insert malicious packet to break the inference. In addition, CellDAM is stable by repeating inference three times to confirm the target, avoiding a malicious attacker to inject noises and break the inference.

Attacker compromises or breaks SecHub The attacker can either gain access to SecHub, or break the communication between SecHub and the device. However, neither is possible. First, SecHub is available to end users without exposing any unnecessary interfaces, such as Internet access or wireless control. A user or an application will be unable to add/change/delete the SecHub software. SecHub is solely used for CellDAM detection and mitigation purposes. A passcode is set to access its functionalities. Second, SecHub communicates securely with the protected device. The user installs an application in the protected device using the certificate that comes with the SecHub. The device thus mutually authenticates with SecHub and encrypts all traffic between them. Only nonsensitive information is exchanged over this channel.

Attacker leverages SecHub to force handover The attacker could force the device to switch band by deteriorating the signal strength sensed by SecHub. However, such an attack is very limited in its impact. First, the device has immediate data access after moving to a new cell. The handover disruption is short. Meanwhile, the attacker cannot control which new cell the device moves to. For this purpose, the attacker needs to launch a time-consuming identification procedure on all local cells. The attacker cannot control which cell/base station the victim switches to, either. Instead, the band switching in 5G will prioritize the cells from the same (legitimate) base station. Forcing the device to an illegitimate base station is thus unlikely. Therefore, the attacker gains little from forcing

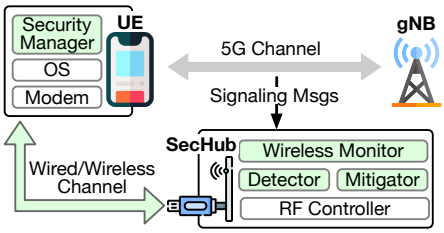


Figure 6: Implementation of CellDAM. Green blocks are CellDAM modules.

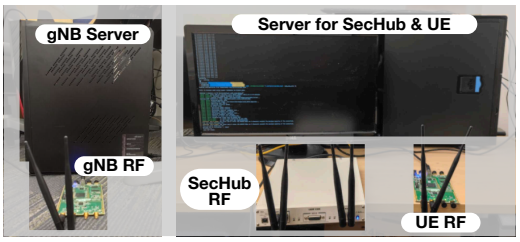


Figure 7: Testbed Setup for CellDAM.

a handover with SecHub. Defending against it is out of the scope of CellDAM.

9 Implementation

We implement CellDAM as shown in Figure 6. We implement the defense model with the least privilege (Defence model 3). Based on our discussion, we implement SecHub to perform attack detection and mitigation. A security manager app on UE facilitates SecHub for rootless signaling capture. We next elaborate on each component. It could be adapted to the other two defense models with little change. For detection, our detector module could be directly used by the other two models. For mitigation, they only require to trigger one extra command after detection.

Wireless Monitor A wireless monitor is deployed based on srsRAN [51] to perform the rootless signaling capture through the RF controller with the SDR devices. We implement it with 2,794 lines of C++ code. The monitor collaborates with the UE to camp on the target cell, infer the UE's C-RNTI, and simultaneously capture real-time messages on both the uplink and downlink channels. After the capture, it decodes the signaling messages accordingly and passes them to the attack detector.

Detector with state-dependent model checking We implement the state-dependent model checking attack detector with 1,252 lines of Python. The real-time traces from the wireless monitor will be fed into the detector for undesired behavior and potential attacks. If any consecutive signaling messages violate the cross-layer model checking, potential attacks will be reported by the detector. The detector will further notify the mitigator module to trigger the mitigation.

Mitigator We deploy the mitigator with 860 lines of Python code. After detecting any attacks, the mitigator triggers the victim handover. With the current signal conditions acquired from the security manager application from the victim, the mitigator calculates the minimum transmit power to trigger the UE handover with the SDR devices. It then notifies the RF controller to initiate signals targeting the victim UE. This will trigger handover to escape from the attacker's cell.

Security Manager App On the phone side, we deploy a security manager application with 1,264 lines of Java. The application monitors the current band, PCI, and signal conditions (RSRP and RSRQ) with the Android Telephony API [10] to facilitate the rootless signaling capture. It also generates a corresponding UDP traffic after receiving the traffic interval from SecHub for the collaborated traffic fingerprinting. It supports exchanging the data with SecHub through a wired (USB) or wireless (Bluetooth in the current implementation) connection leveraging Android APIs [9]. It supports the X.509 certificate to facilitate the mutual authentication and encryption between the app and SecHub. We also implement an equivalent application for srsUE running on user-space. The same set of information is extracted from the srsUE by hooking the current srsUE functions. Then the information is shared with the SecHub with the socket API.

10 Evaluation

10.1 Evaluation Setup

Testbed Setup We construct a testbed for experimental validation, as shown in Figure 7. The gNB and UE are built upon srsRAN [51] 5G protocols. The physical layer encoding is still kept with 4G due to current hardware limitations. CellDAM does not rely on any 5G-specific PHY feature. The gNB software is run on an i7-9700K PC with Ubuntu 20.04. The UE runs on an Intel Xeon Silver 4214 server running Ubuntu 20.04. Both use USRP B210 as their RF frontend, with the frequency set to an unlicensed 2.4GHz ISM band. SecHub is co-located on the same server as the UE and uses USRP X300 as the RF frontend. The security manager application runs on the same server as a user-space process and shares the information extracted from srsUE with SecHub. The mobile version of security manager application is tested on a Pixel 4a with Snapdragon 730 running Android 12.

Attack Reproduction All attacks listed in Table 2 are recreated within the testbed in order to evaluate CellDAM's ability of attack detection and mitigation. We realize attacks on both data packets and signaling. We simplify the attacks with partial software emulation on our testbed for controllability and reproducibility In attacks with relay FBS, we set up both the FBS and the real gNB in the testbed. We emulate the radio link between relay FBS and real gNB in software with ZeroMQ [61] to avoid interference with the link between UE and relay FBS, which uses physical link with USRP.

On the other hand, the attacks without relay FBS rely on manipulation of the underlying data channel. We emulate the attacker using a separate thread within the gNB and UE program, which has access to the transmitting and receiving signal buffer. Eavesdropping is realized with inspection of the receiving buffer, while the forging or corruption attacks are

Attack	A1	A2	A3	A4	A5	A6	A7	A8
Precision	0.989	1	0.999	1	0.996	1	0.996	0.999
Recall	0.705	0.976	1	0.989	1	1	1	1
F1	0.823	0.988	0.999	0.994	0.998	1	0.998	0.999

Table 3: Effectiveness of attack detection with CellDAM.

emulated by injecting encoded attack messages or noise to the transmitting buffer.

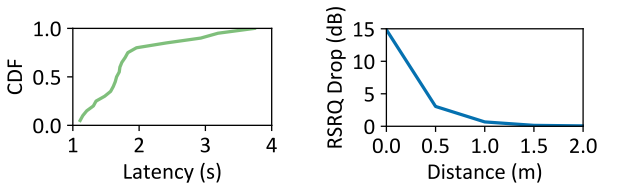
Ethical Considerations This work does not raise any ethical issues. Our testbed is carefully controlled for experiments, operating on an unlicensed 2.4GHz ISM band. The experimentation is conducted within a 5MHz channel centered at 2.49GHz. We ensure no nearby device is using the frequency band. The radio signal emitted by the testbed only reaches a few meters, ensuring that no other device is affected or attempts to communicate with our testbed. Meanwhile, we are working with collaborating mobile operators about the discovered solutions and will open source CellDAM.

10.2 Evaluation Results

Evaluation of CellDAM Attack Detection In this subsection, we answer the question of whether CellDAM can detect all attacks displayed in Table 2. For this purpose, we inject attack messages according to the attacker procedure, and observe if the detector can initiate warnings as expected. We test the attack detection under two different traffic types: A lightweight traffic with ping and a heavy traffic with iPerf3 [30] to saturate the channel. For each attack, we repeat 1,000 times under each scenario. We evaluate the results with three metrics: precision, recall, and F1 score. The ground truth can be easily obtained, as whether there is an ongoing attack is controlled by us.

Table 3 summarizes the precision, recall, and F1 score for the CellDAM detection. As shown in the table, the detection reaches a precision of 98.9%∼100% for different attack types. The high precision is achieved by the correctness of the DFA and verification, as the normal operations in legitimate 5G delivery will follow the correct procedure. Meanwhile, the recall is 70.5%∼100% for different attack detection. Note that, the relatively low recall for A1 is because a heavy-traffic scenario will extend the ON state for a device. The device can use other FBS detection methods [8, 65] to complement CellDAM. The recall is high for other attacks, as they will incur undesired data-plane signaling. This results in a high F1 score of 0.823∼0.999. The detection works well for both light and heavy traffic. This is because CellDAM only requires inspection on lightweight data-plane signaling messages. We also measure the average detection latency for attack detection. The signaling messages could be captured and fed into the detector for real-time detection. The detection achieves an average latency of 28ms. Therefore, CellDAM can quickly spot potential adversaries and take action.

Evaluation of CellDAM Mitigation We evaluate the success rate for mitigation. When SecHub detects any attack, it will trigger the target device handover to escape from the attack. We enable the handover-related RRC messages and config in gNB and let the UE monitor the handover events. Note that our testbed does not support real handover to a different cell; if UE receives a handover command, we assume a successful handover. We evaluate the ratio of successful handover and the corresponding latency for each mitigation. In all 40 rounds of experiments, SecHub successfully triggers the UE handover. Figure 8a shows the CDF of the latency. The results show that the average mitigation latency is 1.85s. 90% of them could successfully handover within 3 seconds, and all handovers are triggered within 4 seconds.



(a) CDF of the mitigation latency for "quick dodge".

(b) Relationship between the distance and RSRQ drop.

Figure 8: Mitigation Performance.

In our design, signals from SecHub will have minimum impact on other devices. We show this point by measuring the perceived RSRQ drops at the UE at different distances from the SecHub. Figure 8b shows that the RSRQ drop at 1m and 2m are only 0.67dB and 0.06dB, respectively. With the controlled power of SecHub, our mitigation will only trigger handover on the close-by protected device, while not affecting other users or devices.

Impact of CellDAM mitigation on applications We discussed in §7 that handover-based cell switching will incur little overhead on applications. We now evaluate the disruption time on the iPerf3 application during the band switch. We decode the logs and calculate the interval of application packets before and after the handover. The average disruption is 72.3 ms, with a 95th percentile of 83.7ms. We further note that, the packets during 5G handover will be kept and delivered by the new cell afterwards. Therefore, an application would only experience a small delay caused by CellDAM, without triggering any data loss or TCP connection reset.

Evaluation of SecHub Rootless Capture We then present our microbenchmarks for inferring signaling messages. We measure the ratio of correctly captured signaling in both uplink and downlink to show the effectiveness of our rootless signaling capture. We record the traces of PUCCH/PDCCH (SR and DCI) and PDSCH/PUSCH (MAC CE and RLC Control) at the UE as the ground truth. We capture the traces through the SecHub under different traffic scenarios and mobility. We use ping and iPerf3 for the light and heavy traffic scenarios, respectively. The UE and SecHub are kept static or moving at a speed of around 5km/h for mobility. The ratio of correctly decoded signaling is calculated by comparing the traces captured on the SecHub and the ground truth.

Table 4 shows the success ratio for the rootless capture. For

Traffic	Control		Data	
	PUCCH	PDCCH	PUSCH	PDSCH
Light	99.1%	100%	100%	99.7%
Heavy	98.7%	99.9%	98.1%	97.2%
Light-M	98.8%	99.9%	99.1%	98.3%
Heavy-M	98.1%	99.8%	98.0%	97.3%

Table 4: Ratio of success rootless signaling message capture under different traffic scenarios and mobility (-M: Mobile).

the control messages, 99.1% of PUCCH and 100% of PDCCH are successfully captured and decoded from the SecHub with the light traffic. For the heavy traffic, the SecHub still achieves a high success rate with 98.7% for PUCCH and 99.9% for the PDCCH. For data messages, SecHub successfully decodes all the PUSCH messages and 99.7% of PDSCH data under the light traffic scenario. 98.1% of the PUSCH and 97.2% of the PDSCH traffic is successfully captured and decoded for the heavy traffic scenario. We observe similar numbers (97.3-99.9%) in mobility cases (Light-M) and (Heavy-M). The ratio is not impacted by mobility, because the UE and SecHub experience similar channel conditions. Whether one message is decoded on UE or not, SecHub will produce similar results.

The high success rate is possible with accurate C-RNTI inference. We quantify the success rate with the collaborated traffic fingerprinting. Since the inference can be done without actively sending signals over the air, we perform the verification on both the commercial network and our testbed. The ground truth of C-RNTI can be acquired in gNB (by checking logs) and in COTS UE (by using MobileInsight [37]). Every time SecHub collects 5s of traces for the inference after the traffic pattern coordination with the target UE. We perform 120 rounds of experiments with 60 rounds on the testbed and 60 rounds on the commercial network.

On the testbed, SecHub correctly infers the C-RNTI in all 60 rounds, achieving a 100% success rate. On the commercial network, with the increased device number, SecHub successfully infers 98.3% of the C-RNTI. We further measure the overheads caused by the background ping in the continuous tracking. The result shows that CellDAM involves 0.14 KBps traffic overhead, which is marginal on the target device. The results show that CellDAM could continuously perform the monitoring during the mobility, and protect the victim without any root privilege.

11 Related Work

As new attacks on the 5G/4G have drawn increasing attention, detecting potential attacks is a popular research topic in recent years. Due to the high overhead and long cycle for network-side detection [47], current detection methods are mainly on the device side. PHOENIX [19] proposes a solution for control-plane monitoring. [53] studies the device-side attack detection for core network attacks. [16, 18, 25, 38, 63] detect cellular attacks by analyzing on-device application traces. [8, 65] detect the existence of FBS with physical characteristics of FBS such as power or signal signatures. No prior work studies attack detection for data-plane protocols. CellDAM provides the first detection scheme for attacks on data-plane packets/signaling in 5G. Unlike other works that require root access and expose additional risks [62], CellDAM detects the attack without extra privilege.

Existing mitigation for attacks on 4G/5G protocols either needs root access [33] or requires protocol changes [48, 49, 54, 64]. To our knowledge, CellDAM is the first solution that provides rootless mitigation for data-plane attacks without firmware/standard changes. Other mitigation methods for attacks on mobile apps [11, 13, 39] or cellular-based services [35, 40, 55, 58] are orthogonal to our work.

Prior studies have leveraged model checking to verify the cellular standards and discover new vulnerabilities in the protocols. [26, 28] exposed attacks in 4G LTE by adversarial model-based testing. Previous work also formally analyzed the 5G protocol components including the 5G-AKA procedures [12, 17] and NAS/RRC signalings [27]. However, the existing cellular protocol verification runs offline on the control plane and does not have runtime detection. CellDAM performs the runtime verification to discover the potential attacks timely. It targets the relatively more difficult problem of verifying the data plane, whose traffic is heavier.

12 Conclusion

Detection of data-plane attacks at the mobile device has not been viewed favorably to date. This is due to the excessive processing and energy overhead associated with 5G high data rate. In this work, we show how to use data-plane signaling messages to devise a detection solution that yields one or two orders of magnitude lower overhead. We leverage the fact that data-plane attacks would exhibit certain data signaling misbehavior. Our reactive solution may defend against certain attacks that the current proactive schemes cannot. It can work on normal user devices without root privilege or infrastructure upgrade. Once CellDAM detects an attack, it further mitigates attack damages via handover to another available channel.

In a broader scope, we believe data-plane security deserves more attention given that activating all security measures on lightweight control-plane messages is relatively straightforward. In contrast, data plane delivery involves complex interactions across protocols and the adversary has plenty playground to launch various attacks from applications, transport layer, to IP and 5G protocols. While the end-to-end approach and existing 5G data-plane security may secure application data, it fails to secure the 5G infrastructure and mobile device. To this end, this work reports our initial effort to explore a lightweight, reactive solution to 5G data-plane security.

Acknowledgments. We would like to thank the anonymous reviewers and our shepherd, Dr. Aaron Schulman, for their constructive comments and feedback.

References

[1] 3GPP. TS33.501: Security architecture and procedures for 5G System-V15.4.0, May. 2019.

[2] 3GPP. TS36.321: Evolved Universal Terrestrial Radio Access (E-UTRA); Medium Access Control (MAC) protocol specification, Sep. 2019.

[3] 3GPP. NR; Medium Access Control (MAC) protocol specification, Dec. 2020.

[4] 3GPP. TS33.809: Study on 5G security enhancements against False Base Stations (FBS), Dec. 2020.

[5] 3GPP. NR; Radio Link Control (RLC) protocol specification, Jan. 2021.

[6] 3GPP. TS38.211: NR; Physical channels and modulation, Jan. 2021.

[7] 3GPP. TS33.501: Security architecture and procedures for 5G System-V16.4.0, Mar. 2022.

[8] ALI, A., AND FISCHER, G. Enabling fake base station detection through sample-based higher order noise statistics. In *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)* (2019), IEEE, pp. 695–700.

[9] API, A. B. https://developer.android.com/guide/topics/connectivity/bluetooth.

[10] API, A. T. https://developer.android.com/reference/android/telephony/package-summary.

[11] BALAPOUR, A., NIKKHAH, H. R., AND SABHERWAL, R. Mobile application security: Role of perceived privacy as the predictor of security perceptions. *International Journal of Information Management 52* (2020), 102063.

[12] BASIN, D., DREIER, J., HIRSCHI, L., RADOMIROVIC, S., SASSE, R., AND STETTLER, V. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (2018), pp. 1383–1396.

[13] BUI, D., YAO, Y., SHIN, K. G., CHOI, J.-M., AND SHIN, J. Consistency analysis of data-usage purposes in mobile apps. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 2824–2843.

[14] CALCULATIONS, U. W. R. https://www.electronicdesign.com/technologies/communications/article/21796484/understanding-wireless-range-calculations.

[15] CASATI, L., AND VISCONTI, A. The dangers of rooting: data leakage detection in android applications. *Mobile Information Systems 2018* (2018).

[16] CHLOSTA, M., RUPPRECHT, D., HOLZ, T., AND PÖPPER, C. Lte security disabled: misconfiguration in commercial networks. In *Proceedings of the 12th conference on security and privacy in wireless and mobile networks* (2019), pp. 261–266.

[17] CREMERS, C., AND DEHNEL-WILD, M. Component-based formal analysis of 5g-aka: Channel assumptions and session confusion.

[18] DABROWSKI, A., PIANTA, N., KLEPP, T., MULAZZANI, M., AND WEIPPL, E. Imsi-catch me if you can: Imsi-catcher-catchers. In *Proceedings of the 30th annual computer security applications Conference* (2014), pp. 246–255.

[19] ECHEVERRIA, M., AHMED, Z., WANG, B., ARIF, M. F., HUSSAIN, S. R., AND CHOWDHURY, O. Phoenix: Device-centric cellular network protocol monitoring using runtime verification. *arXiv preprint arXiv:2101.00328* (2021).

[20] FALKENBERG, R., AND WIETFELD, C. Falcon: An accurate real-time monitor for client-based mobile network data analytics. In *2019 IEEE Global Communications Conference (GLOBECOM)* (2019), IEEE, pp. 1–7.

[21] GASPARIS, I., QIAN, Z., SONG, C., AND KRISHNAMURTHY, S. V. Detecting android root exploits by learning from root providers. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 1129–1144.

[22] HO, T.-H., DEAN, D., GU, X., AND ENCK, W. Prec: practical root exploit containment for android devices. In *Proceedings of the 4th ACM conference on Data and application security and privacy* (2014), pp. 187–198.

[23] HOLTMANNS, S., RAO, S. P., AND OLIVER, I. User location tracking attacks for lte networks using the interworking functionality. In *2016 IFIP Networking conference (IFIP Networking) and workshops* (2016), IEEE, pp. 315–322.

[24] HOLTMANNS, S., RAO, S. P., AND OLIVER, I. User location tracking attacks for lte networks using the interworking functionality. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops* (2016), pp. 315–322.

[25] HONG, B., BAE, S., AND KIM, Y. Guti reallocation demystified: Cellular location tracking with changing temporary identifier. In *NDSS* (2018).

[26] HUSSAIN, S., CHOWDHURY, O., MEHNAZ, S., AND BERTINO, E. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *Network and Distributed Systems Security (NDSS) Symposium 2018* (2018).

[27] HUSSAIN, S. R., ECHEVERRIA, M., KARIM, I., CHOWDHURY, O., AND BERTINO, E. 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 669–684.

[28] HUSSAIN, S. R., KARIM, I., ISHTIAQ, A. A., CHOWDHURY, O., AND BERTINO, E. Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 1082–1099.

[29] IETF. The Transport Layer Security (TLS) Protocol Version 1.3, August 2018. https://datatracker.ietf.org/doc/rfc8446/.

[30] IPERF3. https://github.com/esnet/iperf.

[31] KASPERSKY. Rooting your Android: Advantages, disadvantages, and snags, June 2017. https://usa.kaspersky.com/blog/android-root-faq/11581/.

[32] KOHLS, K., RUPPRECHT, D., HOLZ, T., AND PÖPPER, C. Lost traffic encryption: fingerprinting LTE/4G traffic on layer two. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks* (2019), pp. 249–260.

[33] KOTULIAK, M., ERNI, S., LEU, P., ROESCHLIN, M., AND CAPKUN, S. Ltrack: Stealthy tracking of mobile phones in lte. In *31st USENIX Security Symposium (USENIX 2022)* (2022).

[34] LAB, K. Rooting your android: Advantages, disadvantages, and snags. https://www.kaspersky.com/blog/android-root-faq/17135/, Jun. 2017.

[35] LI, C.-Y., TU, G.-H., PENG, C., YUAN, Z., LI, Y., LU, S., AND WANG, X. Insecurity of voice solution volte in lte mobile networks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 316–327.

[36] LI, M., ZHU, H., GAO, Z., CHEN, S., YU, L., HU, S., AND REN, K. All your location are belong to us: Breaking mobile social networks for automated user location tracking. In *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing* (2014), pp. 43–52.

[37] LI, Y., PENG, C., YUAN, Z., LI, J., DENG, H., AND WANG, T. Mobileinsight: Extracting and analyzing cellular network information on smartphones. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking* (2016), pp. 202–215.

[38] LI, Y., PENG, C., ZHANG, Z., TAN, Z., DENG, H., ZHAO, J., LI, Q., GUO, Y., LING, K., DING, B., ET AL. Experience: a five-year retrospective of mobileinsight. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (2021), pp. 28–41.

[39] LU, H., XING, L., XIAO, Y., ZHANG, Y., LIAO, X., WANG, X., AND WANG, X. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020), pp. 569–585.

[40] LU, Y.-H., LI, C.-Y., LI, Y.-Y., HSIAO, S. H.-Y., XIE, T., TU, G.-H., AND CHEN, W.-X. Ghost calls from operational 4g call systems: Ims vulnerability, call dos attack, and countermeasure. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (2020), pp. 1–14.

[41] MJØLSNES, S. F., AND OLIMID, R. F. Easy 4G/LTE IMSI catchers for non-programmers. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security* (2017), Springer, pp. 235–246.

[42] OPENSIGNAL. USA 5G Experience Report, Jan 2022. https://www.opensignal.com/reports/2022/01/usa/mobile-network-experience-5g.

[43] PARK, C., BAE, S., OH, B., LEE, J., LEE, E., YUN, I., AND KIM, Y. Doltest: In-depth downlink negative testing framework for lte devices. In *USENIX Security Symposium* (2022).

[44] POSITIVE TECHNOLOGIES. Security assessment of Diameter networks, 2020.

[45] QUALCOMM. QxDM Professional - QUALCOMM eXtensible Diagnostic Monitor. http://www.qualcomm.com/media/documents/tags/qxdm.

[46] RAO, S. P., KOTTE, B. T., AND HOLTMANNS, S. Privacy in lte networks. In *Proceedings of the 9th EAI International Conference on Mobile Multimedia Communications* (2016), pp. 176–183.

[47] RUPPRECHT, D., DABROWSKI, A., HOLZ, T., WEIPPL, E., AND PÖPPER, C. On security research towards future mobile network generations. *IEEE Communications Surveys & Tutorials 20*, 3 (2018), 2518–2542.

[48] RUPPRECHT, D., KOHLS, K., HOLZ, T., AND PÖPPER, C. Breaking LTE on layer two. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1121–1136.

[49] RUPPRECHT, D., KOHLS, K., HOLZ, T., AND PÖPPER, C. IMP4GT: Impersonation attacks in 4G networks. In *Symposium on Network and Distributed System Security (NDSS). ISOC* (2020).

[50] SHAO, Y., LUO, X., AND QIAN, C. Rootguard: Protecting rooted android phones. *Computer 47*, 6 (2014), 32–40.

[51] SRSRAN. https://www.srslte.com/.

[52] STATISTA. Where 5G Technology Has Been Deployed , July 2022. https://www.statista.com/chart/23194/5g-networks-deployment-world-map/.

[53] TAN, Z., DING, B., ZHANG, Z., LI, Q., GUO, Y., AND LU, S. Device-centric detection and mitigation of diameter signaling attacks against mobile core. In *2021 IEEE Conference on Communications and Network Security (CNS)* (2021), IEEE, pp. 29–37.

[54] TAN, Z., DING, B., ZHAO, J., GUO, Y., AND LU, S. Data-plane signaling in cellular iot: attacks and defense. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (2021), pp. 465–477.

[55] TU, G.-H., LI, C.-Y., PENG, C., LI, Y., AND LU, S. New security threats caused by ims-based sms service in 4g lte networks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1118–1130.

[56] US, W. What is 5G? Facts Stats You Need to Know, April 2022. https://www.whistleout.com/CellPhones/Guides/5G-statistics.

[57] VIRTUALISATION, N. F. An introduction, benefits, enablers, challenges & call for action. In *White Paper, SDN and OpenFlow World Congress* (2012).

[58] WANG, S., TU, G.-H., LEI, X., XIE, T., LI, C.-Y., CHOU, P.-Y., HSIEH, F., HU, Y., XIAO, L., AND PENG, C. Insecurity of operational cellular iot service: new vulnerabilities, attacks, and countermeasures. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (2021), pp. 437–450.

[59] XU, D., ZHOU, A., ZHANG, X., WANG, G., LIU, X., AN, C., SHI, Y., LIU, L., AND MA, H. Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 479–494.

[60] YANG, H., BAE, S., SON, M., KIM, H., KIM, S. M., AND KIM, Y. Hiding in plain signal: Physical signal overshadowing attack on LTE. In *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 55–72.

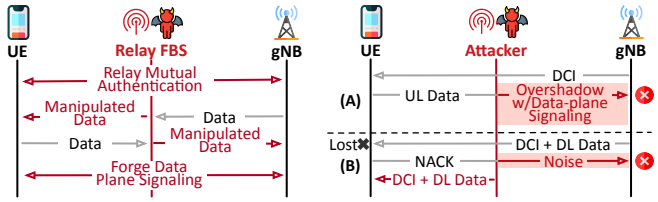
[61] ZEROMQ. ZeroMQ: An open-source universal messaging library . https://zeromq.org/, Jan 2022.

[62] ZHANG, H., SHE, D., AND QIAN, Z. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 1093–1104.

[63] ZHANG, Y., LIU, B., LU, C., LI, Z., DUAN, H., HAO, S., LIU, M., LIU, Y., WANG, D., AND LI, Q. Lies in the air: Characterizing fake-base-station spam ecosystem in china. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020), pp. 521–534.

[64] ZHAO, J., DING, B., GUO, Y., TAN, Z., AND LU, S. Securesim: rethinking authentication and access control for sim/esim. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (2021), pp. 451–464.

[65] ZHUANG, Z., JI, X., ZHANG, T., ZHANG, J., XU, W., LI, Z., AND LIU, Y. Fbsleuth: Fake base station forensics via radio frequency fingerprinting. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (2018), pp. 261–272.



(a) Relay FBS (b) Overshadowing

Figure 9: Viable data-plane methodologies for data injection/manipulation.

A Details of Each Attack and Its Detection

In this section, we present the details of each attack shown in Table 2, including the attack procedure that leverages the forgery messages and the attack consequences. We also elaborate on how each attack incurs undesired behavior that can be detected by `CellDAM` validations.

A.1 Injecting Data with FBS

Attack Procedure The attacker can use a relay fake base station (FBS) to manipulate data packets [48]. The detailed attack procedure is shown in Figure 9a. The attacker sets up a fake base station (FBS). It usually runs on a different band from the real base station, and sends strong broadcast signal to lure the device into connection. The attack also sets up a fake UE that connects to the real base station. It then relays the connection setup messages without any manipulation from/to the real base station. Afterwards, although all data packets are encrypted, the attacker can flip the bits to change the content of the forgery, if data-plane integrity protection is not enforced. This is doable as the encryption is done with simple XOR. as long as the original content is known (e.g., DNS server set up by the operator), the attacker can flip bits and change the contents to target values. For instance, the attacker can manipulate the IP header of a DNS request and compensate the IP header checksum to pass the checks [48].

Undesired Behavior In this attack, we assume the adversary cannot infer the data-plane configurations encrypted in RRC messages. Therefore, when the attacker forwards the data packet, some configurations might be incorrectly set and detected on the device side. One example is DRX configuration. Without the configuration, the forwarded packet might fall in the DRX OFF period, as shown in Figure 10a. As the DRX ON period is usually very short (e.g., 10ms), most messages might be delivered outside of the period, violating check c_4. Although the FBS can repeat transmitting until the victim device acknowledges to ensure delivery, this behavior will be detected by `CellDAM`. Note that, 5G DRX includes the mechanism to stay in ON period for an extended period when a new data is received. Therefore, when the traffic is heavy, the device might keep staying in ON state. As we claimed in §10, the attack will be more detectable for light traffic. For this attacker, the PHY layer detection methods mentioned in §11 can help detect FBS that transmits abnormal signal.

A.2 Data Manipulation with Retransmission

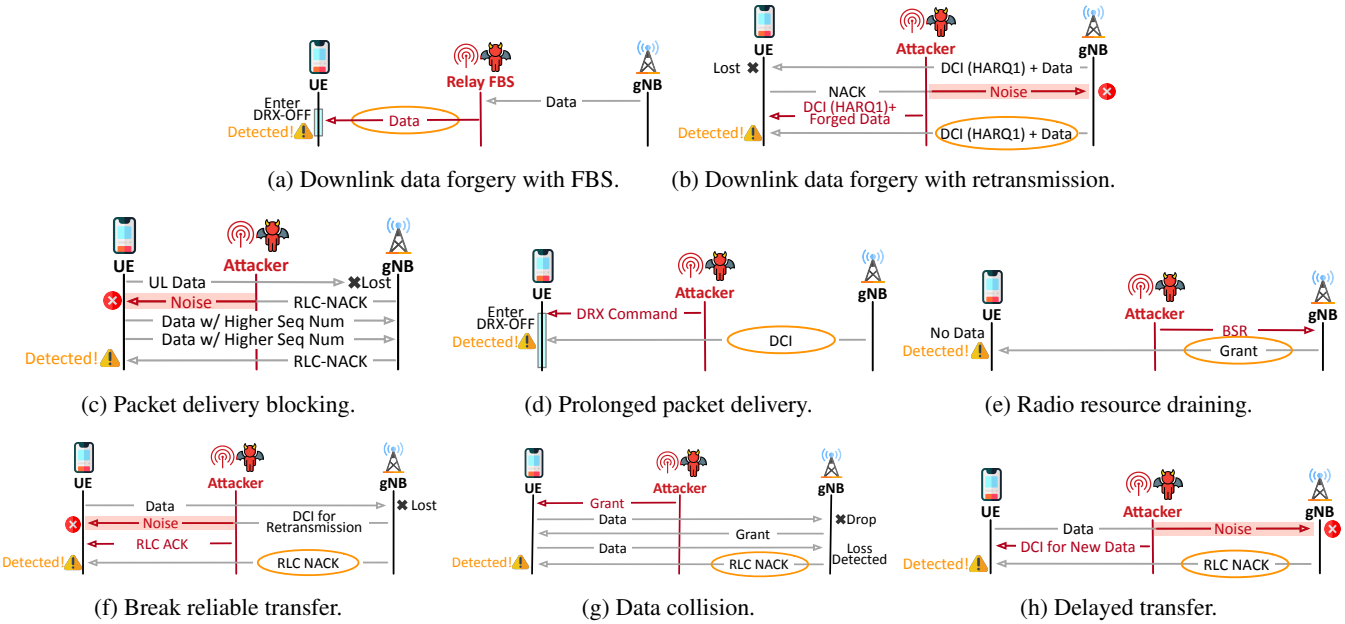
Attack Procedure We also show that, the attacker can serve man-in-the-middle to manipulate data packets without FBS. The detailed attack procedure is shown in Figure 9b. This approach can manipulate data plane packet without FBS. We consider the victim device is directly connected to the authentic base station. Note that, the attacker might not be able to forge data-plane packets directly, as they are encrypted (unlike integrity protection, which is optional). Therefore, to forge data packets, the attacker still needs to take the bit flipping approach as in A1. One viable way is to forge the data as retransmission. The attacker can eavesdrop on the channel and look for data transmission that fails on victim device (i.e., trigger NACK), while it successfully decodes the encrypted data (due to less noisy environment, etc.). The attacker can then forge the DCI and manipulated data as the retransmission.

Undesired Behavior This attack requires sending forged DCI and data to the device. This DCI can be received in an improper context. Each DCI includes a HARQ ID, which indicates the process of the transmission. Each process takes a stop-and-wait procedure. Before the last packet is acknowledged, the process will not move on to transmit the next one. Therefore, the DCI from the authentic gNB can arrive after the DCI forged by attacker and before the forged retransmission has been acknowledged. This causes an undesired behavior that can be observed on the device with c_1. This is shown in Figure 10b.

We further note two things. First, this attack method can forge *uplink* data packet without obvious undesired behavior on the device side. We do not consider such attack with pure uplink forgery in this work. However, a reasonable forgery attack needs to manipulate both uplink and downlink data. Second, an interested reader might ask whether the attacker can use the legitimate DCI to send the forged data. However, the DL DCI and the corresponding forged data are usually sent in the same time slot in 5G. It is thus hardly possible to infer DCI for data forgery in advance.

A.3 Packet Delivery Blocking

Attack Procedure We have introduced this attack in §3.2 as an example of corrupting data-plane signaling. We now present this attack in more details as shown in Figure 10c. The attacker first eavesdrops on the data channel and learns the packet sequence number that is not delivered over the air. It then selectively corrupts the RLC control that NACKs the packet. Since the packet is not acknowledged, UE will still send uplink data without retransmitting the missing one. These new data packets are blocked in the gNB, which suffers



(a) Downlink data forgery with FBS. (b) Downlink data forgery with retransmission.

(c) Packet delivery blocking. (d) Prolonged packet delivery. (e) Radio resource draining.

(f) Break reliable transfer. (g) Data collision. (h) Delayed transfer.

Figure 10: Illustration of undesired behavior caused by attacks.

head-of-line blocking for more than 100ms. No new data will be forwarded during this time period.

Undesired Behavior The attack will cause undesired behavior on RLC. From the perspective of the device, the uplink RLC will receive either ACK or NACK after $T_{reordering}$ when the timer expires. Even if the signaling is corrupted, which is rare given its small size, the MAC layer should retransmit it. Therefore, if the device finds no RLC control after timer expires and MAC retransmissions (which has a configured max count), `CellDAM` detects a potential corruption attack. We note that all MAC retransmission could fail due to extremely weak channel instead of an attacker. In this case, although no attack is present, switching to a better wireless channel with `CellDAM` is a reasonable facilitating option.

A.4 Prolonged Packet Delivery

We consider connected mode Discontinuous Reception (CDRX) in this attack. In the RRC connected state, the gNB will only deliver data during DRX ON state. A device will be in DRX ON for a small time period in a fixed periodicity. If any data is received during this period, the DRX ON state is extended for a constant amount of time. gNB will configure the ON period, DRX cycle, and the extended timer amount in encrypted RRC messages.

Attack Procedure The detailed procedure is shown in Figure 10d. In this attack, the adversary forges a DRX command to the device. DRX command terminates the DRX ON state prematurely and the device enters the sleep mode. Therefore, the device will be unable to receive all subsequent transmission in the current DRX cycle with DRX turned to OFF state. The data delivery can be delayed for hundreds of milliseconds given long DRX cycle. The attack is adapted from [54]. Note that, this attack will not stop uplink data delivery, as a UL data transmission initiated by UE can again switch the DRX state to DRX ON.

Undesired Behavior It is not possible to receive downlink DCI or data during DRX OFF. If such an event happens, the previous DRX command could be forged. As `CellDAM` does not have root access, it is difficult for `SecHub` to monitor internal DRX state in real-time or infer the state with encrypted DRX configurations in RRC messages. However, there is another indication for a message during DRX OFF: If the device is in DRX OFF, it will not respond to any downlink data with ACK or NACK in PUCCH. Therefore, `SecHub` can detect such attack by checking the downlink data without any acknowledgment, after an incoming DRX command. This violates the validation check c_4. In addition, gNB will not send DRX command when the previous DL transmission has not finished, which means there will be more transmission in the DRX cycle. This violated c_1.

A.5 Radio Resource Draining

Attack Procedure We have introduced this attack in §3.2 as an example of manipulating data-plane signaling. Here we present more details. 5G/4G adopts scheduling-based data delivery. To transmit uplink data, the device needs to send the buffer status report (BSR) to the gNB for asking grants. An attacker can forge a BSR to the gNB. In the forged BSR, the attacker falsely indicates the victim device has a large amount of data to send. The gNB subsequently assigns excessive resource blocks to the device, wasting wireless resource and

blocking other users' access. The detailed procedure is shown in Figure 10e. The attack is adapted from [54].

Undesired Behavior Although the attacker forges an uplink message, it will incur observable undesired behavior on the device side as well. This is because the DCI (for UL grant) will be triggered by the forged BSR message. If the device receives grant when there is no prior request, the grant can be caused by a forged request by the attacker. We notice that, a gNB can send a device "free" small grants in case the device has something to send. However, these grants are small for the device to sufficiently deliver BSR. Therefore, to reduce false positive, we set a threshold (120 bytes) in the verification check c_5 to find unwanted resources.

A.6 Break Reliable Transfer

Attack Procedure The detailed procedure of the attack is shown in Figure 10f. The attacker can corrupt the data retransmission on MAC layer. It then forges an RLC ACK to UE. Receiving it, the victim device RLC protocol wrongly thinks the packet has been delivered, discarding it in the buffer. Therefore, this packet cannot be reliably delivered in 5G. This might further trigger TCP retransmission, which can cause more serious damage.

Undesired Behavior From the perspective of the base station, it will detect a packet gap in RLC protocol when it receives a later packet from UE. Therefore, it will still attempt to recover it by sending an RLC NACK. The NACK timing is unknown for the attacker, thus cannot be targeted for corruption. The device side will thus receive an RLC NACK first and then an ACK. This behavior is undesired in 5G and can be detected by validation check c_3.

A.7 Data Collision

Attack Procedure The detailed procedure of the attack is shown in Figure 10g. The attacker forges grant to the victim device. The device will send data using the forged grant. However, any data using these non-authorized grants will not be correctly accepted by the gNB. In addition, the legitimate transmission in the same time and frequency by another user will be corrupted by the victim's false transmission.

Undesired Behavior The attack will trigger undesired behavior on the device side. Since the UL data using false grants will not be accepted, gNB will not ACK or NACK the message delivery and consequently trigger an RLC NACK. `CellDAM` detects the undesired behavior with RLC control NACK and lack of MAC layer feedback with validation c_5.

A.8 Delayed Transfer

Attack Procedure The detailed procedure of the attack is shown in Figure 10h. In this attack, the adversary sends noises and corrupts the uplink data. The attacker can learn the time and frequency of the delivery by eavesdropping on DCI for UL grants. It consequently forges DCI for new data (i.e., indicator for ACK) to stop the UE from retransmitting the corrupted data. Consequently, the UE will start sending new data on MAC. This will later trigger retransmission on the RLC layer, which can take up to hundreds of milliseconds compared to a MAC fast retransmission.

Undesired Behavior The forged DCI from the attacker can be sent in a wrong context, where two consecutive DCIs are received but the data using the first one has not been acked yet. This can also be detected with c_1.

B Deriving Minimal Power for Targeted Switching

The solution should not affect other devices. `SecHub` adaptively controls its power upon triggering the handover. Previous 5G measurements show that -20dB RSRQ is enough to trigger the handover in more than 98% cases [59]. `SecHub` adaptively derives the minimal power so that RSRQ drops to -20dB, thus triggering handover. The current RSRQ is derived by:

$$RSRQ = \frac{N \times RSRP}{RSSI}$$

To reduce the RSRQ to -20dB, the minimal power (P_m) needed by `SecHub` follows:

$$\frac{N \times RSRP}{RSSI + P_m} = -20dB = \frac{1}{100}$$

Thus, the P_m could be derived by:

$$P_m = N \times RSRP \times (100 - \frac{1}{RSRQ})$$

All needed information can be acquired from the victim by the OS API (e.g., Android [10]) without root privilege. Furthermore, the power density is inversely proportional to the square of the distance from the antenna [14]. Assume `SecHub` is located close to the device (<0.1m). The RSRQ drop at the 1m distance is smaller than 1dB, which could be neglected by other devices. Only the victim device perceives a notable RSRQ drop and triggers its handover.

We admit that, even theoretically not affecting any other device, sending weak signals might require licensing from the operational networks or government. We envision `SecHub` can acquire such permission from mobile operators, or even manufactured by the operators themselves. If this privilege is not available, `CellDAM` could fallback to using airplane switching. Even if the cell is not changed after toggling, the device will be reassigned a physical ID C-RNTI, which makes it more difficult for the attacker to track the victim.

C List of Accepted Messages

We elaborate on c_1 by enumerating each state and the list of accepted message for each one. The results are shown in Table 5. If a message is in the list, we show the next state if all other validations are passed. Otherwise, we mark an $\times$ in the table which means an undesired behavior that fails c_1. As we mentioned, our method prioritizes soundness. Therefore, for messages that are not explicitly considered, `CellDAM` will ignore them and stay in the current state.

D Continuous Inference

We describe how `SecHub` could keep tracking the C-RNTI used by the victim device.

Challenge: Dynamic configurations Upon user mobility, the configurations could be updated within an encrypted message after the device connects to a new gNB. It is also possible that gNB updates the C-RNTI for the device upon RRC state changes without user mobility. Tracking the up-to-date configurations for the target device is critical.

Can we track the config change? One solution idea is to track the configuration change once it happens and launch the inference again. This is possible when a handover happens. `CellDAM` develops an application on the target device to track the possible configuration changes due to mobility. The application leverages the existing API to detect the PCI/band change due to handover. It requires no root access. The application could track the updates with OS-level API and notify the `SecHub` to start a new round of C-RNTI inference.

However, the same method cannot be used to infer the config change within the same cell. `SecHub` or OS APIs cannot report the change of configurations from the base station.

Idea: Prevent config change in a cell Since change detection is hard, we approach it differently by keeping the configuration constant. Indeed, this is possible. `CellDAM` leverages the operational C-RNTI update logic in the cellular deployment to retain the same C-RNTI when the device stays on the same gNB. Our study on commercial devices and operators shows that, the current gNB updates C-RNTI when the device transits from the RRC-Idle state to the RRC-Connected state. The gNB recycles the C-RNTI from the idle devices and reuses them for other devices. Therefore, we aim to keep the device in RRC-Connected to avoid configuration change.

Preventing configuration change for continuous inference We trigger a lightweight ping in the background inside the application. Our experiments show that the ping traffic with 2s interval could keep the C-RNTI unchanged. We validate it on 216 cells from three major US carriers. In all tests, the C-RNTI remains unchanged for at least 30 minutes with our light background traffic. To tolerate unexpected updates, `SecHub` also triggers the C-RNTI inference (in §6) every 10 minutes to validate that the current configuration is up-to-date.

Marginal energy consumption The ping messages incur low traffic volume to keep the device in RRC-Connected. We further note that, this has small impact on 5G energy saving. This is because, the device could still go to sleep mode for energy reservation in logical RRC-Connected state. A 5G device saves power by entering Discontinuous Reception or DRX OFF mode. However, given the device will quickly re-enter DRX OFF (within 100ms) after a data transmission, the infrequent messages (every 2s) would only incur small energy overhead. Besides, regular user traffic will also wake up the device, during which the extra ping does not further reduce sleep period. In addition, the impact on the energy from the sleep mode has a smaller impact as compared to other factors, such as the screen brightness. We run the application for 30 minutes while normally using the device (with mixed heavy and infrequent messages). The additional energy only incurs 0.5% extra energy on average.

E ACRONYMS

5GC 5G Core Network
AKA Authentication and Key Agreement
BSR Buffer Status Report
C-RNTI Cell Radio Network Temporary Identifier
CE Control Element
COTS Commercial Off-the-shelf
DCI Downlink Control Information
DFA Deterministic Finite Automata
DL Downlink
DRX Discontinuous Reception
DTLS Datagram Transport Layer Security
FSM Finite State Machine
gNB gNodeB, 5G Base Station
HARQ Hybrid Automatic Repeat Request
LTE Long Term Evolution
MAC Medium Access Control
NAS Non Access Stratum
NSA Non Standalone
PCI Physical Cell ID
PDCCH Physical Downlink Control Channel
PDCP Packet Data Convergence Protocol
PDSCH Physical Downlink Shared Channel
PHY Physical Layer
PUCCH Physical Uplink Control Channel
PUSCH Physical Uplink Shared Channel
RLC Radio Link Control
RRC Radio Resource Control
RSRP Reference Signals Received Power
RSRQ Reference Signal Received Quality
SN Sequence Number
SR Scheduling Request
TLS Transport Layer Security
UE User Equipment
UL Uplink

Table 5: List of the accepted data-plane signaling for each DFA state and their state transition. We do not include s_5 and s_9 in the table, as they are accept states. $\times$ means that the message in this state is not allowed and cannot pass c_1, $-$ means that the state is unchanged with this message. "Different" or "same" means the receiving DCI compared with the first DCI for an RLC data packet, i.e., RLC retransmission will reset HARQ and NDI with the first DCI after.

Data-Plane Signaling Message	Current State						
	s_1	s_2	s_3	s_4	s_6	s_7	s_8
MAC DCI for DL grant with same HARQ, same NDI	$-$	$-$	$-$	$-$	s_7	$\times$	$\times$
MAC DCI for DL grant with same HARQ, flipped NDI	$-$	$-$	$-$	$-$	$\times$	$\times$	$-$
MAC DCI for DL grant with different HARQ	$-$	$-$	$-$	$-$	$-$	$-$	$-$
MAC DCI for UL grant with same HARQ, same NDI	s_2	$\times$	s_2	$\times$	$-$	$-$	$-$
MAC DCI for UL grant with same HARQ, flipped NDI	$\times$	$\times$	s_4	$\times$	$-$	$-$	$-$
MAC DCI for UL grant with different HARQ	$-$	$-$	$-$	$-$	$-$	$-$	$-$
PUCCH ACK for the previous DCI	$-$	$-$	$-$	$-$	$\times$	s_8	$\times$
PUCCH NACK for the previous DCI	$-$	$-$	$-$	$-$	$\times$	s_6	$\times$
RLC Control with ACK for this packet	$\times$	$\times$	$\times$	s_5	$\times$	$\times$	s_9
RLC Control with NACK for this packet	$\times$	s_1	s_1	$\times$	s_6	s_6	$\times$
DRX Command	$-$	$\times$	$\times$	$\times$	$-$	$\times$	$\times$
BSR	$-$	$-$	$-$	$-$	$-$	$-$	$-$
Any other messages	$-$	$-$	$-$	$-$	$-$	$-$	$-$

LOCA: A Location-Oblivious Cellular Architecture

Zhihong Luo *UC Berkeley*
Silvery Fu *UC Berkeley*
Natacha Crooks *UC Berkeley*
Shaddi Hasan *Virginia Tech*
Christian Maciocco *Intel*
Sylvia Ratnasamy *UC Berkeley*
Scott Shenker *UC Berkeley & ICSI*

Abstract

Cellular operators today know both the identity and location of their mobile subscribers and hence can easily profile users based on this information. Given this status quo, we aim to design a cellular architecture that protects the location privacy of users from their cellular providers. The fundamental challenge in this is reconciling privacy with an operator's need to provide services based on a user's identity (e.g., post-pay, QoS and service classes, lawful intercept, emergency services, forensics).

We present LOCA, a novel cellular design that, for the first time, provides location privacy to users without compromising on identity-based services. LOCA is applicable to emerging MVNO-based cellular architectures in which a virtual operator acts as a broker between users and infrastructure operators. Using a combination of formal analysis, simulation, prototype implementation, and wide-area experiments, we show that LOCA provides provable privacy guarantees and scales to realistic deployment figures.

1 Introduction

Providing users with *location* privacy is an important part of the larger challenge of online privacy. Unfortunately, today's cellular architecture offers little location privacy: network operators know the identity of a user and the geographic location of the access point to which that user connects and hence can trivially track a user's location in time. There is mounting concern over this situation as cellular providers are reported to routinely share their users' location profiles [28, 29, 62, 66, 105]. Moreover, 5G is likely to require smaller cell sizes [19] thus exposing much finer-grained location information and exacerbating the privacy problem.

Hiding a user's location from their network operator is challenging because connecting to an access point fundamentally reveals the user's location. One approach to improving privacy is to hide the user's *identity* from the network operator using so-called "blindly signed tokens" [23, 78, 86]. However, as discussed in §3, this approach comes at the cost of preventing network operators from providing *identity-based services*. These are services whose correct execution depends on the user's identity, such as post-pay [22], QoS prioritization [1] and lawful interception [3]. Such services are an essential part of today's networks and hence it is unlikely that operators can/will abandon them in exchange for improved user privacy. Thus, our question is whether we can enable location privacy

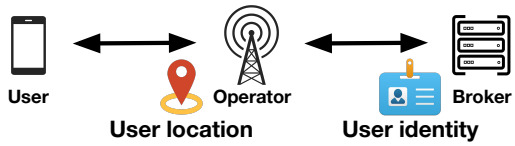


Figure 1: LOCA's overall architecture.

(*i.e.,* ensuring that network operators cannot easily track or infer a user's location) *without* compromising on identity-based services.

Privacy and identity-based services might seem to be fundamentally at odds. However, we see a way forward via mobile *virtual* network operators (MVNOs) such as Google Fi and Cricket [30, 49]. MVNOs are service providers that do not own radio infrastructure but instead provide user-facing services (sales, billing, *etc.*) while relying on business agreements with some number of traditional mobile network operators (MNOs) to provide the radio infrastructure. In this scenario, users pay MVNOs for service and MVNOs settle with MNOs on behalf of users. In other words, with MVNOs in the picture we can decouple infrastructure operation from user management and the MVNO acts as a broker between the user and the infrastructure operator.[1]

As shown in Fig. 1, our insight is that the existence of a broker between the user and operator enables us to reconcile privacy with identity-based services by strategically hiding different pieces of information from each party: the broker (*i.e.,* MVNO) knows the user's identity but not her location, while the operator (*i.e.,* MNO) knows the user's location but not her identity. With this arrangement, the broker can still tell the operator what identity-based services are to be applied to the user without revealing the user's identity, and the operator can implement the required services without knowing the identity of the user on whose behalf they are implemented.

However, hiding information in this manner is challenging for four reasons. First, in order to hide the user's identity from the operator, we must hide not just her identity but also her *trajectory* across multiple cell towers. This is because the operator could still infer the user's identity based on the sequence of towers she has visited, a form of privacy loss we refer to as *trajectory leakage* (§3.3).

Second, in order to hide the user's location from the broker, we must also hide the *identity of its operator* from the broker. This is because the locations of an operator's cell

[1] In this paper, we use the terms MVNO and broker interchangeably; we do the same with the terms MNO and operator.

tower deployments are public knowledge and hence can reveal a user's location [81]. The emergence of operators with small footprints, such as private and enterprise 5G networks, underscores the importance of this [13, 38, 52, 97].

The last two challenges arise because of this need to hide the identity of the operator from the broker. Brokers will always want to ensure that only authorized operators service their users. Since our approach hides the operator's identity from the broker, we now need a solution that allows the broker to verify the legitimacy of an operator *without revealing the operator's identity*. Lastly, when it comes time to settle payments, the operator should be able to claim payment from the broker *without* revealing what users it has served (since doing so would otherwise reveal user locations).

We design a privacy-preserving protocol that addresses the challenges above. Our contribution lies in developing new techniques (*e.g.,* aggregate claims) and synthesizing them with existing ones (*e.g.,* blind signatures, zero-knowledge proofs) into an end-to-end **Location-Oblivious Cellular Architecture (LOCA)**. To our knowledge, LOCA is the first system to enable location privacy for users while also supporting a provider's operational goals such as usage-based billing, QoS and service levels, lawful intercept, and so forth.

We evaluate the privacy and scalability of our protocol through formal analysis, simulation, prototype implementation, and wide-area experiments. We recognize that LOCA does introduce certain complexity and system overheads. However, our evaluation shows that these overheads are modest and within reach of what can be practically supported today. An important part of our contribution is thus in exposing the architectural complexity and performance tradeoffs that might be necessary to achieve our privacy goals.

Our work is based on certain assumptions about user and operator incentives. We assume that privacy concerns will influence some users in their selection of providers which will incentivize some operators to adopt the proposed techniques.[2] In addition, a growing number of jurisdictions have enacted policies that require providers to protect user privacy and, as discussed in §3, our architecture makes it easier for a provider to ensure compliance with these legal requirements. We do not assume that this motivation will apply to all users or operators: since our architecture can co-exist with the existing cellular infrastructure, we envision it will be applied to (by) the subset of users (providers) that are motivated by location privacy.

Finally, we recognize that there are many ways in which a user's location may be revealed through their online activities (*e.g.,* posting timestamped photos). We do not claim to prevent all forms of location leakage. Our focus is only on preventing the leakage of location information that today occurs every time a user connects to the cellular network.

In summary, the contributions of this paper are: (1) a new approach to preserve user location privacy while providing identity-based services; (2) the detailed design and implementation of a protocol (LOCA) based on this approach, and an evaluation of its performance and scalability; and (3) a formal analysis of the privacy provided by LOCA. Looking forward, we view LOCA as a first step towards privacy-preserving cellular infrastructure with room for improvement along multiple dimensions. We discuss these limitations extensively in the paper to motivate efforts on addressing these issues.

[2] Such market dynamics are already emerging in other contexts such as the smartphone market [10, 57, 85].

2 Background

The cellular ecosystem: MNOs and MVNOs Traditionally, the two main participants in a cellular network are the user with her device (called User Equipment, or UE) and the Mobile Network Operator (MNO). The MNO owns and operates cellular infrastructure and also provides user support services such as sales, billing and customer care. The user typically enters into a contractual agreement with one MNO which serves as her "home" provider. The user then consumes cellular services from her home provider or visited MNOs that her home provider has roaming agreements with.

In recent years, we've seen the rise of Mobile *Virtual* Network Operators (MVNOs). MVNOs are service providers that do not own radio infrastructure, but instead provide user-facing services (sales, billing, *etc.*), often focusing on serving specific underserved market segments [72, 91], while relying on business agreements with some number of MNOs to provide use of their radio infrastructure. In other words, the MVNO acts as a *broker* between the user and the infrastructure operator. In this scenario, the user contracts with an MVNO, and the MVNO in turn contracts with MNOs. Two well-known MVNOs in the US are Google Fi [49] and Cricket [30]. MVNOs can be involved in cellular operations to varying degrees, ranging from fully offloading to MNOs to operating their own core networks.

Identity-based services: These are services whose correct execution depends on the user's identity. An example of such services is lawful interception, a function that allows law enforcement agencies to selectively wiretap individual users [3, 4, 39]. In most countries, operators are legally required to support lawful interception. Additional examples of identity-based services include: (i) post-pay, which relies on identity-based accounting to charge a user based on her service consumption; (ii) QoS prioritization, where the network's treatment of a user's traffic depends on details of the user's subscription plan and past usage; (iii) deep packet inspection (DPI), where traffic is filtered based on the user's identity for purposes such as parental controls.

Location privacy in cellular networks: Location privacy, as defined in [17], is "the ability to prevent other parties from learning one's current or past locations". In the cellular context, this means that neither MVNOs nor MNOs should be able to learn a particular user's current or past locations. The exception is when location information must be revealed for legal purposes like emergency services and forensics.

3 Approach and Design Rationale

In this section, we briefly discuss the goals and assumptions that motivate LOCA's approach.

3.1 System Assumptions and Threat Model

System model: LOCA assumes a broker-centric architecture like today's MVNOs. This architecture involves three entities: (i) users, (ii) brokers, and (iii) operators. Operators own and operate cellular infrastructure. Brokers act as intermediaries between users and operators: a user subscribes to services from her broker, and the broker represents the user to operators, including handling settlements with each. LOCA requires brokers to authenticate their users.[3] The user need not be aware of the specific operator her device is attached to.

Threat model: We adopt a common threat model among privacy preserving systems that seek to prevent inadvertent information leakage between participants [25, 34, 54, 61, 79]. We assume brokers and operators are *semi-honest* (*i.e.,* honest-but-curious) and *non-colluding*: they follow the protocol but will attempt to extract user location information from the protocol execution, and that brokers and operators do not collude. We also assume that operators may attempt to *overbill* brokers by lying about session usages or what users they serve[4]. Attacks based on out-of-protocol information or collusion are out of scope but discussed in §5.

Incentives: One might ask why brokers and operators would implement the changes we propose. We believe that adopting our system is beneficial to them for both financial and legal reasons: as users are becoming more privacy-conscious [50, 74, 104], brokers that offer an opt-in location-oblivious service will be more attractive to customers. Second, doing so may soon become mandatory: regulations like GDPR recommend the privacy-by-design approach, which continues to place increasingly strong requirements on manipulating PII [16, 98, 107, 108]. By implementing a design such as ours, brokers and operators reduce their risk of inadvertently infringing privacy regulations. We explicitly assume that these benefits will outweigh the benefits of selling location data or implementing ad-hoc approaches to enforcing regulations, and thus we focus on the technical feasibility of a location-oblivious cellular architecture that also supports operational goals like usage-based billing and customized service levels.

3.2 Goals

Consider a user U, operator O, and broker B. We say that U's location privacy is violated when O and/or B know both U's identity and location. Today's cellular protocol trivially reveals both U's identity and her location. By protocol we mean the messages – their syntax and semantics – exchanged between U, B, and O as defined by the standard. Today, protocol messages carry U's identity, and the identity of the tower that U attaches to reveals U's location. Hence simply implementing the protocol allows an operator to track U's location with no special effort. In contrast, we are interested in modifying the existing cellular protocol standard to protect user privacy.

Arch	Operator (O)	Broker (B)	ID-based SVC
Today	UID, Location, Trajectory	UID, OID	Full
PGPP	Location, Trajectory	OID	Partial
LOCA	Location	UID	Full

Table 1: Comparison of today's MVNO architecture, PGPP and LOCA in terms of information revealed to participants and support for identity-based services (ID-based SVC); U/OID: U/O's identity.

3.3 Approach

In research, the state of the art is the recently proposed PGPP protocol [86] which tries to provide location privacy by *hiding* U's identity from O and B. In PGPP, users are identified by a "blindly signed token" [23, 78] which they obtain during a registration phase prior to consuming service.[5] I.e., a user prepays for a certain quota of service (*e.g.,* some number of minutes of connectivity at a specified data rate) and in return obtains a blindly-signed token. When connecting to the network, the user presents this token via which the broker can authenticate the user without learning her identity.

To our knowledge, PGPP is the first system that tries to provide location privacy for cellular users. However, as we detail in §8, PGPP faces two drawbacks. First, PGPP does not easily allow operators to support identity-based services, which are widely deployed in today's networks. Second, a user's *trajectory* across towers is still visible to operators and hence the protocol is vulnerable to "trajectory-based location leakage" in which the operator can learn the user's identity by correlating her trajectory with other out-of-band information.[6] In designing LOCA, we wished to avoid these limitations which, as we will see, leads to an altogether different approach.

In summary, our goal in LOCA is to design a cellular protocol that protects the location privacy of users by achieving the following properties: no party in the protocol (broker, or operator) should simultaneously know both the identity and the location of a user; the protocol should also not reveal the user's trajectory to either broker or operator. Finally, the protocol should support identity-based services including post-pay and lawful intercept. In this work, we propose LOCA, a new cellular protocol that achieves these stronger privacy guarantees while supporting identity-based services.

We briefly comment on the scope and limitations of LOCA

[3] For MVNOs who by default offload all cellular operations, they can still support LOCA users by deploying their own authentication servers.

[4] One might ask whether we need to protect against over-billing if the operator is semi-honest. The reason we do so is because, as we'll see, *once we have privacy*, it becomes much easier for an operator to overbill since the broker cannot tell which users were serviced by the operator and hence cannot check the operator's billing claims. Hence, an operator can follow the protocol and yet overbill with impunity. To avoid this, we assume operators may overbill and design our protocol to prevent this.

[5] Such a token is *blindly* signed by the broker who can later verify the signature without being able to link it back to the original signing request.

[6] For example, consider a user that regularly travels between their home and office location: the operator could narrow down the identity of the user by correlating this trajectory with residential information in billing records.

as presented in this paper. Our goal is to safeguard users' location privacy at the *protocol* layer. This raises the bar relative to today's protocols but isn't sufficient to safeguard against violations that might occur outside the protocol, at other layers. For example: at the application layer, a user's identity might be revealed by inspecting their packets [14, 88], or physical-layer characteristics (*e.g.,* signal patterns) might be exploited to track a specific device [33, 47]. Such attacks are possible but (to our knowledge) not exploited today. However, if cellular protocols evolve to protect privacy, such app/physical layer leakages could become a more important issue. Fortunately, the research literature provides solutions to such attacks [43, 56, 60, 103, 110, 114] that we believe can coexist with protocol-layer solutions like LOCA. We elaborate on this in §5.2 but leave an in-depth exploration to future work.

There is an obvious tension between guaranteeing location privacy and offering identity-based services: connecting to a cellular tower fundamentally reveals a user's location, while customizing service to a user requires knowing the user's identity. Our insight is that we can extend broker-centric architectures to create a situation in which the broker knows the user's identity but not their location, while the operator knows the user's location but not their identity; neither broker nor operator knows the user's trajectory.

How do we achieve this? First, to hide U's location from B, we hide the identity and location of the *operator* O from B. Recall that U attaches to the network (and hence to B) via O's infrastructure and hence, if B cannot tell where O is located, then it cannot tell where U is located either. Hiding O's location is not sufficient: we must also hide O's identity from B, as knowing O's identity might be sufficient to narrow down O's location (and hence U's location). An operator's tower locations are public knowledge and, moreover, we're seeing an increasing deployment of small-scale cellular networks due to the emergence of private and enterprise 5G networks, as well as various forms of community networks [13, 38, 52, 97].

As we will describe in §4, we hide O's identity from B by having O obtain an unlinkable token from B during an offline registration process.[7] O later uses this token (denoted $\hat{O}$) as its identifier when interacting with B. By the properties of blind signatures, B can verify that $\hat{O}$ is a pre-authorized operator but cannot link $\hat{O}$ to O. In addition, O hides its IP address from B by using anonymous communication solutions.

The above suffices to hide U's location from B. The other half of our arrangement is to hide U's identity from O. This is easily achieved since O does not need to know U's identity to service U; since B knows U's identity, B can tell O what services are required (rate limits, filtering rules, *etc.*) thus enabling identity-based services without revealing U's identity. Thus, U simply uses a temporary pseudonym (denoted $\hat{U}$) in her interactions with O. Finally, by periodically changing U's temporary pseudonym and randomizing attachment timing, we limit O's ability to track a particular user's trajectory.

[7] The use of such a token is similar to PGPP but used by O instead of U which we will see leads to a very different set of considerations.

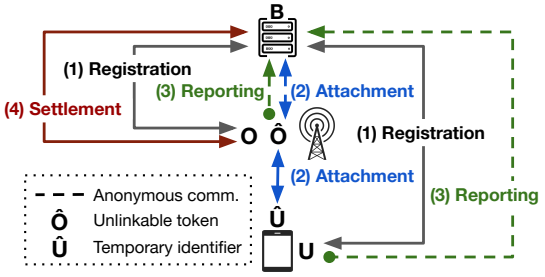


Figure 2: An overview of LOCA's protocols.

As summarized in Table 1, the above approach offers U location privacy while still supporting identity-based services. However it gives rise to a new challenge: how does O receive payment for its services to U? In today's architecture, B directly settles with O based on the service that U received. We wish to preserve this direct billing system between O and B. Yet, our protocol intentionally hides O's identity from B. To address this issue, we devise a solution that allows O to reveal its true identity *only* when claiming payment from B. Our solution leverages zk-proof techniques to design a novel *aggregate claiming* procedure via which (i) O claims payment for an aggregate of the user sessions it has serviced, and (ii) B can verify the correctness of O's claim without revealing the identity of the users that O serviced.

4 Design

At a high level, the process of obtaining cellular services can be broken down into four phases or steps: (i) *registration*, during which the various parties (U, B, O) enter into pairwise contractual relationships: U signs up with B for service, and B with O as an operator for its users; (ii) *attachment* involves the protocol by which U discovers and connects to a tower in O's infrastructure, (iii) *mobility* involves the handover protocols via which U is migrated from one tower to another as needed, and (iv) *settlement* refers to the norms and processes via which B pays O for the service that O has provided B's users.

Of the above, *attachment* and *mobility* are defined by today's 3GPP standard while *registration* and *settlement* are out-of-band processes. Our goal is to implement LOCA with minimal disruptions to today's protocols, and without involving any new entities in the registration or settlements process.

Next, we describe LOCA's operation in these phases, an overview of which is given in Fig. 2.We briefly summarize how each phase is typically implemented in today's networks and then present the changes that LOCA introduces. Finally, we elaborate on how identity-based services work in LOCA.

4.1 Registration

Today: In today's networks, when U signs up with a broker B, they exchange shared secret keys (*SSK*s) that will be used for mutual authentication during the attachment process. In 5G, B also shares its public key (PK_B) with U so that U can encrypt her identity in later attachment requests.

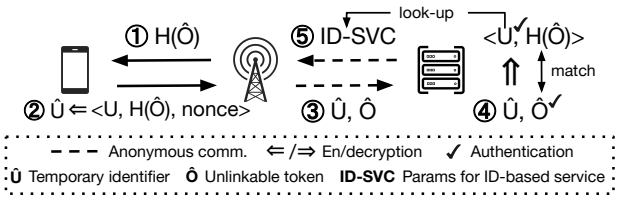


Figure 3: LOCA's attachment procedure.

LOCA: With LOCA, B and U continue to exchange PK_B and *SSK*. Like today, these keys will be used for mutual authentication between U and B (§4.2) and to hide U's identity from O. The main change LOCA introduces is in the registration process between B and O. When B and O sign up with each other, LOCA requires that they participate in a blind signature protocol [23, 78] as a result of which O obtains unlinkable tokens (denoted as $\hat{O}$) that are blind-signed by B. When $\hat{O}$ is later presented to B, the blinding process ensures that B can verify the signature but cannot link $\hat{O}$ to O. Thus blind tokens allow B to authenticate O without learning O's identity. LOCA uses a standard blind-signing protocol [23] (summarized in Appendix A). In addition to blind tokens, B and O also exchange a shared hash function H that will be used in our attachment and settlement processes as described later.

4.2 Attachment

Today: Attachment today involves three main steps. First, O broadcasts its identity on the radio control channel that U listens on to discover O. Next, after discovering O, U sends an *attachment request* to O who forwards the request to B for authentication. In 5G, U uses an encrypted identifier (termed SUCI [2]) in this attachment request. Finally, once U has been authenticated, B responds to O authorizing service. B's response includes U's permanent identifier (termed SUPI [5]).

Thus today's attachment process reveals O's identity to U in the first step. In the second step, B learns O's identity (and hence U's location) from both the contents of the attachment request and the act of receiving it from O (which reveals O's IP address). Finally, O learns U's identity via the authorization response it receives from B.[8] Thus today's attachment reveals U's identity and location to both B and O.

LOCA: We describe LOCA's attachment process with an emphasis on how we prevent (i) B from learning O's identity and (ii) O from learning U's identity. As mentioned in §3.3, we achieve the former by having O interact with B as $\hat{O}$ (O's unlinkable tokens) via anonymous communication channels. LOCA achieves (ii) by encrypting U's identity (with PK_B) and never exposing it outside of B. As shown in Fig. 3, LOCA's attachment process consists of the following five steps.

(i) Operator discovery. Instead of its actual identity, O broadcasts the hash of its token (*i.e.,* H($\hat{O}$)) on the control channel.

(ii) User preparation. U sends an attachment request to O (formatted as a NAS message [7]). This request includes B's identity, U's identity (IMSI) plus a nonce, and H($\hat{O}$). The last two – (IMSI+nonce) and H($\hat{O}$) – are encrypted by B's public key and serve as a temporary identifier for U which we denote as $\hat{U}$. We assume that B has a large user group so that its identity leaks little information on U's identity. The nonce ensures that $\hat{U}$ is different every time U attaches to the same O which helps prevent O from tracking U's trajectory (§4.3).

(iii) Operator preparation. On receiving U's attachment request, O forwards the request to B over an anonymous communication channel and uses its unlinkable token $\hat{O}$ to identify itself. Typical solutions for anonymous communication are Tor [99] and VPN [80] with different performance/security trade-offs, which we will discuss in §6.3. This anonymous channel can be set up offline, prior to attachment, whenever O changes token $\hat{O}$. Thus B does not see O's true identity nor the IP address from which $\hat{O}$ sends the request. The latter is necessary as several studies have shown that IP addresses can often be geo-located with high accuracy [26].

(iv) Broker authorization. On receiving the attachment request, B first verifies the $\hat{O}$ token thus ensuring that the request comes from an operator that B has previously authorized during the registration phase. Next B decrypts the request, and authenticates U via today's challenge-response protocol based on the shared secret key *SSK* [6]. In addition, B verifies that $\hat{O}$ is indeed the operator to which U wants to attach; B can verify this by validating H($\hat{O}$) (using the shared hash function established when O registered with B) and thus prevents replay or hijacking attacks. Once B has authenticated and verified the request, it looks up the parameters associated with U's service plan (as today): *e.g.,* rate limits, QoS parameters, whether to intercept U's traffic, and so forth. B then crafts a response authorizing the attachment (including the proper service parameters, security parameters that allow U to authenticate the network, etc), signs it, and returns it to $\hat{O}$.

(v) Access attachment. B's response authorizes $\hat{O}$ to service U as per the parameters from B. Beyond this point, $\hat{O}$ (*i.e.,* O) serves U as in today's networks. We elaborate on how O provides identity-based services to U in §4.4. Note that O can still perform functions like establishing radio bearers that require binding U's identifier to temporary identifiers like GUTI and RNTI; O simply uses $\hat{U}$ instead of U.

4.3 Mobility

Today: In current networks, mobility is implemented via a handover process, where O initiates U's migrations by directing U to switch from a tower T1 to another T2. This approach ensures a seamless mobility experience for U because U's IP address remains unchanged after the migration. However, as O initiates U's migrations, O trivially observes U's trajectory across handovers, jeopardizing U's location privacy.

LOCA: Trajectory leakages are inevitable if O fully controls U's mobility like today: although LOCA already hides U's identity from O during attachments, O can still track $\hat{U}$'s trajectory and use that to infer U's identity, making LOCA vulnerable to trajectory analysis. To mitigate this fundamental

[8] Prior to 5G, U's permanent identifier (IMSI) was included in the initial attachment request, allowing O to directly discover U's identity. Since 5G, U's attachment request uses an encrypted temporary identifier over the air to defend against IMSI catchers [93]. Nonetheless, O still learns U's permanent (SUPI) identifier from B's authorization response (step 3).

issue, we leverage a user-driven mobility approach proposed in [77]. In this approach, U initiates migrations across towers by simply detaching from T1 and then attaching to T2. U then relies on modern transport protocols like MPTCP [82] and QUIC [71] to maintain connections despite changing IP addresses. Prior work has shown that this user-driven approach does not degrade service even when reattaching on a *per-tower* basis [77]. LOCA adopts and extends this approach to minimize trajectory leakages with two techniques: (i) periodic reattachment and (ii) randomized attachment timings.

First, U will detach and reattach periodically (not at every tower) with a new temporary identifier. Thus, O cannot trivially track U across new sessions based on U's identifiers. The reattachment frequency is a configurable parameter that bounds the length of U's trajectory that is visible to O where length might be measured in time (*e.g.,* valuable for a mostly stationary user), in towers, or some combination thereof.

Even with periodic reattachment, O may still attempt to infer U's trajectory by doing a timing analysis over her detach and attach events. In particular, such analysis would be effective in a naive implementation that uses a fixed interval between when U detaches from T1 and subsequently attaches to T2. To address this issue, we have U wait for a randomized but bounded duration of time before issuing her attachment. When possible, we can also leverage make-before-break attachments[9] in which U may attach to T2 *before* detaching from T1 thus increasing the time window over which U can randomize their attach/detach events which makes inference harder. Together with periodic reattachment, this randomization of U's attachment times limits O's ability to correctly infer U's trajectory, because U's (re)attachments are obfuscated by the periodic (re)attachments from other nearby users.

We recognize that user-driven mobility introduces some complexity as well as dependencies on newer transport stacks, however this tradeoff is fundamentally necessary if we are to prevent trajectory leakages, and supporting these techniques incurs a minimal impact on the user's performance (§6.3). As we will detail in §5.1.3, the obfuscation effect of our approach depends on the specific configurations, *i.e.,* reattachment frequency and attachment time window; as well as the deployment scenarios, *i.e.,* the number of nearby users and the length of U's trajectory. Overall, under realistic deployment scenarios and configurations, the probability that O can correctly infer U's trajectory is negligible.

4.4 Identity-based Services

LOCA ensures that operators and brokers can continue to provide critical identity-based services, including allowing law enforcement agencies to locate specific users when required.

The key reason LOCA can support identity-based services is that brokers continue to know the identity of their users. This enables B and O to collaborate on identity-based services. For instance, during attachment, B can select the service level associated with U's plan and indicate that to O in its authorization response – *e.g.,* via the QoS Class Identifier (QCI) parameter [109]. O then simply enforces the QCI for the duration of its session with $\hat{U}$ without knowing U's true identity.

To realize services such as lawful interception, law enforcement agencies work with B and O. As today, O runs a lawful interception (LI) system — *e.g.,* installing an interception gateway [96]. A law enforcement agency notifies B of the user whose communication it wants to intercept. B passes on this notification to O during the attachment process, and then O's LI systems report the required information to the agency.

Emergency services (*e.g.,* 911 calls) work in a similar manner. A law enforcement agency knows U's identity and needs to learn U's location. The agency reaches out to B; B looks up U's current temporary identifier $\hat{U}$, and asks $\hat{O}$ (via their anonymous communication channel) to reveal $\hat{U}$'s location to law enforcement. Thus, the agency can collect U's current location without violating LOCA's privacy guarantees (*i.e.,* O does not know U's identity while B does not know O's identity or location). The same approach can be used to recover U's past locations based on the records logged at B and O.

4.5 Settlements

Today: In today's MVNO networks, B pays O based on U's service parameters and the resources consumed, as reported by O to B.While differing in the details, existing settlement processes all require that B knows which users/sessions were serviced by O, thus potentially violating user location privacy.

LOCA: To settle O's payments while preserving U's location privacy, LOCA's settlement process contains two phases: a *reporting* phase, where U and O report session usage to B; and a *claiming* phase, where O claims settlement from B.

Reporting phase: In LOCA, we define a *session* as the user-operator association that starts when U completes the attachment process with $\hat{O}$ and ends when U detaches from the same. At some point after a session ends, U and $\hat{O}$ independently send traffic reports to B. Note that O continues to hide its identity and location when sending its report to B. U reveals its identity to B but also sends its reports over an anonymous channel because its IP address can reveal its whereabouts. The traffic report from U lists the sessions in which U participated; $\hat{O}$ does the same for its sessions. Each entry in the list contains a session identifier (SID), usage metrics (*e.g.,* bytes, duration), and QoS metrics (*e.g.,* packet loss rate). In addition, O appends a nonce to each session in its report. These nonces are generated from the shared hash function H known to both O and B, and taking secret inputs that are only known to O. We call these inputs "embedded secrets", and as we will see, O later uses these secrets to claim its settlement from B.

B then compares the reports from U and $\hat{O}$, generates bills for U and publishes a *session table* to start the claiming phase. The table includes the usage calculated based on the reports from $\hat{O}$ and U, for all sessions during the last billing cycle.

[9] The support for make-before-break, so-called dual active protocol stack (DAPS) handovers has been introduced in 5G 3GPP specifications [8, 45, 95].

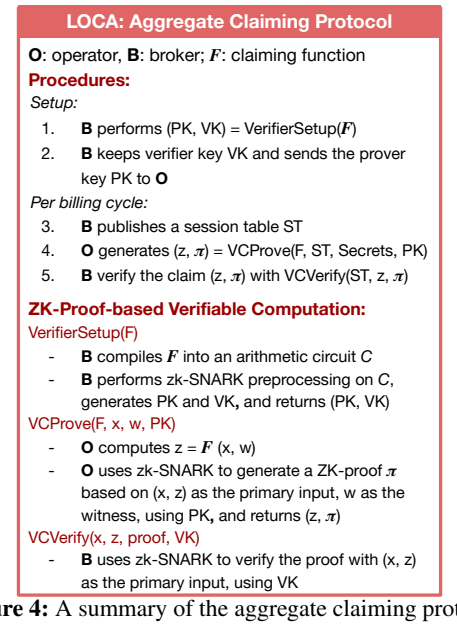


Figure 4: A summary of the aggregate claiming protocol.

When generating statistics in the session table, B can consider factors other than reported usages such as QoS metrics.

Claiming phase: Every billing cycle, O reveals its identity and claims settlement from B but does so without revealing which sessions O has serviced. To achieve this, we must solve three problems: (i) ***No over-claims.*** How does B verify that O is claiming only the sessions O actually serviced? (ii) ***No mis-claims.*** How do we ensure that O can claim the sessions but no one else? (iii) ***Session oblivious.*** How does O claim settlement without revealing to B which sessions it is claiming? We combine zero-knowledge proofs with the above mentioned "embedded secrets" to address (i) and (ii); and "aggregate" claims to address (iii).

Embedded secrets serve as the basis for O proving its session ownership to B. However, naively having O reveal its secrets fails the session oblivious requirement because B now knows what users O has serviced. This leads to our aggregate claiming protocol that fulfills all three requirements:

Aggregate claiming with ZK-proof: First, we observe that in order to generate O's payments, B does not need to know *individual* session ownership; instead, it only needs to know the session ownership in *aggregation*, *i.e.,* the aggregate usages for payments for a specific O. Based on this insight, our *aggregate claiming* mechanism works as follows: the claiming begins with B publishing a session table readable to all Os. O then reveals its identity and claims its payment from B:

Intuitively, O's claim takes the form: "I have sessions that add up to X bytes." Because the number of different sessions that could add up to X is large, it is difficult for B to infer whether an individual session is part of O's claim or not, thus obfuscating the session ownership. In §5.1.2, we show that the expected number of session combinations that add up to the same X grows *exponentially* w.r.t. the total number of sessions in the table via both theoretical and empirical analysis.

Note that this naive aggregate claiming suffices if we assume O will not overbill B. However, it is important to realize that without additional mechanisms (like the zk-proof that follows), O can more easily overbill B without being detected in LOCA than in today's (non-privacy preserving) architecture simply because B does not know what users O serves.

Hence, since naive aggregate claiming allows O to overbill, we extend our solution such that O can *prove* its claim by showing that O knows the embedded secrets corresponding to its claim. For this, we leverage *proof-based verifiable computation* [102], a cryptographic tool that uses zero-knowledge proof to enable one party to prove to another that it has run a computation $z = f(x, w)$, where f is the function, x is the public input, w is the prover's private input and z is the output, without revealing any information about w. Proof-based verifiable computation systems have two components: (i) a zk-SNARK backend [84] that proves and verifies satisfiability of *arithmetic circuits*, and (ii) a compiler frontend that translates program executions to arithmetic circuits. Such an arithmetic circuit is also referred to as "a set of *constraints*".

Fig. 4 describes LOCA's aggregate claiming protocol. First, B performs *VerifierSetup*, where B compiles a claiming function F into an arithmetic circuit, and uses zk-SNARK to preprocess the circuit and generate prover key PK and verifier key VK. This verifier setup step needs to be performed only once, after which B keeps VK and sends PK to each participating O. The claiming function F takes two inputs: a session table with at most N sessions as the public input, and a set of (at most K) secrets as the private inputs. F computes the hashes of the provided secrets, iterates all the sessions in the session table, adds a session's usage to the aggregate usage if one of the precomputed hashes matches the nonce of that session, and finally returns the aggregate usage.

Next, once per billing cycle, each O performs *VCProve*, which involves two steps: (i) O executes the claiming function F with the session table and its embedded secrets, which returns the aggregate usage z for O's sessions. (ii) O passes to zk-SNARK the session table, its secrets, the computed aggregate z and the prover key PK, to generate a zero-knowledge proof π, which allows O to prove to B that it has secrets for sessions that add up to z, without leaking any information about individual session ownership. O then sends a *claim* including the aggregate usage z and proof π to B.

For each O's claim, B performs *VCVerify*, where B uses zk-SNARK to verify the proof π with the session table, the claimed aggregate z and the verifier key VK. If the verification passes, given the soundness property of the zk-SNARK proof system [84], B can confidently approve O's claim and generate O's payment according to the claimed aggregate usage and other factors such as O's reputation. The duration of a billing cycle is configurable: longer cycles lead to larger session tables, which in turn indicates stronger privacy protections (§5.1.2) at the cost of more expensive operations (§6.2).

Session group: The design presented above assumes a single session per token, which may not scale to large deployments:

O generates a proof every billing cycle, and proving with zk-SNARK is expensive [111]. In our setup, as we will show in §6.2, the time complexity to prove a circuit for the claiming function F is $O(K*N)$, where K is the maximum number of sessions O can claim and N the total number of sessions in the session table. Such proving time would be prohibitively long when there are a large number of sessions to claim.

To address this scalability challenge, we introduce the notion of a *session group*, which includes all the sessions that are associated with the same unlinkable token. By grouping multiple sessions into a single session group, we can reduce the number of entries in the session table. To support session groups, we made the following extensions to our protocol:

- **Attachment:** We allow O to use a single token and the corresponding anonymous communication channel for multiple sessions as the same session group.
- **Reporting:** We allow O to send a traffic report containing all the sessions of the session group.
- **Claiming:** We allow B to publish a session group table with one session group for each row. O claims session groups the same way as it claims sessions before.

The size of the session group is tunable in LOCA and determines how many sessions each token is used for. Tuning the group size allows LOCA to explicitly trade off between privacy and scalability: (i) a smaller session group is better for privacy, because it minimizes indirect location leakages (detailed in §5.3), which occur when a user of a session within a session group has her locations leaked, in which case users of other sessions within the group also suffer a privacy loss; (ii) larger session groups are desirable in terms of scalability of zk-SNARK, as it takes longer to actually generate a session group (with users' attachment), while proving cost remains the same, as N is the same, so zk-SNARK proving becomes relatively faster. Fortunately, modern zk-SNARK is fast enough that a balance between privacy and scalability can be achieved: as we will show in our evaluation (§6.2), LOCA can scale to large deployments with sufficiently small session groups and thus introduces only minimal privacy loss.

5 Privacy Analysis

Safeguarding location privacy requires fulfilling three properties: (i) O does not knows U's identity, (ii) B does not know U's location, and (iii) neither B nor O knows U's trajectories. To our knowledge, LOCA is the first protocol to meet these requirements. In this section, we analyze the conditions and assumptions under which LOCA meets these requirements. We show that LOCA achieves all three properties under the assumptions of our threat model which are that participants are semi-honest and do not collude (§5.1). We then briefly consider attacks beyond our threat model and show that LOCA offers substantial protection even when participants use out-of-protocol information (§5.2) or collude (§5.3).

5.1 Semi-honest and Non-colluding

We first analyze LOCA's privacy properties under our threat model of semi-honest and non-colluding participants (§3.1).

5.1.1 Hiding U's identity from O

LOCA hides U's identity from O. Specifically, U's identity is encrypted using B's public key. B is thus the only party that can decrypt and observe U's identity in plaintext. B also never exposes U's identity to O, even after U successfully attaches.

5.1.2 Hiding U's location from B

LOCA hides U's location by (i) hiding O's identity and location when O interacts with B on behalf of U and (ii) hiding which users were serviced by O when O reveals its identity to claim its settlement. Next, we show how LOCA achieves (i) via the security properties of existing cryptographic constructs (*i.e.,* anonymous communication and blind signature) and achieves (ii) via aggregate claiming; we establish the latter property via formal analysis and empirical simulations.

For (i), LOCA leverages anonymous communication such that two parties can communicate without revealing their identities to one another. Similarly, LOCA builds on a blind signature scheme that allows a participant to authenticate another without learning its identity. Taken together, these existing cryptographic constructs allow operators to register and report sessions to brokers without revealing their identities.

Discussing the security of aggregate claiming requires more care. We break this process down into two halves (i) the security of the claiming mechanism itself, (ii) the information leaked by revealing the aggregate value to B. The former follows directly from the security of our zero-knowledge proof construction. We do not discuss this further. Instead, we focus on the impact of B learning the aggregate value of the claimed sessions. Specifically, we show that B has an exponentially small likelihood to correctly infer what sessions/users O has serviced based on the aggregate value.[10] Our core intuition is simple. Let us assume that for N session groups with a uniform distribution of session group usage from 1 to m, operators will claim the aggregate usage of K session groups, which sum to aggregate value S. The total number of possible session group combinations grows exponentially as a function of N. In contrast, the number of possible claimed values only grows linearly ($m*N$). In expectation, there will consequently be exponentially many possible session group combinations that could have summed to S. We formally prove that this result holds as long as the ratio between K (the number of session groups belonging to an operator) and the total number of session groups N falls within a specific range. We identify this range formally below, and show through simulation that these bounds can be further improved and are wide enough to support realistic deployment scenarios.

Theoretical proof: We formulate the aforementioned problem as follows. Consider arrays X and Y, one of size $N-K$,

[10] The general reasoning extends to when B analyzes multiple claims from different operators but we don't get into the details in this paper.

and one of size K, where each cell contains a value from 1 to m drawn from the discrete uniform distribution. Let S be the sum of all elements in Y. We derive a bound on the expected number of possible subsets of elements in X that sum to S.

Theorem 5.1. *Considering two independent arrays X and Y, consisting of $N-K$ and K iid random variables from $U\{1,m\}$, there exists $L(m),U(m)$ such that the expected number of subsets in X, whose sums are equal to the sum of Y, is exponential w.r.t N, if $L(m) \leq \frac{K}{N} \leq U(m)$. Note that $L(m),U(m)$ depend on m, and $0 < L(m) \leq U(m) < 1$, $\forall m \in Z_{>0}$*

The proof, at a high level, works by (i) deriving the closed-form distributions of the sum and the subset sum of an array of discrete uniform variables similar to prior theoretical work [20], (ii) expressing the expected number of matched subsets with these two closed-form distributions, and finally (iii) reducing to an exponential lower bound for the expression. More details of the proof can be found in the appendix.

The reductions in step (iii) are highly conservative. Hence the proven feasible range of ratio $[L(m),U(m)]$ is narrow, and the exponential bound is small. We confirm through simulation that this bound holds analytically for a significantly wider range and encompasses many real-world scenarios:

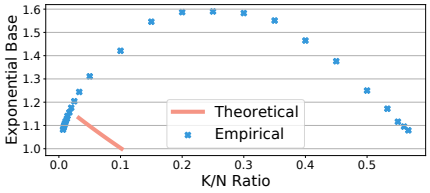


Figure 5: Exponential bounds for different K/N ratios with $m = 5$.

Empirical simulations: In these experiments, our goal is to understand within what range of ratio, the number of matched subsets grows exponentially w.r.t N. For each ratio K/N, we scale N while increasing K proportionally according to the ratio and estimate the expected number of matched subsets for the (K,N). More details about our simulation setup are in Appendix B. Now that we have estimates for multiple (K,N)'s of the ratio K/N, we fit the results with an exponential curve of N by performing linear fittings on the logs of the estimates:

$$R = a * b^N \rightarrow \log(R) = \log(b) * N + \log(a)$$

The slope of the fitted linear curve is thus the log of the exponential base. Our fitted linear curves closely match the logs of estimates with an adjusted R-squared value of over 0.99, which suggests a significant exponential relation between our estimates and N. Fig.5 shows the exponential bounds of different K/N ratios for uniform distribution with m=5. Compared with the theoretical results, the empirical results suggest much larger exponential bounds over a wide range of ratios: exponential base over 1.1 for ratios from $1/150$ to over $1/2$. We observe similar behavior with other values of m and with other non-uniform session group usage distributions.

5.1.3 Hiding U's trajectory

LOCA hides U's trajectory from O via (i) periodic reattachment and (ii) randomized attachment timing. The former prevents O from directly observing U's trajectory, and the later makes accurate timing-based trajectory inference infeasible:

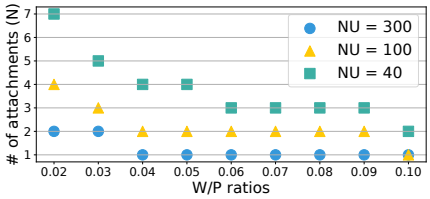


Figure 6: The longest trajectories beyond which the likelihood of correct inference is less than 1% for different NUs and W/Ps.

With periodic reattachments, O is unaware of which attachments belong to U and hence O can only *infer* U's trajectory by correlating between detachment and subsequent attachment. By randomizing attachment timing, any detachment that arises within a time window before and after (with make-before-break handovers) an attachment is equally likely to correlate with that attachment. We call this set of detachments "candidate detachments", and since all users periodically reattach, there is a *lower bound* on the number of candidate detachments. Lastly, to recover U's trajectory, O has to select the correct detachments for all of U's attachments along the trajectory, which becomes *exponentially* harder for longer trajectories. Modelling all these factors, we can analyze the difficulty of trajectory inference in LOCA: denoting time window as W, the reattachment period as P, the number of nearby users as NU, the number of candidate detachments as ND, the number of attachments in U's trajectory as N, we can derive the likelihood of O correctly inferring the trajectory $Prob$:

$$ND \geq 1 + NU * \frac{W}{P}, \quad Prob \approx (\frac{1}{ND})^N$$

This formulation tells us why accurate trajectory inference is infeasible: (i) since $Prob$ decays exponentially w.r.t N, even with a ND of 2 (only one alternative candidate detachment), O has a less than 1% likelihood of inferring a trajectory with more than 6 attachments; (ii) The ratio between the time window and re-attachment period (*i.e.,* $\frac{W}{P}$) is configurable, and a larger ratio increases ND and thus the inference difficulty. Fig.6 shows the longest trajectories that O can infer with a likelihood larger than 1% for different NUs and $\frac{W}{P}$s. For $\frac{W}{P}$ larger than 0.03, O is unable to infer long trajectories ($N > 4$), even if the number of nearby users is small ($NU = 40$).

We've shown that LOCA safeguards user location privacy at the protocol layer and under the assumptions of our threat model. We believe this raises the bar relative to the status quo however, as discussed earlier, LOCA would still be vulnerable to attacks that exploit either: (i) out-of-protocol information or (ii) information from other participants via collusion. We will next discuss such attacks, their impact, and potential mitigation strategies, but leave an in-depth study to future work.

5.2 With out-of-protocol information

Next, we show that (i) as a protocol-layer solution, LOCA does not prevent attacks based on out-of-protocol information, (ii) the impact of these attacks on LOCA is minimal and, (iii) mitigation strategies for these attacks can coexist with LOCA.

Attacks: B and O can compromise U's location privacy by exploiting out-of-protocol information. Here we enumerate some attacks that violate each of the three privacy properties: (i) If O has access to a resident directory near its cell towers, it can nail U down to a smaller user group. O might also learn U's identity by inspecting U's data traffic. (ii) If B is capable of network monitoring, it might learn O's identity by conducting a traffic analysis, where it observes traffic at each operator and correlates that with incoming traffic it receives. (iii) O might track U's trajectory by profiling U's physical-layer characteristics, such as its signal patterns and strengths.
Impact: LOCA's design limits the impacts of out-of-protocol attacks on user's location privacy: (i) Attacks that allow O to uncover $\hat{U}$'s identity only incur *per-hop* leakages: U's identity remains unknown to O when she reattaches with a different $\hat{U}$. (ii) Attacks that allow B to uncover $\hat{O}$'s identity only incur *per-token* leakages: O's identity remains unknown to B when O switches tokens. This means that locations of users who are served by O with a different token from the revealed one remain unknown to B. (iii) Lastly, inter-operator attachments can minimize impacts of attacks that allow O to track U's trajectory. Firstly, instead of having her entire trajectories leaked, U suffers only *per-operator* leakages. Secondly, as U moves in and out of O, it is challenging for O to link all of U's trajectories within its footprint, because O is unaware of U's locations when U connects to other operators.
Mitigation: LOCA can coexist with countermeasures designed for different out-of-protocol information. For instance, for attacks based on traffic characteristics, end-to-end encryptions of U's traffic can help counter packet inspections by O; and communication systems that are robust to traffic analysis like Vuvuzela [101] could be adopted for communications between O and B. For attacks based on physical-layer signals, one could use defense mechanisms such as randomizing transmission coefficients [89] and injecting artificial noises [56].

5.3 With collusion

In the following, we show that (i) there are forms of collusion that lead to violations of user location privacy, and (ii) except for direct collusion between brokers and operators that serve the user, other forms of collusion only incur minimal leakages.
Attacks: Collusion between B and O reveals both U's identity and location. Note that this is the case for any MVNO-based architecture where B knows U's identity (for offering identity-based services) and O knows U's location (as it provides connectivity). Therefore, we focus on showing what other forms of collusion also impair user location privacy. For O, colluding with participants other than B does not provide it with extra information on U's identity or trajectories. For B, however, it can gain additional knowledge regarding U's location by colluding with (i) other users or (ii) other operators. The former is due to the use of session groups. Specifically, B knows that sessions in a session group belong to the same O, hence that users of these sessions have visited the same location at a similar time. Therefore, if some users who share session groups with U reveal their locations to B, then B knows U's location via such collusion. We call these "indirect location leakages". The latter is due to operators sharing the session table in the claiming phase. Specifically, B now effectively has a "smaller" session table consisting of only sessions from non-colluding operators, which is detrimental to the privacy guarantee provided by aggregate claiming.
Impact & Mitigation: While brokers gain extra user location information via collusion with other users or operators, the actual impacts are minimal and can be further reduced with different mitigation strategies. First, the impact of indirect location leakages is bounded by the size of session groups, which in turn depends on how fast the zk-SNARK backend is. Fortunately, even with a single-core backend, aggregate claiming can scale to large deployments with a session group that lasts as little as 20 *s* (§6.2). One could adopt faster backends like distributed zk-SNARK [111] to further reduce the size of session groups and hence leakages. Secondly, since the obfuscation effect of aggregate claiming is *exponential* w.r.t. the size of the session table (§5.1.2), a smaller table still grants sufficient protections. One could use a longer billing period to ensure a large enough session table even with collusion.

6 Implementation and Evaluation

In this section, we present the implementation of our LOCA prototype (§6.1) and investigate the two key questions regarding the feasibility of LOCA: (i) can LOCA scale to realistic deployment sizes? and (ii) how much overhead does LOCA introduce compared to existing cellular protocols? We answer the first question by performing a scalability analysis of the privacy building blocks (§6.2); and the second by conducting a performance analysis with wide-area experiments (§6.3).

6.1 Implementation

We prototyped LOCA as an extension to the CellBricks system [21] which is itself built from open source cellular platforms (Magma [41] and srsLTE [92]). We extended the operator and broker modules with the following: (i) the token generation and verification procedures implemented with rsablind [32]; (ii) the anonymous communication channel between the operator and broker implemented with Tor-socks [51,99] and NordVPN [80] and (iii) the claiming procedure implemented with Pequin [83,102] that has a single-core libsnark [69] as the zk-SNARK backend. In total, our extension includes 478 LoC in C (for claiming), 144 LoC in Go (for unlinkable token), and 16 LoC shell scripts (for anonymous communication and various setup). We prototyped LOCA with these languages as they were used in the original implementations that we extended. We built a testbed with two x86 machines: one as the user's device and the other as the operator's cell and core. We connect each machine to an SDR device (USRP B205-mini [40]) for radio connectivity. Lastly, the broker's service is deployed on AWS instances [15].

As an opt-in service, LOCA can be incrementally deployed and adopted starting with a small number of LOCA-

compatible users, brokers, and operators: users can have partial privacy by signing up with brokers that support LOCA and by using LOCA-based operators when available and falling back to legacy ones otherwise. We leave an evaluation of the privacy benefits under incremental adoption to future work.

6.2 Scaling analysis

LOCA must be able to scale to a large number of operators serving many users. Therefore, we evaluate whether the three privacy building blocks that we adopt can scale to large deployments, on the order of today's large MVNOs.

6.2.1 Blind signature

Blind signatures are used for generating and verifying unlinkable tokens. We measure a blind signature generation throughput of 522/sec and a verification throughput of 17202/sec on a 2.6GHz Intel I7-8850H CPU. These single-core throughputs are significant: generating 50 tokens for 10 operators per second. Moreover, brokers can easily achieve higher throughput with more cores or machines, hence we conclude that scaling blind signature operations will not be a problem.

6.2.2 Anonymous communication

For anonymous communication schemes in LOCA, an operator must have sufficient network capacity to send attachment requests to brokers. We measure the average network throughput of a Tor circuit to be 4.2 Mbps uplink and 6.1 Mbps downlink (consistent with Tor's reports [73]). Such throughput can support ≈ 400 attachment requests per second. Operators can easily scale up the throughput by establishing multiple Tor circuits with the same token. Alternatively, operators can use other anonymous communication schemes that have higher network throughput, such as VPNs (§6.3).

6.2.3 Aggregate Claiming with zk-SNARK

zk-SNARK has a long setup and proving time [111]. Given our aggregate claiming protocol is based on zk-SNARK, we evaluate whether the protocol can scale to large deployments. Since the generated keys are reused across billing cycles, zk-SNARK setup is performed offline only once, which excludes the setup time from the performance critical path. Hence we focus on the zk-SNARK proving time, which is invoked by each operator at every billing cycle to claim its session groups.

As noted in §4.5, LOCA allows claiming sessions in groups with a configurable size: smaller session groups offer stronger privacy guarantees as they minimize indirect location leakages. However, due to the slow zk-SNARK proving, operators may need to use large session groups so that they can *claim session groups faster than the rate of session group creation* and not develop a backlog of unclaimed sessions, at the cost of some privacy loss. To evaluate the amount of such privacy loss, we answer the following question: how small can session groups be while allowing operators to claim them fast enough? Specifically, we would like to obtain a *lower bound* for the average duration of a session group T[11]. As we will show next, even a single-core zk-SNARK implementation is fast enough to support session groups of small T, hence aggregate claiming will not be a scalability bottleneck.

As noted, we let K represent the maximum number of session groups an operator can claim and N represent the maximum number of session groups in the broker's session group table. If we denote $P(K,N)$ as the time it takes for zk-SNARK to prove the circuit of the claiming function parameterized by K and N, we have the following lower bound for T:

$$T \geq \frac{P(K,N)}{K}$$

To obtain the lower bound, we evaluate the proving time of our implementation for the claiming procedure $P(K,N)$. As mentioned in §4.5, proof-based verifiable computation has a compiler frontend and a zk-SNARK backend. Therefore, to evaluate $P(K,N)$, we need to answer two questions: (i) for a given K and N, how many constraints will the claiming function be compiled into? and (ii) how long will zk-SNARK take to prove these circuits of different sizes?

To answer the first question, we compile claiming functions with different Ks and Ns, and find the following formula that closely matches the numbers of constraints:

$$\#\ of\ constraints = K * (128 * N + 35394)$$

Terms in this formula are tied to the logic of the claiming function. As mentioned in §4.5, the claiming function contains two steps: (i) calculating hashes of the K provided secrets, and (ii) iterating through the N rows in the session table, checking whether the hash matches with one of the K precomputed hashes and adding it to the aggregate if so. Therefore, step (i) generates $35394 * K$ constraints, where 35394 is the number of constraints for computing a single SHA256 hash, consistent with prior work [65]; step (ii) contains an outer loop of N and an inner loop of K, which gets unrolled by the compiler into $128 * K * N$ constraints. Therefore, for a large enough N, the number of constraints scale almost linearly with $K * N$.

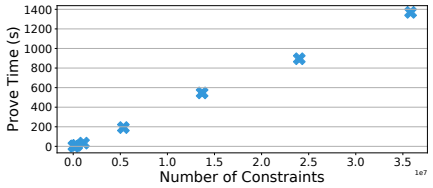


Figure 7: Proving time under varied number of constraints.

To answer the second question, we evaluate the proving time of compiled circuits with different numbers of constraints with a single-core libsnark backend on a 2.5GHz Intel 8259CL CPU. As shown in Fig 7, consistent with prior work [84, 111], the proving time increases linearly with the number of constraints: about 38 seconds per 1 million constraints.

Since we have shown that (i) the number of constraints of the claiming circuit increases linearly w.r.t $K * N$, and (ii) the proving time is linear w.r.t the number of constraints, we know that the *zk-SNARK proving time increases linearly w.r.t* $K * N$, *i.e.,* $P(K,N) = O(K * N)$. The constant factor c depends on the specific compiler frontends and zk-SNARK backends. For

[11] One can calculate the average number of sessions in a group as $T * r$, where r is the deployment-dependent rate of attachments for an operator.

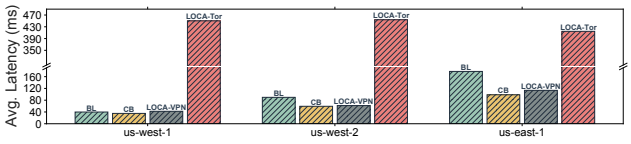


Figure 8: Average attachment latency of Magma baseline (BL), CellBricks (CB), LOCA-VPN and LOCA-Tor.

our implementations, $c \approx 128 * 38\ us = 4.894\ ms$. Therefore,

$$T \geq \frac{P(K,N)}{K} \approx \frac{c*K*N}{K} = c*N$$

This means that the lower bound on the duration of the session group grows *linearly w.r.t.* N. As stated earlier, we are mostly interested in cases of large Ns (*i.e.,* larger numbers of smaller session groups) as these lead to stronger privacy guarantees (§5.1.2). Fortunately, even with only the single-core libsnark backend, the lower bound of T for large N is reasonably small. As an example, the largest circuit that we evaluated (K=64, N=4096) has proving time $P(64,4096)$=1369 s; this translates to a lower bound of T=$P(64,4096)/64$=21.4 s. The asymptotic expression of $T = c*N = 4.864\ ms * 4096 \approx 20\ s$ matches with the measurement. The gap is due to ignoring the $35394 * K$ term, which will reduce as N goes even bigger.

Therefore, with $N = 4096$, the smallest session group that a single-core zk-SNARK can support has a duration of 20 s. This means that users who attach more than 20 s apart cannot reveal any information about each other's location, even if one user's location were leaked to the broker. We do not evaluate circuits with more than $35M$ constraints due to the scaling limit of the libsnark implementation. Recent work [111] on distributed zk-SNARK allows faster proving of much larger circuits, the evaluation of which is left to future work.

6.3 Performance analysis

Lastly, we would like to understand the performance that users receive with LOCA. Procedures like token generation and aggregate claiming happen off the critical path of users receiving services, thus do not affect user experience. Instead, we focus on the attachment procedure, since LOCA's attachment is both more complex and more frequent than today's protocols. We thus measure the additional latency overhead that LOCA adds to the attachment procedure.

We replicate the wide-area test setup from CellBricks [77]: the user equipment and the operator's cell and cellular core are always located in our local testbed, and we run experiments with the subscriber database (in the case of Magma) and the broker hosted on AWS EC2 [9]. This matches deployment practice where certain core network components are run in the carrier's datacenter. For each setup, we repeat the same attachment request using different cellular implementations 100 times and report the average performance.

Fig.8 shows the attachment latency after removing the time spent in lower radio layers (*i.e.,* RRC layer and below) for different placements of the subscriber database and broker. We compare four schemes: (i) unmodified Magma (baseline, denoted BL, that captures today's cellular architecture), (ii) CellBricks (denoted CB), LOCA's attachment protocol with (iii) VPN (denoted LOCA-VPN) and (iv) Tor (denoted LOCA-Tor) as the anonymous communication channel.

We make two observations from these results. First, the choice of anonymous communication scheme introduces a tradeoff between trust assumptions and attachment latency: LOCA-VPN requires trusting the VPN provider but achieves faster attachments than LOCA-Tor. In fact, LOCA-VPN is only 5 to 15 *ms* slower than CellBricks and still faster than today's attachment (*i.e.,* Magma). The reason we outperform Magma's attachment latency is because today's attachment procedure requires two round trips to the cloud, while CellBricks optimized this process to a single round-trip; since we build on CellBricks, we inherit this performance gain.

Our second observation is that even the slower LOCA-Tor is sufficiently fast for periodic reattachments: prior work [77] shows that attachment latencies of up to 500 *ms* have a minimal impact on application performance, even when users reattach on a per-tower basis. Hence LOCA-Tor, with a constant 400 *ms* latency due to the overhead of Tor [73], can support frequent reattachments with minimal disruptions.

7 Discussion

Viewing LOCA as a first step towards privacy-preserving cellular infrastructure, we next discuss two notable areas for improvement and potential directions to achieving them: (i) supporting beyond semi-honest and non-colluding participants, and (ii) improving non-privacy-related aspects of LOCA.

7.1 Beyond semi-honest and non-colluding

As stated in §3.1, there are both financial and legal reasons for brokers and operators to be semi-honest and not collude. However, relaxing these assumptions can certainly facilitate adoption. We next discuss directions towards such relaxation.

Semi-honest: LOCA suffers from privacy leakages in the face of various active attacks, *e.g.,* those based on out-of-protocol information (§5.2), which restricts it to semi-honest participants. We see two orthogonal directions towards supporting more aggressive participants. First, one could adopt specific defense mechanisms for different attacks (*e.g.,* traffic analysis, device fingerprinting) that have been proposed in prior work [43, 56, 60, 103, 110, 114]. LOCA, as a protocol-layer solution, can coexist with these mechanisms. Second, instead of averting attacks, one can detect these attacks and punish the misbehaving participants. The detection mechanism can involve multiple parties. For instance, operator over-reporting usage can be detected by brokers cross comparing the operator's reports with the ones from users. For the punishment mechanism, a promising approach is to build up a reputation system [77], where misbehaviors are factored into participant's reputation scores. Participants with poor reputation then receive degraded treatments: *e.g.,* a broker can decline to authorize an operator in the registration phase (§4.1). Such an approach is appealing in the cellular context, where brokers and operators need to remain operational for long enough to see a profit, allowing their track records to be built up.

Non-colluding: As elaborated in §5.3, except for direct collusion between brokers and operators that serve the user, other forms of collusion only incur minimal leakages in LOCA. An interesting question is then whether we could relax this requirement of no broker-operator collusion. Intuitively, preserving location privacy with *arbitrary* collusion seems unattainable: if a broker colludes with all the operators, it easily knows both the user's identity and all of her locations. Instead, we believe it is both feasible and interesting to investigate whether one could provide *partial* privacy guarantee if only a *subset* of operators collude with brokers. Under such a scenario, the coverage of non-colluding operators forms a region where little location information is revealed. Such a region is referred to as a *mix zone* and widely studied for location privacy in non-cellular contexts [17, 18, 53], and future work could leverage the insights of these work for cellular privacy.

7.2 Beyond privacy

Another area for improvement is the design and evaluation on non-privacy-related aspects of LOCA, such as performance and operational support. For performance, in §6.3, we measure LOCA's attachment latency to be less than 500 ms even with slower anonymous communication channel (*i.e.,* Tor), which was evaluated in [77] to have minimal performance impacts to applications like voice calls, video streaming and web browsing. It would be interesting to evaluate on more challenging applications such as video conferencing. Moreover, besides reducing trajectory leakages (§5.1.3), make-before-break handovers are expected to have better performance as well, the evaluation of which in LOCA is left to future work.

For operational support, LOCA supports tasks like identity-based services by having brokers offload these tasks to authorized but identity unknown operators (§4.4). However, there might be tasks that require knowledge of the operator's identity, such as recording misbehaving operators (for the aforementioned reputation system) and performing on-site inspections. To support these tasks, one potential approach is to involve a trusted third party when generating unlinkable tokens (§4.1). The goal is that upon legitimate requests, this third party can later assist in revealing the operator's identity for a token. One promising direction towards achieving this goal is to extend the registration phase with cryptographic constructs like secure multi-party computation (MPC) [35, 48, 113].

8 Related Work

Cellular: There has been extensive prior work on mitigating privacy violations by third parties other than network operators [11, 46, 58, 63, 68, 87, 93, 94, 100]. Our work instead focuses on protecting a user's location privacy from the network operator itself. To our knowledge, PGPP [86] is the only prior work that systematically studies this issue. As discussed earlier, PGPP adopts a different approach based on hiding users' identities from the network operator, which however compromises the network's ability to provide identity-based services and does not address the issue of trajectory-related leakages. One advantage of PGPP is higher tolerance for collusions, as it hides user's identity from both operators and brokers. However, it also assumes semi-honest participants who will not actively thwart its privacy mechanisms.

CellBricks [77] is a new cellular architecture that aims to democratize cellular access by enabling users to easily leverage small-scale operators. LOCA borrows the idea of user-driven mobility, although we use it for privacy reasons while CellBricks requires it to give users the ability to dynamically select an operator of their choice. CellBricks does not address the issue of location privacy and hence is similar to 3GPP protocols in this regard. In fact, we note that the importance of hiding O's identity from B is greater under the CellBricks vision of larger numbers of smaller-scale operators.

General location privacy: There is extensive prior work on location privacy in non-cellular contexts [17, 36, 67, 76, 90, 106, 112]. These reveal four general methods for protecting location privacy: (i) regulatory strategies – government rules to regulate the use of personal information; (ii) privacy policies – trust-based agreements between individuals and whoever is receiving their location data; (iii) anonymity – use a pseudonym and create ambiguity by grouping with other people. (iv) obfuscation – temporal or spatial degradation of the location data. Regulatory strategies and privacy policies are orthogonal to computational countermeasures like techniques adopted in LOCA. In the cellular context, neither obfuscation nor anonymity is desirable: obfuscation is not feasible, because a user's location data is generated by the infrastructure, the temporal or spatial resolution of which is not determined by the user; anonymity is the approach adopted by PGPP [86] which, as discussed earlier, compromises on identity-based services. LOCA exploits the unique role of brokers and adopts a novel approach to preserving location privacy while supporting identity-based services. LOCA's approach of strategically hiding different pieces of information from each party has been investigated for preserving privacy in other contexts as well, such as Apple's private relay [31].

Applications of LOCA's privacy building blocks: Blind signatures have been applied for e-voting [59, 64, 75]. Anonymous communication has been used in social networking and web browsing [44, 55, 99]. Proof-based verifiable computation has been used in outsourced computing [24, 27, 70]. LOCA synthesizes these building blocks to support cellular procedures like attachment and aggregate claiming.

9 Conclusion

We presented LOCA, a novel cellular architecture that provides location privacy while supporting identity-based services such as usage-based billing, QoS, and lawful intercept.

We view our work as a first step towards enabling privacy-preserving communication infrastructure and hope that future work will extend our design to address additional threat models and reduced overheads, as well as explore the applicability of LOCA's design to other access technologies.

References

[1] 3GPP. Lte;telecommunication management; performance management (pm); performance measurements evolved universal terrestrial radio access network (e-utran). Technical Specification (TS) 32.425, 3rd Generation Partnership Project (3GPP), 08 2016. Version 13.5.0.

[2] 3GPP. 5g; security architecture and procedures for 5g system. Technical Specification (TS) 33.501, 3rd Generation Partnership Project (3GPP), 10 2018. Version 15.2.0.

[3] 3GPP. Lawful Interception (LI);Handover interface for the lawful interception of telecommunications traffic. https://www.etsi.org/deliver/etsi_es/201600_201699/201671/03.02.01_50/es_201671v030201m.pdf, 2018.

[4] 3GPP. Lawful interception architecture and functions. Technical Specification (TS) 33.107, 3rd Generation Partnership Project (3GPP), 07 2019. Version 15.6.0.

[5] 3GPP. 5g; security architecture and procedures for 5g system. Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP), 10 2020. Version 16.6.0.

[6] 3GPP. Lte; 3gpp system architecture evolution (sae); security architecture. Technical Specification (TS) 33.401, 3rd Generation Partnership Project (3GPP), 03 2020. Version 15.11.0.

[7] 3GPP. Non-Access Stratum. https://www.3gpp.org/technologies/keywords-acronyms/96-nas, 2020.

[8] 3GPP. Nr and ng-ran overall description. Technical Specification (TS) 38.300, 3rd Generation Partnership Project (3GPP), 07 2020. Version 16.2.0.

[9] Amazon Web Service. Aws ec2 regions. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html, 2021.

[10] Apple. Privacy - apple. https://www.apple.com/privacy/, 2021.

[11] Myrto Arapinis, Loretta Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: fix and verification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 205–216, 2012.

[12] Richard Arratia and Louis Gordon. Tutorial on large deviations for the binomial distribution. *Bulletin of mathematical biology*, 51(1):125–131, 1989.

[13] AT&T. AT&T private cellular networks. https://www.business.att.com/products/att-private-cellular-networks.html, 2020.

[14] AT&T. Deep packet inspection explained. https://cybersecurity.att.com/blogs/security-essentials/what-is-deep-packet-inspection, 2021.

[15] AWS. Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/.

[16] Leda Bargiotti, Inge Gielis, Bram Verdegem, Pieter Breyne, Francesco Pignatelli, Paul Smits, Ray Boguslawski, et al. Guidelines for public administrations on location privacy: European union location framework. Technical report, Joint Research Centre (Seville site), 2016.

[17] Alastair R Beresford and Frank Stajano. Location privacy in pervasive computing. *IEEE Pervasive computing*, 2(1):46–55, 2003.

[18] Alastair R Beresford and Frank Stajano. Mix zones: User privacy in location-aware services. In *IEEE Annual conference on pervasive computing and communications workshops, 2004. Proceedings of the Second*, pages 127–131. IEEE, 2004.

[19] Naga Bhushan, Junyi Li, Durga Malladi, Rob Gilmore, Dean Brenner, Aleksandar Damnjanovic, Ravi Teja Sukhavasi, Chirag Patel, and Stefan Geirhofer. Network densification: the dominant theme for wireless evolution into 5g. *IEEE Communications Magazine*, 52(2):82–89, 2014.

[20] Camila CS Caiado and Pushpa N Rathie. Polynomial coefficients and distribution of the sum of discrete uniform variables. In *Eighth Annual Conference of the Society of Special Functions and their Applications, Pala, India, Society for Special Functions and their Applications*, 2007.

[21] CellBricks. Cellbricks. https://cellbricks.github.io/, 2021.

[22] Mobile Internet Resource Center. Top pickfeatured overview: postpaid consumer plans by verizon (cellular data plans). https://www.rvmobileinternet.com/gear/the-verizon-plan/, 2021.

[23] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[24] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Carlos Cid. Multi-client non-interactive verifiable computation. In *Theory of Cryptography Conference*, pages 499–518. Springer, 2013.

[25] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[26] Gene Connolly, Anatoly Sachenko, and George Markowsky. Distributed traceroute approach to geographically locating ip devices. In *Second IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings*, pages 128–131. IEEE, 2003.

[27] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE, 2015.

[28] Joseph Cox. I gave a bounty hunter $300. then he located our phone. https://www.vice.com/en/article/nepxbz/i-gave-a-bounty-hunter-300-dollars-located-phone-microbilt-zumigo-tmobile, 2019.

[29] Joseph Cox. Stalkers and debt collectors impersonate cops to trick big telecom into giving them cell phone location data. https://www.vice.com/en/article/panvkz/stalkers-debt-collectors-bounty-hunters-impersonate-cops-phone-location-data, 2019.

[30] Cricket. Cricket wireless. https://www.cricketwireless.com/, 2021.

[31] Jason Cross. icloud+ private relay faq: Everything you need to know. https://www.macworld.com/article/348965/icloud-plus-private-relay-safari-vpn-encryption-privacy.html, 2021.

[32] CryptoBallot. Rsa blind signing using a full domain hash. https://github.com/cryptoballot/rsablind, 2021.

[33] Boris Danev, Davide Zanetti, and Srdjan Capkun. On physical-layer identification of wireless devices. *ACM Computing Surveys (CSUR)*, 45(1):1–29, 2012.

[34] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. {DORY}: An encrypted search system with distributed trust. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1101–1119, 2020.

[35] Wenliang Du and Mikhail J Atallah. Secure multi-party computation problems and their applications: a review and open problems. In *Proceedings of the 2001 workshop on New security paradigms*, pages 13–22, 2001.

[36] Matt Duckham and Lars Kulik. Location privacy and location-aware computing. *Dynamic & mobile GIS: investigating change in space and time*, 3:35–51, 2006.

[37] Steffen Eger. Stirling's approximation for central extended binomial coefficients. *The American Mathematical Monthly*, 121(4):344–349, 2014.

[38] Ericsson. Evolving cellular IoT for industry digitalization. https://www.ericsson.com/en/internet-of-tRFWirelessWorldhings/iot-connectivity/cellular-iot, 2020.

[39] ETSI. Lawful intercept ETSI. https://www.etsi.org/technologies/lawful-interception, 2020.

[40] Ettus. Usrp b205mini. https://www.ettus.com/all-products/usrp-b205mini-i/, 2020.

[41] Facebook. Magma. https://www.magmacore.org/, 2021.

[42] Nour-Eddine Fahssi. Some identities involving polynomial coefficients. *arXiv preprint arXiv:1507.07968*, 2015.

[43] Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. Protecting privacy of {BLE} device users. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1205–1221, 2016.

[44] Eran Gabber, Phillip B Gibbons, Yossi Matias, and Alain Mayer. How to make personalized web browsing simple, secure, and anonymous. In *International Conference on Financial Cryptography*, pages 17–31. Springer, 1997.

[45] Ruchi Garg. Dual active protocol stack handover (daps ho). https://www.linkedin.com/pulse/dual-active-protocol-stack-handover-daps-ho-ruchi-garg/, 2021.

[46] M Køien Geir et al. Privacy enhanced mutual authentication in lte. In *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 614–621. IEEE, 2013.

[47] Hadi Givehchian, Nishant Bhaskar, Eliana Rodriguez Herrera, Héctor Rodrigo López Soto, Christian Dameff, Dinesh Bharadia, and Aaron Schulman. Evaluating physical-layer ble location tracking attacks on mobile devices. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1690–1704. IEEE, 2022.

[48] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78(110), 1998.

[49] Google. Google-Fi. https://fi.google.com/about/, 2021.

[50] Swish Goswami. The rising concern around consumer data and privacy. https://www.forbes.com/sites/forbestechcouncil/2020/12/14/the-rising-concern-around-consumer-data-and-privacy/?sh=6e6200a6487e, 2020.

[51] David Goulet. Torsocks. https://github.com/dgoulet/torsocks, 2021.

[52] GSMA. Enabling neutral host: CCS case study. https://www.gsma.com/futurenetworks/wp-content/uploads/2018/09/180920-CCS_GSMA_Case_Study-FINAL_NE-Modelling-removed.pdf, 2020.

[53] Nan Guo, Linya Ma, and Tianhan Gao. Independent mix zone for location privacy in vehicular networks. *IEEE Access*, 6:16842–16850, 2018.

[54] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 91–107, 2016.

[55] Nguyen Phong Hoang and Davar Pishva. Anonymous communication and its importance in social networking. In *16th International Conference on Advanced Communication Technology*, pages 34–39. IEEE, 2014.

[56] Jinsong Hu, Shihao Yan, Feng Shu, Jiangzhou Wang, Jun Li, and Yijin Zhang. Artificial-noise-aided secure transmission with directional modulation based on random frequency diverse arrays. *IEEE Access*, 5:1658–1667, 2017.

[57] Huawei. Huawei privacy. https://consumer.huawei.com/en/privacy/, 2021.

[58] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[59] Subariah Ibrahim, Maznah Kamat, Mazleena Salleh, and Shah Rizan Abdul Aziz. Secure e-voting with blind signature. In *4th National Conference of Telecommunication Technology, 2003. NCTT 2003 Proceedings.*, pages 193–197. IEEE, 2003.

[60] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Diaz, and Matthew Wright. Toward an efficient website fingerprinting defense. In *European Symposium on Research in Computer Security*, pages 27–46. Springer, 2016.

[61] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. *IACR Cryptol. Eprint Arch.*, 2011:272, 2011.

[62] Kate Kaye. The $24 billion data business that telcos don't want to talk about. https://adage.com/article/datadriven-marketing/24-billion-data-business-telcos-discuss/301058?mod=article_inline, 2019.

[63] Mohammed Shafiul Alam Khan and Chris J Mitchell. Trashing imsi catchers in mobile networks. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 207–218, 2017.

[64] Malik Sikandar Hayat Khiyal, Aihab Khan, Saba Bashir, Farhan Hassan Khan, and Shaista Aman. Dynamic blind group digital signature scheme in e-banking. *International Journal of Computer and Electrical Engineering*, 3(4):514–519, 2011.

[65] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961. IEEE, 2018.

[66] KrebsOnSecurity. Tracking firm locationsmart leaked location data for customers of all major u.s. mobile carriers without consent in real time via its web site. https://krebsonsecurity.com/2018/05/tracking-firm-locationsmart-leaked-location-data-for-customers-of-all-major-u-s-mobile-carriers-in-real-time-via-its-web-site/, 2018.

[67] John Krumm. A survey of computational location privacy. *Personal and Ubiquitous Computing*, 13(6):391–399, 2009.

[68] Denis Foo Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location leaks on the gsm air interface. *ISOC NDSS (Feb 2012)*, 2012.

[69] SCIPR Lab. Libsnark. https://github.com/scipr-lab/libsnark, 2021.

[70] Junzuo Lai, Robert H Deng, HweeHwa Pang, and Jian Weng. Verifiable computation on outsourced encrypted data. In *European Symposium on Research in Computer Security*, pages 273–291. Springer, 2014.

[71] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196, 2017.

[72] Sangwon Lee, Sylvia M Chan-Olmsted, and Hsiao-Hui Ho. The emergence of mobile virtual network operators (mvnos): An examination of the business strategy in the global mvno market. *The International Journal on Media Management*, 10(1):10–21, 2008.

[73] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. A case study on measuring statistical data in the Tor anonymity network. In *Proceedings of the Workshop on Ethics in Computer Security Research (WECSR 2010)*, LNCS. Springer, January 2010.

[74] Natasha Lomas. Uh oh! european carriers are trying to get into 'personalized' ad targeting. https://techcrunch.com/2022/06/24/trustpid/, 2022.

[75] Lourdes López-García, Luis J Dominguez Perez, and Francisco Rodríguez-Henríquez. A pairing-based blind signature e-voting scheme. *The Computer Journal*, 57(10):1460–1471, 2014.

[76] Zhaojun Lu, Gang Qu, and Zhenglin Liu. A survey on recent advances in vehicular network security, trust, and privacy. *IEEE Transactions on Intelligent Transportation Systems*, 20(2):760–776, 2018.

[77] Zhihong Luo, Silvery Fu, Mark Theis, Shaddi Hasan, Sylvia Ratnasamy, and Scott Shenker. Democratizing cellular access with cellbricks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 626–640, 2021.

[78] Anna Lysyanskaya, Ronald L Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In *International Workshop on Selected Areas in Cryptography*, pages 184–199. Springer, 1999.

[79] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[80] NordVPN. NordVPN. https://nordvpn.com, 2021.

[81] OpenCellID. The world's largest open database of cell towers. https://www.opencellid.org/, 2021.

[82] Christoph Paasch and Sebastien Barre. Multipath TCP. https://www.multipath-tcp.org, 2021. Accessed: 2020-04-29.

[83] The Pepper Project. Pequin: An end-to-end toolchain for verifiable computation, snarks, and probabilistic proofs. https://github.com/pepper-project/pequin, 2021.

[84] Charles Rackoff and Daniel R Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Annual International Cryptology Conference*, pages 433–444. Springer, 1991.

[85] Samsung. Samsung's approach to privacy. https://www.samsung.com/us/account/our-approach-to-privacy/, 2021.

[86] Paul Schmitt and Barath Raghavan. Pretty good phone privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1737–1754, 2021.

[87] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical attacks against privacy and availability in 4g/lte mobile communication systems. *arXiv preprint arXiv:1510.07563*, 2015.

[88] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM conference on special interest group on data communication*, pages 213–226, 2015.

[89] Yi-Sheng Shiu, Shih Yu Chang, Hsiao-Chun Wu, Scott C-H Huang, and Hsiao-Hwa Chen. Physical layer security in wireless networks: A tutorial. *IEEE wireless Communications*, 18(2):66–74, 2011.

[90] Reza Shokri, George Theodorakopoulos, Jean-Yves Le Boudec, and Jean-Pierre Hubaux. Quantifying location privacy. In *2011 IEEE symposium on security and privacy*, pages 247–262. IEEE, 2011.

[91] Pham Hai Son, Sudan Jha, Raghvendra Kumar, Jyotir Moy Chatterjee, et al. Governing mobile virtual network operators in developing countries. *Utilities Policy*, 56:169–180, 2019.

[92] srsRAN. srsLTE: Your own mobile network. https://www.srslte.com/, 2020.

[93] Daehyun Strobel. Imsi catcher. *Chair for Communication Security, Ruhr-Universität Bochum*, 14, 2007.

[94] Keen Sung, Brian Neil Levine, and Marc Liberatore. Location privacy without carrier cooperation. In *IEEE Workshop on Mobile Security Technologies, MOST*, page 148. Citeseer, 2014.

[95] Techplayon. 5g nr dual active protocol stack (daps) handover – 3gpp release 16. https://www.techplayon.com/5g-nr-dual-active-protocol-stack-daps-handover-3gpp-release-16/, 2020.

[96] TelcoBridges. Lawful intercept solutions. https://www.telcobridges.com/solutions/operators/lawful-intercept, 2021.

[97] Telecoms. Neutral host networks and how to support them. https://telecoms.com/opinion/neutral-host-networks-and-how-to-support-them/, 2020.

[98] Tessian. 22 biggest gdpr fines of 2019, 2020, and 2021 (so far). https://www.tessian.com/blog/biggest-gdpr-fines-2020/, 2021.

[99] Tor. Tor. https://www.torproject.org/, 2021.

[100] Fabian Van Den Broek, Roel Verdult, and Joeri de Ruiter. Defeating imsi catchers. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, pages 340–351, 2015.

[101] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.

[102] Riad S Wahby, Srinath TV Setty, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[103] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 143–157, 2014.

[104] Lance Whitney. Data privacy is a growing concern for more consumers. https://www.techrepublic.com/article/data-privacy-is-a-growing-concern-for-more-consumers/, 2021.

[105] Zack Whittaker. Us cell carriers are selling access to your real-time phone location data. https://www.zdnet.com/article/us-cell-carriers-selling-access-to-real-time-location-data/, 2018.

[106] Björn Wiedersheim, Zhendong Ma, Frank Kargl, and Panos Papadimitratos. Privacy in inter-vehicular networks: Why simple pseudonym change is not enough. In *2010 Seventh international conference on wireless on-demand network systems and services (WONS)*, pages 176–183. IEEE, 2010.

[107] Josephine Wolff and Nicole Atallah. Early gdpr penalties: Analysis of implementation and fines through may 2020. *Journal of Information Policy*, 11:63–103, 2021.

[108] Ben Wolford. What are the gdpr fines? https://gdpr.eu/fines/, 2021.

[109] RF Wireless World. LTE QoS quality of service, class identifier(QCI), QoS in LTE. https://www.rfwireless-world.com/Tutorials/LTE-QoS.html, 2021.

[110] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9. Citeseer, 2009.

[111] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 675–692, 2018.

[112] Hui Zang and Jean Bolot. Anonymization of location data does not work: A large-scale measurement study. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 145–156, 2011.

[113] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. Secure multi-party computation: theory, practice and applications. *Information Sciences*, 476:357–372, 2019.

[114] Yulong Zou, Jia Zhu, Xianbin Wang, and Victor CM Leung. Improving physical-layer security in wireless communications using diversity techniques. *IEEE Network*, 29(1):42–48, 2015.

Appendix

A Unlinkable token

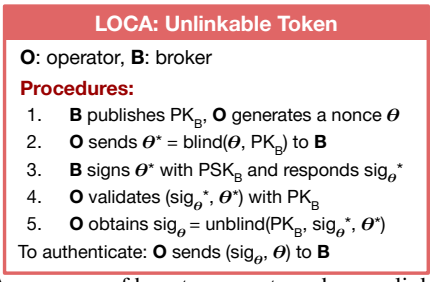


Figure 9: A summary of how to generate and use unlinkable tokens.

Fig.9 summarizes how O obtains tokens. The protocol starts with B publishing its public key PK_B and O generating a nonce θ (*i.e.,* the token). To get B *blindly-sign* the θ, O first *blinds* θ using PK_B and sends the blinded token θ^* to B, requesting B to sign the token. O reveals its identity to B who can decide whether to accept this request. Next, B signs θ^* using its private key PSK_B and returns the blind signature sig^*_θ to O which O can validate using PK_B.
Next, O obtains the *unblinded signature* of the token using PK_B, sig^*_θ, and θ^*. To authenticate itself to B, O sends θ and the unblinded signature sig_θ to B. B can then verify the token's authenticity with sig_θ as a normal digital signature. Note that B *cannot* link θ to O since θ was blindly-signed and never seen by B (only the blinded θ^* was).

B Aggregate Claiming

Simulation setup: For each ratio K/N, we scale N while increasing K proportionally according to the ratio and estimate the expected number of matched subsets for the (K,N). For example, for the ratio K/N=1/10, we run experiments for (K,N)=$(1,10),(2,20),...,(5,50)$. For each (K,N), we again make the simplification to not consider subsets that contain any of the K session groups. By doing so, we can independently sample arrays Xs of length $(N-K)$ and Ys of length K, count the number of subsets in X that have the same sum as Y for each pair of (X,Y), and report the average of all the pairs as the estimate for (K,N), denoted as R. To ensure that the estimate is accurate, we use a large number of samples for each (K,N), up to 2^{25} so that the simulation can finish within a reasonable time frame.
Theoretical proof: For this proof, we make use of *polynomial coefficients*, also named as extended binomial coefficients, which are natural extensions of the well-known binomial coefficient. For $n,m \in Z_{>0}$ Polynomial coefficients $\binom{n}{k}_m$ is the coefficient of x^k in the following expansion:

$$(1+x+...+x^m)^n = \sum_{k=0}^{k=mn} \binom{n}{k}_m x^k$$

Note that $\binom{n}{k}_m = 0$ for $k \notin \{0,..,mn\}$. Binomial coefficient is the special case where $m=1$. An equivalent definition of $\binom{n}{k}_m$ is:

$$\binom{n}{k}_m = \sum_{\substack{k_0\geq 0,...,k_m\geq 0 \\ k_0+...+k_m=n \\ 0\cdot k_0+...+m\cdot k_m=k}} \binom{n}{k_0,...,k_m} \tag{1}$$

It is known that polynomial coefficients are symmetric: $\binom{n}{k}_m = \binom{n}{mn-k}_m$, and $\binom{n}{k}_m$ is a non-decreasing function of k for $0 \leq k \leq \lfloor \frac{mn}{2} \rfloor$ and a non-increasing function for $\lceil \frac{mn}{2} \rceil \leq k \leq mn$ [42].
Prior work has shown that the sum of N iid random variables from the discrete uniform distribution of $\{0,...,m\}$ ($U\{0,m\}$), denoted as S_N, has the following closed-form distribution expressed with polynomial coefficients [20]:

$$\begin{aligned}
P(S_N=y) &= \sum_{\substack{a_0\geq 0,...,a_m\geq 0 \\ a_0+...+a_m=N \\ 0\cdot a_0+...+m\cdot a_m=y}} P(a_0,...,a_m) && (a_i \text{ stands for the number of elements equal to } i) \\
&= \sum_{\substack{a_0\geq 0,...,a_m\geq 0 \\ a_0+...+a_m=N \\ 0\cdot a_0+...+m\cdot a_m=y}} (\frac{1}{m+1})^N \binom{N}{a_0,...,a_m} && (\text{each element can be putted into bucket } i \text{ with a likelihood of } \frac{1}{m+1}) \\
&= (\frac{1}{m+1})^N \binom{N}{y}_m && (\text{according to definition (1)})
\end{aligned}$$

Lemma B.1. *The distribution of sums of N iid random variables from $U\{1,m\}$ has the following closed-form expression:* $P_N(y) = (\frac{1}{m})^N \binom{N}{y-N}_{m-1}$

Proof. The proof is similar to the proof above for sums of N iid random variables from $U\{0,m\}$, with some minor adjustment:

$$\begin{aligned}
P_N(y) &= \sum_{\substack{a_1\geq 0,\ldots,a_m\geq 0\\ a_1+\ldots+a_m=N\\ 1\cdot a_1+\ldots+m\cdot a_m=y}} (\frac{1}{m})^N \binom{N}{a_1,\ldots,a_m} \\
&= \sum_{\substack{a_1\geq 0,\ldots,a_m\geq 0\\ a_1+\ldots+a_m=N\\ 0\cdot a_1+\ldots+(m-1)\cdot a_m=y-N}} (\frac{1}{m})^N \binom{N}{a_1,\ldots,a_m} && \text{(align with the format of (1))} \\
&= (\frac{1}{m})^N \binom{N}{y-N}_{m-1} && \text{(according to definition (1))}
\end{aligned}$$

□

Lemma B.2. *The distribution of subset sums of N iid random variables from $U\{1,m\}$ has the following closed-form expression:* $Q_N(y) = (\frac{1}{2m})^N \sum_{k=0}^{N} \binom{N}{k} m^k \binom{N-k}{y-(N-k)}_{m-1}$

Proof. Subset sums of $U\{1,m\}$ can be equivalently treated as sums of elements X_i that has the following distribution:

$$P(X_i) = \begin{cases} \frac{1}{2}, & \text{if } X_i = 0 \\ \frac{1}{2m}, & \text{if } X_i \in \{1,\ldots,m\} \\ 0, & \text{otherwise} \end{cases}$$

Therefore, we can calculate the probability of a subset sum by multiplying the probability of having different numbers of zeros with the probability of adding up to the sum with the remaining non-zero elements:

$$\begin{aligned}
Q_N(y) &= \sum_{k=0}^{N} \binom{N}{k} (\frac{1}{2})^k \sum_{\substack{a_1\geq 0,\ldots,a_m\geq 0\\ a_1+\ldots+a_m=N-k\\ 1\cdot a_1+\ldots+m\cdot a_m=y}} (\frac{1}{2m})^{N-k} \binom{N-k}{a_1,\ldots,a_m} \\
&= \sum_{k=0}^{N} \binom{N}{k} (\frac{1}{2})^k \sum_{\substack{a_1\geq 0,\ldots,a_m\geq 0\\ a_1+\ldots+a_m=N-k\\ 0\cdot a_1+\ldots+(m-1)\cdot a_m=y-(N-k)}} (\frac{1}{2m})^{N-k} \binom{N-k}{a_1,\ldots,a_m} && \text{(similar to Lemma B.1)} \\
&= \sum_{k=0}^{N} \binom{N}{k} (\frac{1}{2})^k (\frac{1}{2m})^{N-k} \binom{N-k}{y-(N-k)}_{m-1} && \text{(according to definition (1))} \\
&= (\frac{1}{2m})^N \sum_{k=0}^{N} \binom{N}{k} m^k \binom{N-k}{y-(N-k)}_{m-1}
\end{aligned}$$

□

Lemma B.3. *Here we define a "head" function $H(hN;N,p)(0\leq h\leq 1)$ and a "tail" function $T(tN;N,p)(0\leq t\leq 1)$, where:*

$$H(hN;N,p) = \sum_{k=0}^{hN} \binom{N}{k} p^k; \qquad T(tN;N,p) = \sum_{k=tN}^{N} \binom{N}{k} p^k$$

Then we can prove the following: if $h < \frac{p}{1+p}$, $H(hN;N,p) \leq (b_H)^N$, with $b_H < (1+p)$. Similarly, if $t > \frac{p}{1+p}$, $T(tN;N,p) \leq (b_T)^N$, with $b_T < (1+p)$.

Proof. Here we show the proof for the head function, the proof for the tail function is very similar. The idea is to rewrite the head function to follow the format of a binomial distribution, so that we could use tail bounds for binomial distributions to

provide a lower bound. Specifically:

$$\begin{aligned}
H(hN;N,p) &= \sum_{k=0}^{hN} \binom{N}{k} p^k \\
&= (1+p)^N \sum_{k=0}^{hN} \binom{N}{k} (\frac{p}{1+p})^k (\frac{1}{1+p})^{N-k} \\
&= (1+p)^N \sum_{k=0}^{hN} \binom{N}{k} (p')^k (1-p')^{N-k} && \text{(with } p' = \frac{p}{1+p}\text{)} \\
&= (1+p)^N F(hN;N,p')
\end{aligned}$$

where $F(hN;N,p')$ refers to the probability of having at most hN successes in a Binomial trial $B(N,p')$. For $F(hN;N,p')$, it's known that if $\frac{hN}{N} = h \leq p'$, we have the following tail bounds [12]:

$$F(hN;N,p') \leq \exp[-Nf(h,p')] \qquad \text{with } f(h,p') = \begin{cases} 2(h-p')^2, & \text{with Hoeffding's inequality} \\ D(h||p'), & \text{with Chernoff bound} \end{cases}$$

where $D(a||p)$ is the relative entropy between a $Bernoulli(a)$ (a-coin) and a $Bernoulli(p)$ (p-coin):
$D(a||p) = (a)\log\frac{a}{p} + (1-a)\log\frac{1-a}{1-p}$. For either of this, $f(h,p') > 0$ for $h < p'$. Therefore, for $H(hN;N,p)$, if $h < p' = \frac{p}{1+p}$, we have:

$$\begin{aligned}
H(hN;N,p) &= (1+p)^N F(hN;N,p') \\
&\leq (1+p)^N \exp[-Nf(h,p')] \\
&= \{(1+p)\exp[-f(h,p')]\}^N \\
&= (b_H)^N && \text{(with } b_H = (1+p)\exp[-f(h,p')] < (1+p)\text{)}
\end{aligned}$$

□

Lemma B.4.

$$\frac{1}{m^2+m+1} < 1 - \frac{ln(m)}{ln(m+1)}, \quad \forall m \in Z_{>0}$$

Proof. This is equivalent as showing

$$h(m) = (m^2+m+1)ln(m) - (m^2+m)ln(m+1) < 0, \quad \forall m \in Z_{>0}$$

Taking derivative, we have

$$\begin{aligned}
h'(m) &= (2m+1)\ln(\frac{m}{m+1}) + 1 + \frac{1}{m} \\
&\leq (2m+1)(\frac{m}{m+1} - 1) + 1 + \frac{1}{m} && \text{(with } \ln(x) \leq x-1\text{)} \\
&= \frac{-m^2+m+1}{m(m+1)} < 0 && \text{(for } m \geq 2 > \frac{1+\sqrt{5}}{2}\text{)}
\end{aligned}$$

Therefore, because (i) $h(m)$ monotonically decreases for $m \geq 2$, and (ii) $h(1), h(2) < 0$, we have $h(m) < 0, \forall m \in Z_{>0}$ □

$f[n]$ is said to be exponential w.r.t n iff:

$$\exists M \in R_{>0}, c \in R_{>1}, \quad \lim_{n\to\infty} \frac{f[n]}{c^n} = M$$

Therefore, a sufficient condition for $f[n]$ to be exponential is that

$$\exists M \in R_{>0}, c \in R_{>1}, N_0 \in Z, \quad f[n] \geq M \cdot c^n, \quad \forall n \geq N_0$$

With this we could prove the following lemma:

Lemma B.5. *For any integer $K \geq 1$, $h_K[n] = a^n - \sum_{i=1}^{K} b_i^N$ is exponential w.r.t n, if $a > 1$ and $a > \max_{i:i\in\{1,..,K\}} b_i$*

Proof. $h_K[n]$ can be rewritten as $h_K[n] = \sum_{i=1}^{K} \frac{1}{K}a^n - b_i^n$. We can prove that $\frac{1}{K}a^n - b_i^n$ are exponential for all b_i's. Specifically, we can show that.

$$\forall c \in \{x \in R \mid \max(b_i, 1) < x < a\}, M \in R_{>0}, \exists N(a,c,M) \in Z, \quad \frac{1}{K}a^n - b_i^n > Mc^n, \quad \forall n \geq N(a,c,M)$$

This because $\frac{1}{K}a^n - b_i^n > Mc^n \Leftrightarrow \frac{1}{k}(\frac{a}{c})^n > M + (\frac{b_i}{c})^n$. By having $n \geq N(a,c,M) = \lceil log_{\frac{a}{c}}[k(M+1)] \rceil$, we have $\frac{1}{k}(\frac{a}{c})^n > M+1 > M + (\frac{b_i}{c})^n$. Therefore, $h_K[n] = \sum_{i=0}^{K} \frac{1}{K}a^n - b_i^n$ is obviously exponential:

$$\forall c \in \{x \in R \mid \max(\max_{i:i\in\{1,..,K\}} b_i, 1) < x < a\}, M \in R_{>0}, \exists N(a,c,M) \in Z, \quad h_K[n] > KMc^n, \quad \forall n \geq N(a,c,M) \qquad \square$$

We are now ready to prove the main theorem, which, in the context of LOCA, states that with the usage of each session as a uniform distribution of $\{1,...,m\}$, a bTelco's number of sessions as N_B and the total number of sessions a broker receives from all bTelcos as N_A, if $\frac{N_B}{N_A}$ meets certain requirements (depending on m), the expected number of session subsets that have the same aggregate usage as the bTelco's N_B sessions grows exponentially w.r.t the total number of sessions, N_A.
Next, we prove that a lower bound of this expected number, considering subsets consisting of only the remaining $N_A - N_B$ sessions, is already exponential w.r.t N_A:

Theorem B.6. *Considering two independent arrays X and Y, consisting of $N_A - N_B$ and N_B iid random variables from $U\{1,m\}$, there exists $L(m), U(m)$ such that the expected number of subsets X, whose sums are equal to the sum of Y, is exponential w.r.t N_A, if $L(m) \leq \frac{N_B}{N_A} \leq U(m)$. Note that $L(m), U(m)$ depend on m, and $0 < L(m) \leq U(m) < 1$, $\forall m \in Z_{>0}$*

Proof. We prove this theorem by showing a valid $L(m), U(m)$ pair. We denote this expected number as $E(m, N_A, N_B)$, and derive its closed-form expression by using the distributions of sums and subset sums, which were computed in Lemma B.1 and Lemma B.2. Specifically, for a random array B, the probability that its sum equals to y is $P_{N_B}(y)$, and the expected number of subsets in A that add to y is $2^{N_A-N_B}Q_{N_A-N_B}(y)$:

$$\begin{aligned}
E(m,N_A,N_B) &= 2^{N_A-N_B}\sum_{y=N_B}^{mN_B} P_{N_B}(y)Q_{N_A-N_B}(y) \\
&= 2^{N_A-N_B}\sum_{y=N_B}^{mN_B}(\frac{1}{m})^{N_B}\binom{N_B}{y-N_B}_{m-1}(\frac{1}{2m})^{N_A-N_B}\sum_{k=0}^{N_A-N_B}\binom{N_A-N_B}{k}m^k\binom{N_A-N_B-k}{y-(N_A-N_B-k)}_{m-1} \qquad (\text{B.1, B.2}) \\
&= (\frac{1}{m})^{N_A}\sum_{y=N_B}^{mN_B}\binom{N_B}{y-N_B}_{m-1}\sum_{k=0}^{N_A-N_B}\binom{N_A-N_B}{k}m^k\binom{N_A-N_B-k}{y-(N_A-N_B-k)}_{m-1} \\
&= (\frac{1}{m})^{N_A}\sum_{k=0}^{N_A-N_B}\binom{N_A-N_B}{k}m^k\sum_{y=N_B}^{mN_B}\binom{N_B}{y-N_B}_{m-1}\binom{N_A-N_B-k}{y-(N_A-N_B-k)}_{m-1}
\end{aligned}$$

By using identities of polynomial coefficients [42], we can transform the last term as:

$$\begin{aligned}
\sum_{y=N_B}^{mN_B}\binom{N_B}{y-N_B}_{m-1}\binom{N_A-N_B-k}{y-(N_A-N_B-k)}_{m-1} &= \sum_{y=N_B}^{mN_B}\binom{N_B}{(m-1)N_B-(y-N_B)}_{m-1}\binom{N_A-N_B-k}{y-(N_A-N_B-k)}_{m-1} && (\text{symmetry}) \\
&= \binom{[N_B]+[N_A-N_B-k]}{[(m-1)N_B-(y-N_B)]+[y-(N_A-N_B-k)]}_{m-1} && (\text{Vandermonde}) \\
&= \binom{N_A-k}{(m+1)N_B-N_A+k}_{m-1}
\end{aligned}$$

Therefore, we have a closed-form expression for $E(m,N_A,N_B)$:

$$E(m,N_A,N_B) = (\frac{1}{m})^{N_A}\sum_{k=0}^{N_A-N_B}\binom{N_A-N_B}{k}m^k\binom{N_A-k}{(m+1)N_B-N_A+k}_{m-1}$$

According to the definition of polynomial coefficients:

$$\binom{N_A-k}{(m+1)N_B-N_A+k}_{m-1} \geq 1 \text{ if } 0 \leq (m+1)N_B-N_A+k \leq (m-1)(N_A-k) \Leftrightarrow N_A-(m+1)N_B \leq k \leq N_A-(1+\frac{1}{m})N_B \tag{2}$$

Therefore, we could obtain a lower bound for E:

$$\begin{aligned} E(m,N_A,N_B) &\geq (\frac{1}{m})^{N_A}\left\{\sum_{k=0}^{N_A-N_B}\binom{N_A-N_B}{k}m^k - \sum_{k=0}^{N_A-(m+1)N_B-1}\binom{N_A-N_B}{k}m^k - \sum_{k=N_A-(1+\frac{1}{m})N_B+1}^{N_A-N_B}\binom{N_A-N_B}{k}m^k\right\} \\ &= (\frac{1}{m})^{N_A}\left\{(1+m)^{N_A-N_B} - H(N_A-(m+1)N_B-1;N_A-N_B,m) - T(N_A-(1+\frac{1}{m})N_B+1;N_A-N_B,m)\right\} \end{aligned}$$

According to Lemma B.3, we know that with

$$\frac{N_A-(m+1)N_B}{N_A-N_B} \leq \frac{m}{1+m} \quad \Longrightarrow \quad \frac{N_B}{N_A} \geq \frac{1}{m^2+m+1} \tag{3}$$

$$\frac{N_A-(1+\frac{1}{m})N_B}{N_A-N_B} \geq \frac{m}{1+m} \quad \Longrightarrow \quad \frac{N_B}{N_A} \leq \frac{m}{2m+1} \tag{4}$$

We have:

$$\begin{aligned} E(m,N_A,N_B) &\geq (\frac{1}{m})^{N_A}\left\{(1+m)^{N_A-N_B} - H(N_A-(m+1)N_B-1;N_A-N_B,m) - T(N_A-(1+\frac{1}{m})N_B+1;N_A-N_B,m)\right\} \\ &\geq (\frac{1}{m})^{N_A}\left\{(1+m)^{N_A-N_B} - (b_H)^{N_A-N_B} - (b_T)^{N_A-N_B}\right\} \qquad \text{with } b_H < (1+m), b_T < (1+m) \end{aligned}$$

Therefore, according to Lemma B.5, we can ignore the head and tail, and E has an exponential lower bound w.r.t N_A as long as the base of $(\frac{1}{m})^{N_A}(1+m)^{N_A-N_B}$ is larger than 1. Since:

$$(\frac{1}{m})^{N_A}(1+m)^{N_A-N_B} = \left[\frac{(1+m)^{1-\frac{N_B}{N_A}}}{m}\right]^{N_A}$$

to ensure base larger than 1, we need

$$\frac{(1+m)^{1-\frac{N_B}{N_A}}}{m} > 1 \quad \Longrightarrow \quad \frac{N_B}{N_A} < 1 - \frac{\ln(m)}{\ln(m+1)} \tag{5}$$

Combining constraints (3), (4) and (5), we show an exponential lower bound for $E(m,N_A,N_B)$ with the following $\frac{N_B}{N_A}$:

$$\frac{1}{m^2+m+1} \leq \frac{N_B}{N_A} < 1 - \frac{\ln(m)}{\ln(m+1)}$$

As proved in Lemma B.4, such a range is always valid for any $m \in Z_{>0}$. Therefore, we show that for $L(m) = \frac{1}{m^2+m+1}, U(m) = 1-\frac{\ln(m)}{\ln(m+1)}$, $E(m,N_A,N_B)$ has an exponential lower bound of $\left[\frac{(1+m)^{1-\frac{N_B}{N_A}}}{m}\right]^{N_A}$.

We believe one could achieve a much large exponential bounds and/or a wider range for feasible $\frac{N_B}{N_A}$ by carefully reexamining the two reductions that we made:

- Only the remaining N_A-N_B elements are considered for subsets that have the same sum. This is to simplify the problem so that we can treat it as a problem involving two independent arrays of length N_A-N_B and N_B. This, however, significantly underestimates the number of matched subsets, especially when $\frac{N_B}{N_A}$ is high.
- The reduction we made in (2): $\binom{N_A-k}{(m-1)N_B-N_A+k}_{m-1} \geq 1$ is obviously coarse grained. □

Lastly, we can prove the other side of the story: if $\frac{N_B}{N_A}$ is too large or too small, the expected number of subsets out of $N_A - N_B$ elements that have the same sum as the N_B elements *does not* grow exponentially w.r.t to N_A:

Theorem B.7. *Considering two independent arrays X and Y, consisting of $N_A - N_B$ and N_B iid random variables from $U\{1,m\}$, there exists $LL(m), UU(m)$ such that the expected number of subsets in X, whose sums are equal to the sum of Y, can not be exponential w.r.t N_A, if $\frac{N_B}{N_A} \leq LL(m)$ or $\frac{N_B}{N_A} \geq UU(m)$. Note that $0 < LL(m) \leq UU(m) < 1$, $\forall m \in Z_{>0}$*

Proof. Firstly, we show that such a $UU(m)$ exists. We start with the closed-form expression of $E(m,N_A,N_B)$ that is derived in the proof above.

$$
\begin{aligned}
E(m,N_A,N_B) &= (\frac{1}{m})^{N_A} \sum_{k=N_A-(m+1)N_B}^{N_A-(1+\frac{1}{m})N_B} \binom{N_A-N_B}{k} m^k \binom{N_A-k}{(m+1)N_B-N_A+k}_{m-1} \\
&\leq (\frac{1}{m})^{N_A} \sum_{k=0}^{N_A-(1+\frac{1}{m})N_B} \binom{N_A-N_B}{k} m^k \binom{N_A-k}{(m+1)N_B-N_A+k}_{m-1} \\
&\leq (\frac{1}{m})^{N_A} \sum_{k=0}^{N_A-(1+\frac{1}{m})N_B} \binom{N_A-N_B}{k} m^k \binom{N_A}{(m+1)N_B-N_A+k}_{m-1} \qquad (\binom{n}{k}_m \geq \binom{n-\Delta}{k}_m \ \forall \Delta \in Z_{\geq 0})
\end{aligned}
$$

As noted above $\binom{n}{k}_m$ is a non-increasing function of k for $\lceil \frac{mn}{2} \rceil \leq k \leq mn$ [42], therefore, if we have

$$
(m+1)N_B - N_A \geq \lceil \frac{(m-1)N_A}{2} \rceil \quad \Longrightarrow \quad \frac{N_B}{N_A} > \frac{1}{2}
$$

$\binom{N_A}{(m+1)N_B-N_A+k}_{m-1}$ decreases as k increases from 0 to $N_A - (1+\frac{1}{m})N_B$, thus:

$$
\begin{aligned}
\binom{N_A}{(m+1)N_B-N_A+k}_{m-1} &\leq \binom{N_A}{(m+1)N_B-N_A}_{m-1} \qquad (\text{for } k \in \{0,..,N_A-(1+\frac{1}{m})N_B\}) \\
&= d^{N_A}(m,N_A,\frac{N_B}{N_A}) \qquad (\text{with } d(m,N_A,\frac{N_B}{N_A}) = \left[\binom{N_A}{(m+1)N_B-N_A}_{m-1}\right]^{\frac{1}{N_A}}) \qquad (6)
\end{aligned}
$$

where:

$$
d(m,N_A,\frac{N_B}{N_A}) = \left[\binom{N_A}{(m+1)N_B-N_A}_{m-1}\right]^{\frac{1}{N_A}} = \left[\binom{N_A}{[(m+1)\frac{N_B}{N_A}-1]N_A}_{m-1}\right]^{\frac{1}{N_A}}
$$

We denote

$$
f(m,\frac{N_B}{N_A}) = \lim_{N_A \to \infty} d(m,N_A,\frac{N_B}{N_A})
$$

There are two properties of $g(m,\frac{N_B}{N_A})$ that we leverage here: (i) $1 \leq f(m,\frac{N_B}{N_A}) \leq m$, this is because $\binom{N_A}{[(m+1)\frac{N_B}{N_A}-1]N_A}_{m-1} < m^{N_A}$ and $\binom{N_A}{[(m+1)\frac{N_B}{N_A}-1]N_A}_{m-1}$ is a non-decreasing function w.r.t N_A; (ii) $f(m,\frac{N_B}{N_A})$ is a non-increasing function w.r.t $\frac{N_B}{N_A}$ for $\frac{1}{2} < \frac{N_B}{N_A} < \frac{m}{m+1}$, this is because $d(m,N_A,\frac{N_B}{N_A})$ is a decreasing function w.r.t $\frac{N_B}{N_A}$. Moreover, $f(m,\frac{m}{m+1}) = 1$ and $f(m,\frac{1}{2}) = m$. The later is because, as shown in [37], $\binom{n}{\frac{m}{2}n}_m \sim \frac{(m+1)^n}{\sqrt{2\pi n \frac{m(m+2)}{12}}}$ as $n \to \infty$, which indicates that $\lim_{n\to\infty}[\binom{n}{\frac{m}{2}n}_m]^{\frac{1}{n}} = m+1$. Therefore, $f(m,\frac{1}{2}) = \lim_{N_A\to\infty}[\binom{N_A}{\frac{m-1}{2}N_A}_{m-1}]^{\frac{1}{N_A}} = m$. With (6), we now have the following upper bound for $E(m,N_A,N_B)$ (to simply notation, we use d for $d(m,N_A,\frac{N_B}{N_A})$):

$$
E(m,N_A,N_B) \leq (\frac{d}{m})^{N_A} \sum_{k=0}^{N_A-(1+\frac{1}{m})N_B} \binom{N_A-N_B}{k} m^k
$$

Similar to Lemma B.3, we can then bound the term $\sum_{k=0}^{N_A-(1+\frac{1}{m})N_B}\binom{N_A-N_B}{k}m^k$ by using tail bounds of binomial distribution. Specifically, as mentioned in the proof of Lemma B.3:

$$\sum_{k=0}^{N_A-(1+\frac{1}{m})N_B}\binom{N_A-N_B}{k}m^k \leq \{(1+m)\exp[-D(\frac{N_A-(1+\frac{1}{m})N_B}{N_A-N_B}||\frac{m}{1+m})]\}^{N_A-N_B}$$
$$\text{if} \quad \frac{N_A-(1+\frac{1}{m})N_B}{N_A-N_B} \leq \frac{m}{1+m} \quad \Rightarrow \quad \frac{N_B}{N_A} \geq \frac{m}{2m+1} \tag{7}$$

where $D(a||p)$ is the relative entropy between a $Bernoulli(a)$ (a-coin) and a $Bernoulli(p)$ (p-coin): $D(a||p) = (a)\log\frac{a}{p} + (1-a)\log\frac{1-a}{1-p}$. Here we denote

$$g(m,\frac{N_B}{N_A}) = (1+m)\exp[-D(\frac{N_A-(1+\frac{1}{m})N_B}{N_A-N_B}||\frac{m}{1+m})]$$
$$= (1+m)\exp[-D(\frac{1-(1+\frac{1}{m})\frac{N_B}{N_A}}{1-\frac{N_B}{N_A}}||\frac{m}{1+m}))]$$

Note that $g(m,\frac{N_B}{N_A})$ is a decreasing function of $\frac{N_B}{N_A}$ for $\frac{N_B}{N_A} \geq \frac{m}{2m+1}$. Moreover, $\lim_{\frac{N_B}{N_A}\to\frac{m}{m+1}} g(m,\frac{N_B}{N_A}) \to 0$, and $g(m,\frac{1}{2}) > 1$ for $m \geq 2$. The later is because $g(m,\frac{1}{2}) = (1+m)\exp[-D(\frac{m-1}{m}||\frac{m}{1+m}))]$ is an increasing function with m, thus $g(m,\frac{1}{2}) \geq g(2,\frac{1}{2}) = 3\exp[-D(\frac{1}{2}||\frac{2}{3}))] > 1$.
With (7), we now have:

$$E(m,N_A,N_B) \leq (\frac{d}{m})^{N_A}[g(m,\frac{N_B}{N_A})]^{N_A-N_B}$$
$$= \left\{(\frac{d}{m})[g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}}\right\}^{N_A}$$

Considering the limit of the base:

$$\lim_{N_A\to\infty}[\frac{d(m,N_A,\frac{N_B}{N_A})}{m}][g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}}$$
$$= [g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}}\lim_{N_A\to\infty}[\frac{d(m,N_A,\frac{N_B}{N_A})}{m}]$$
$$= [g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}}[\frac{f(m,\frac{N_B}{N_A})}{m}]$$

With the properties of $f(m,\frac{N_B}{N_A})$ and $g(m,\frac{N_B}{N_A})$ that we mentioned, one could prove that there exists a ratio $R(m)$ such that:

$$h(m,\frac{N_B}{N_A}) = [g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}}[\frac{f(m,\frac{N_B}{N_A})}{m}] \leq 1 \quad \forall \quad \frac{N_B}{N_A} \geq R(m)$$
$$\text{where } h(m,R(m)) = 1$$

This means that $E(m,N_A,N_B)$ is not exponential w.r.t N_A for $\frac{N_A}{N_B} \geq R(m)$
We could prove that there is a unique $R(m)$ with $h(m,R(m)) = 1$, and $\frac{m}{2m+1} < R(m) < \frac{m}{m+1}$, with the following properties of $f(m,\frac{N_B}{N_A})$ and $g(m,\frac{N_B}{N_A})$: (i) both of them are decreasing functions w.r.t $\frac{N_B}{N_A}$; (ii) $1 \leq f(m,\frac{N_B}{N_A}) \leq m$, $f(m,\frac{m}{m+1}) = 1$ and $f(m,\frac{1}{2}) = m$; (iii) $g(m,\frac{1}{2}) > 1$, and $\lim_{\frac{N_B}{N_A}\to\frac{m}{m+1}} g(m,\frac{N_B}{N_A}) \to 0$.
Firstly, because $\frac{f(m,\frac{N_B}{N_A})}{m} \leq 1$, for $\frac{N_B}{N_A}$ such that $g(m,\frac{N_B}{N_A}) < 1 \Rightarrow [g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}} < 1$, we have
$h(m,\frac{N_B}{N_A}) = [g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}}[\frac{f(m,\frac{N_B}{N_A})}{m}] < 1$. Moreover, since $\lim_{\frac{N_B}{N_A}\to\frac{m}{m+1}} g(m,\frac{N_B}{N_A}) \to 0$, we know that $R(m) < \frac{m}{m+1}$

Secondly, for $\frac{N_B}{N_A}$ such that $g(m,\frac{N_B}{N_A})>1$, we have $h(m,\frac{N_B}{N_A})$ as a decreasing function w.r.t $\frac{N_B}{N_A}$. Moreover, since $g(m,\frac{1}{2})>1$, $f(m,\frac{1}{2})=m$, we have $h(m,\frac{1}{2})=[g(m,\frac{1}{2})]^{1-\frac{1}{2}}[\frac{f(m,\frac{1}{2})}{m}]>1$. Therefore, we know that $R(m)>\frac{1}{2}$.

Therefore, we have proved a valid $UU(m)$: $\frac{1}{2}<UU(m)=R(m)<\frac{m}{m+1}$, where $R(m)$ is defined as:

$$
\begin{aligned}
R(m) &= \arg_{\frac{1}{2}<\frac{N_B}{N_A}<\frac{m}{m+1}} h(m,\frac{N_B}{N_A})=1 \\
\text{where } h(m,\frac{N_B}{N_A}) &= [g(m,\frac{N_B}{N_A})]^{1-\frac{N_B}{N_A}}[\frac{f(m,\frac{N_B}{N_A})}{m}] \\
f(m,\frac{N_B}{N_A}) &= \lim_{N_A\to\infty}\left[\binom{N_A}{[(m+1)\frac{N_B}{N_A}-1]N_A}_{m-1}\right]^{\frac{1}{N_A}} \\
g(m,\frac{N_B}{N_A}) &= (1+m)\exp[-D(\frac{1-(1+\frac{1}{m})\frac{N_B}{N_A}}{1-\frac{N_B}{N_A}}||\frac{m}{1+m})]
\end{aligned}
$$

and $E(m,N_A,N_B)$ is not exponential if $\frac{N_B}{N_A}\geq UU(m)$.
Next, we show a valid $LL(m)$ by using a similar approach. We use E to represent $E(m,N_A,N_B)$ in the follows.

$$
\begin{aligned}
E &\leq (\frac{1}{m})^{N_A}\sum_{k=N_A-(1+m)N_B}^{N_A-N_B}\binom{N_A-N_B}{k}m^k\binom{N_A-k}{(m+1)N_B-N_A+k}_{m-1} \\
&\leq (\frac{1}{m})^{N_A}\sum_{k=N_A-(1+m)N_B}^{N_A-N_B}\binom{N_A-N_B}{k}m^k\binom{(m+1)N_B}{(m+1)N_B-N_A+k}_{m-1} && (\binom{n}{k}_m\geq\binom{n-\Delta}{k}_m\ \forall\Delta\in Z_{\geq 0}) \\
&< (\frac{1}{m})^{N_A}m^{(m+1)N_B}\sum_{k=N_A-(1+m)N_B}^{N_A-N_B}\binom{N_A-N_B}{k}m^k && (\binom{(m+1)N_B}{(m+1)N_B-N_A+k}_{m-1}<m^{(m+1)N_B}) \\
&\leq (\frac{1}{m})^{N_A}m^{(m+1)N_B}\left\{(1+m)\exp[-D(\frac{N_A-(1+m)N_B}{N_A-N_B}||\frac{m}{1+m})]\right\}^{N_A-N_B} && (\frac{N_A-(m+1)N_B}{N_A-N_B}\geq\frac{m}{1+m}\Rightarrow\frac{N_B}{N_A}\leq\frac{1}{m^2+m+1}) \\
&= \left\{m^{(m+1)\frac{N_B}{N_A}-1}\left\{(1+m)\exp[-D(\frac{1-(1+m)\frac{N_B}{N_A}}{1-\frac{N_B}{N_A}}||\frac{m}{1+m})]\right\}^{1-\frac{N_B}{N_A}}\right\}^{N_A}
\end{aligned}
$$

We could prove that for $l(m,\frac{N_B}{N_A})=m^{(m+1)\frac{N_B}{N_A}-1}\left\{(1+m)\exp[-D(\frac{1-(1+m)\frac{N_B}{N_A}}{1-\frac{N_B}{N_A}}||\frac{m}{1+m})]\right\}^{1-\frac{N_B}{N_A}}$, there exists a $0<S(m)<\frac{1}{m^2+m+1}$, which is the $LL(m)$ that we try to prove, that $l(m,\frac{N_B}{N_A})\leq 1$ for any $\frac{N_B}{N_A}\leq S(m)$. This is because (i) $l(m,\frac{N_B}{N_A})$ is an increasing function of $\frac{N_B}{N_A}$, and (ii) $l(m,\frac{1}{m^2+m+1})=m^{-\frac{m^2}{m^2+m+1}}(1+m)^{\frac{m^2+m}{m^2+m+1}}>(1+m)^{\frac{m}{m^2+m+1}}>1$ and $\lim_{\frac{N_B}{N_A}\to 0}l(m,\frac{N_B}{N_A})=0<1$.
Therefore, we have proved a valid $LL(m)$: $0<LL(m)=S(m)<\frac{1}{m^2+m+1}$, where $S(m)$ is defined as:

$$
\begin{aligned}
S(m) &= \arg_{0<\frac{N_B}{N_A}<\frac{1}{m^2+m+1}} l(m,\frac{N_B}{N_A})=1 \\
\text{where } l(m,\frac{N_B}{N_A}) &= m^{(m+1)\frac{N_B}{N_A}-1}\left\{(1+m)\exp[-D(\frac{1-(1+m)\frac{N_B}{N_A}}{1-\frac{N_B}{N_A}}||\frac{m}{1+m})]\right\}^{1-\frac{N_B}{N_A}}
\end{aligned}
$$

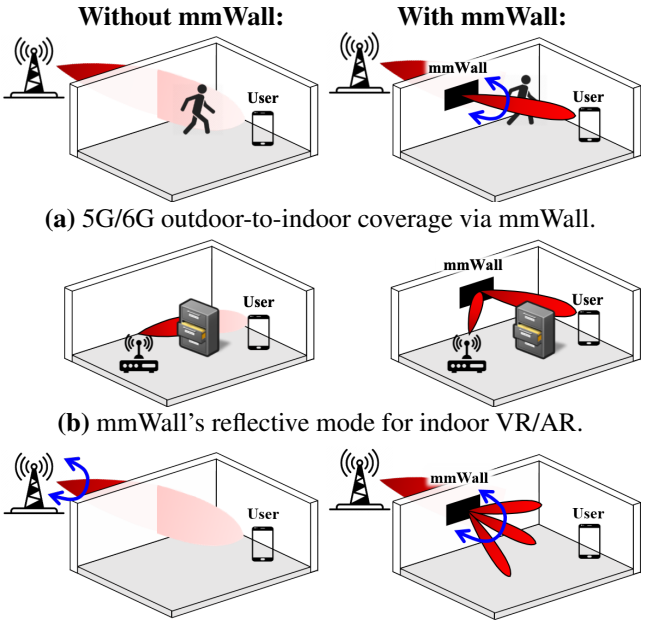
and $E(m,N_A,N_B)$ is not exponential if $\frac{N_B}{N_A}\leq LL(m)$. □

mmWall: A Steerable, Transflective Metamaterial Surface for NextG mmWave Networks

Kun Woo Cho[1], Mohammad H. Mazaheri[3], Jeremy Gummeson[2], Omid Abari[3], Kyle Jamieson[1]
Princeton Univ.[1]*, Univ. of Massachusetts Amherst*[2]*, UCLA*[3]

Abstract

Mobile operators are poised to leverage millimeter wave technology as 5G evolves, but despite efforts to bolster their reliability indoors and outdoors, mmWave links remain vulnerable to blockage by walls, people, and obstacles. Further, there is significant interest in bringing outdoor mmWave coverage indoors, which for similar reasons remains challenging today. This paper presents the design, hardware implementation, and experimental evaluation of *mmWall*, the first electronically almost-360° steerable metamaterial surface that operates above 24 GHz and both refracts or reflects incoming mmWave transmissions. Our metamaterial design consists of arrays of varactor-split ring resonator unit cells, miniaturized for mmWave. Custom control circuitry drives each resonator, overcoming coupling challenges that arise at scale. Leveraging beam steering algorithms, we integrate mmWall into the link layer discovery protocols of common mmWave networks. We have fabricated a 10 cm by 20 cm mmWall prototype consisting of a 28 by 76 unit cell array and evaluated it in indoor, outdoor-to-indoor, and multi-beam scenarios. Indoors, mmWall guarantees 91% of locations outage-free under 128-QAM mmWave data rates and boosts SNR by up to 15 dB. Outdoors, mmWall reduces the probability of complete link failure by a ratio of up to 40% under 0–80% path blockage and boosts SNR by up to 30 dB.



(a) 5G/6G outdoor-to-indoor coverage via mmWall.

(b) mmWall's reflective mode for indoor VR/AR.

(c) mmWall's beam splitting, for link establishment.

Figure 1: mmWall re-focuses outdoor coverage indoors towards the user and potentially around obstacles, provides path diversity indoors by reflection, and splits an incoming beam for fast link establishment.

1 Introduction

Millimeter-wave (mmWave) spectrum has emerged in the 5G/6G era as a key next generation wireless network enabler, fulfilling user demands for high spectral efficiency and low latency wireless networks. Higher carrier frequencies offer greater network capacity: for instance, the maximum carrier frequency of the 4G LTE band at 2.4 GHz provides an available spectrum bandwidth of only 100 MHz, while mmWave (above 24 GHz) can easily hold spectral bandwidths five to ten times greater, enabling multi-Gbit/sec data rates. Hence, mmWave spectrum enables a plethora of mobile applications that are currently infeasible due to their requirements of very high data rates, such as virtual and augmented reality (VR/-AR), camera-based purchase tracking in smart stores, and robotic automation in smart warehouses.

mmWave technology faces significant headwinds, however, in at least three key scenarios:

1. 5G outdoor coverage is difficult to bring indoors, as exterior building walls block mmWave signal, as do outdoor windows' tinted glass (Fig. 1(a)). Attenuation at 28 GHz is *ca.* 40 dB versus 4 dB through indoor glass [44], as outdoor metalized glass coatings attenuate by 25–50 dB per layer [35]. Currently, operators are forced to offload

mmWave traffic onto lower frequencies or off their networks entirely (Wi-Fi) when users move indoors, incurring handover delay and application disruptions.

2. Indoors, people, furniture, doors, and other clutter block mmWave (Fig. 1(b)), forcing data to flow over a much less reliable reflection path. Indeed, in an extensive indoor measurement campaign at 28 GHz, MacCartney *et al.* observe a close-in best non-line of sight path loss exponent *ca.* 3, with a normally-distributed additional loss with an 11 dB variance [22]. While the resulting temporary outages are common, highly demanding applications like VR/AR streaming cannot tolerate these glitches.
3. Third, NextG cellular providers face challenges in adopting mmWave frequencies outdoors for primary service as well as wireless backhaul because mmWave signals are readily absorbed by foliage, and reflection off buildings is largely specular, constraining the angle of reflection to be equal to the angle of incidence. Measurements in New York City highlight this issue: 28 GHz data shows most links greater than 200 meters in outage [3].

This paper describes the design and implementation of ***mmWall***, an electronically reconfigurable surface that addresses all three foregoing use cases, also shown in Figure 1. Like much prior work (§2), mmWall leverages *metamaterials*, artificial composite materials engineered at a sub-wavelength scale to exhibit unique electromagnetic properties that do not exist in naturally occurring materials [17]. But mmWall is the first practical work to our knowledge to use a specific class of metamaterials capable of refracting incoming radiation with (theoretically) no loss: *Huygens* metamaterials [9, 25]. mmWall is a reconfigurable intelligent surface that uses a novel Huygens metasurface (HMS) metamaterial to reflect/refract and precisely steer incoming mmWave beams towards desired directions, thus enhancing path diversity for mmWave networks. Work has shown that surfaces that can steer incoming mmWave transmissions in this way have the potential to dramatically improve spatial multiplexing [41] and spectral efficiency [39] of networks as a whole. Hence when obstacles like a human body or outdoor foliage blocks the line of sight (LoS) or non line of sight (NLoS) paths, mmWall can often provide an alternative path that is not a simple reflection or a straight-line transmission, and hence would not otherwise exist. In the first scenario, mmWall can refract mmWave signals from outdoors to steer them directly towards an indoor receiver, making outdoor to indoor communication possible. In the second scenario above, mmWall reflects mmWave beams at non-specular angles (those for which the angle of reflection is not equal to the angle of incidence). And in the third scenario, mmWall can reflect outdoor transmissions at non-specular angles, ameliorating outdoor blockages.

mmWall is electronically reconfigurable to either reflect or refract incoming energy, allowing it to time-multiplex the different roles of each of the three above use cases without human intervention, while installed in a fixed location. Also, its multi-beam functionality (Fig. 1(c)) enables fast beam search, and support for multiple users at the same time. mmWall has no RF chain, and its electric components draw only a couple-of-hundred microwatts order of power. Consequently, it consumes much lower power compared to a conventional AP that necessitates multiple RF chains for multi-beam operations. To our knowledge, mmWall is the first surface able to achieve near-360° angular coverage (§5).

This work addresses several hardware and software design challenges that arise in the realization of such a design. Since mmWave transmissions are "pencil-beam" in nature, they work only when the transmitter's beam is perfectly aligned with the receiver's beam. To correctly steer the beam towards the receiver, we design a metamaterials-based surface that can precisely control the phases of the incoming signal, focusing signal power in a narrow beam. Secondly, since the size of meta-atom scales with its operating frequency, mmWall's meta-atoms are much smaller than the conventional antennas and therefore extremely sensitive to coupling. Hence, we not only scale the surface to mmWave frequency but also deliberately design the control lines to avoid undesirable coupling. Lastly, existing systems uses their own beam searching protocol to find the best alignment. To make mmWall compatible with different mmWave applications, we design an effective beam alignment protocol that leaves the existing systems unchanged [16].

Contributions and Results. mmWall is the first design that can arbitrarily reflect, refract, and split the mmWave beam in a nearly lossless manner. We analyze our meta-atom designs and compare them with simulation results, allowing our designs to scale to different frequencies for potential applications like Terahertz communication. To the best of our knowledge, this is the first study that theoretically analyzes and builds a working prototype of a reconfigurable Huygens metasurface at mmWave frequency. We have designed and implemented mmWall hardware with a novel control network in custom PCB, and in §5, evaluate its performance through experiments in environments matching the scenarios, we outline above. Our empirical results show that when both the AP and the client are in the same room, we can provide an SNR of 25 dB or more for all locations in a $10 \times 8m$ room, using a single mmWall surface. This SNR is sufficient to support 128-QAM in 91% of locations. Moreover, the SNR improves to 30 and 35 dB when we place two surfaces, respectively, on different walls. Finally, we show the effectiveness of mmWall in bringing outdoor mmWave networks indoors. When the AP is 6 meters away from the building, mmWall improves the SNR by up to 30 dB, providing an SNR of 20 dB or more in all locations in a room using a single surface placed on a window.

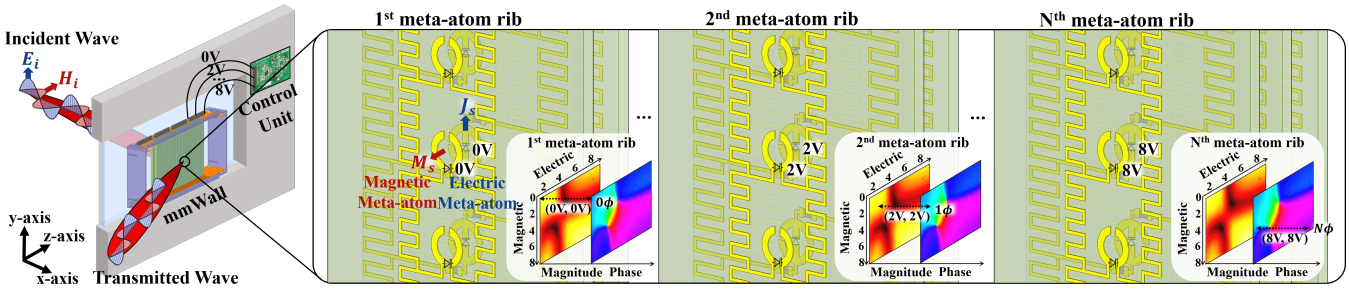


Figure 2: mmWall's design converts an incident mmWave beam to a refracted (or reflected, not shown) beam via field discontinuities created by current in its resonators. *Inset:* magnetic meta-atoms are shown in front of the electric meta-atoms.

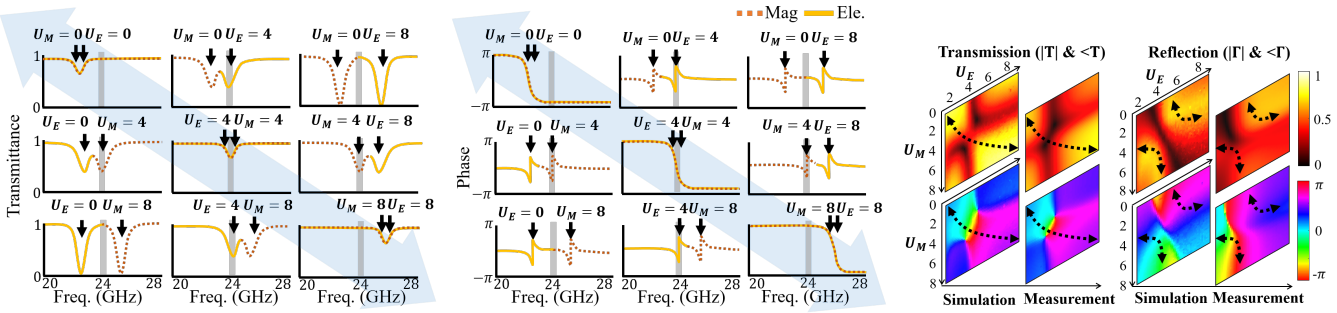
2 Related Work

HMSs comprise a layer of co-located *magnetic* and *electric* meta-atom, etched onto the two respective sides of a dielectric substrate (Fig. 2, *inset*). The magnetic meta-atom is an enclosed metallic ring with one split, while the electric meta-atom has two splits and a metal strip in the center (Fig. 4(b)). As the incident wave passes through the magnetic meta-atom, the wave's magnetic field H_i induces a rotating current (green arrows in Fig. 4(b), *upper*) on the magnetic meta-atom that, in turn, creates a magnetic response ($\vec{M}_s$ along the z-axis in Fig. 2). Likewise, the electric meta-atom is excited by the wave's electric field $\vec{E}_i$, resulting in two symmetric, oscillating current loops (green arrows in Fig. 4(b), *lower*) that create an electric response ($\vec{J}_s$ along the y-axis in Fig. 2). These responses interact with the wave's fields, causing an abrupt phase shift. By varying the applied voltage to a tunable component loaded on each meta-atom, the surface precisely controls the responses, thereby allowing any phase shift from 0° to 360° with near-unity transmission and/or reflection.

Prior work in passive HMSs [8,9,29,40] has demonstrated a "lensing" effect and negative refraction index [29] and the engineering of complex beam patterns [8]. However, they lack the capability to reconfigure and both refract and reflect the beam. Prior work in actively-controlled HMSs [4,7,21,38,43] uses varactors or PIN diodes to tune each element in a continuous or binary (*i.e.*, on-off) manner, respectively. Such devices can shift signals' frequency [21] and polarization [5,38]. While these designs have shown great promise in theoretical prediction models [23] and/or at low frequencies [36], they do not scale to higher mmWave frequencies in a straightforward way, due to a mismatch between the required meta-atom size and a varactor's size, and the attenuation that commonly available substrate would induce on an incident mmWave signal. Scaling these designs also requires narrower trace widths that are hard to fabricate and prone to breaking during diode soldering. More importantly, they focus on steering one beam in a one-sided direction, rather than steering one or more beams in a reflective and/or transmissive direction. mmWall is the first mmWave work to do so. Evaluation efforts in this group of prior work stop short of realistic end-to-end experiments.

Work in actively-controlled mmWave Reconfigurable Intelligent Surfaces (RISs) includes a solely reflective, PIN-diode based surface at 2.3 and 28 GHz [7], whose evaluation at 28 GHz states a gain of 19 dBi, but which stops short of further experimental evaluation of steerability or any further end-to-end evaluation at 28 GHz. Tang *et al.* describe similar PIN-diode, reflective surfaces at 27 and 33 GHz, model path losses in such scenarios, and experimentally evaluate [33]. Tan *et al.* consider a similar design at 60 GHz [31], but neither consider HMS-based designs such as mmWall's, which can shift between reflective (on both sides of the surface) and transmissive modes instantly via electronic control. Existing reflective RISs only reflect on one side, while mmWall performs both indoor *and* outdoor non-specular reflections from a fixed location. In press releases ([a], [b], [c]) NTT DoCoMo describe reflective, outdoor-to-indoor surfaces operating at 28 GHz. They state top line experimental results, but do not disclose design details or details of their experimental evaluation. Other work uses split ring resonators as antennas for a Massive MIMO base station [28], a related but distinct application to mmWall. This paper is an extension of the authors' previous workshop publication [6] that describes a new control line design, documents real hardware implementation, and presents significant new evaluation results in realistic, diverse scenarios.

Recent work in passive non-HMS based mmWave RISs includes proposals that reflect signals at angles of reflection different than incidence [11,26], but cannot be tuned to target a receiver's location, hence wasting incident energy and resulting in at most 10 dB of gain, significantly below mmWall's achieved gain. Also, these approaches do not refract as mmWall can, yielding reduced applicability. Recent amplify-and-forward proposals for Wi-Fi [42] use a mesh topology, but do not scale to mmWave frequencies, and at mmWave [1] are limited to indoor reflection. Recent complementary approaches leverage multi-beam transmission [18,19], sensing and leveraging ambient reflectors [37], and use Wi-Fi as a control plane to discover mmWave links [20,30]. While they align with mmWall's goals, such approaches cannot create paths whose reflection angles diverge from their incident angles, or refract through a surface.



(a) Transmission magnitude $|T|$ **(b)** Transmission phase shift $\angle T$ **(c)** T and Γ magnitude, phase at 24 GHz.

Figure 3: Unit cell response *v.* electric- and magnetic-side control voltages U_E and U_M—**(a):** magnitude and **(b):** phase. **(c):** HFSS simulation (*left*) and near-field, real world VNA measurement (*right*)— arrows indicate control voltage pairs that yield a 360° phase shift of the incoming signal, with high transmission or reflection magnitude.

3 Design

We describe in turn mmWall's hardware (§3.1), their control mechanism (§3.2), and their link layer integration (§3.3).

3.1 Surface Hardware

mmWall's unit cells (also known as *meta-atoms*) are stacked vertically with a $\lambda/3$ separation, on each Rogers substrate board (also known as a *meta-atom rib*), as shown in Fig. 2 (see §3.2.1 for a discussion of vertical and horizontal unit cell spacing considerations for beamforming). A control unit connected to mmWall provides a set of voltages to the ribs. In Fig. 2, 0V is applied to both magnetic and electric meta-atoms on the first rib, causing the meta-atoms to shift the phase by 0ϕ with minimal loss. For the second rib, 2V is applied to shift the phase by 1ϕ. Ultimately, the beam is steered by all N ribs collectively forming an array factor.

3.1.1 Design Goals

The two primary design goals of the unit cell are to simultaneously **1)** achieve transmission T or reflection Γ loss levels as close to zero as possible, and **2)** effect any phase shift in $[0, 2\pi]$ on the incoming signal, both at mmWave frequencies.

The unit cell consists of two meta-atoms, *magnetic* and *electric*. The magnetic (electric) meta-atom induces a magnetic (electric) field response to the incoming signal that can resonate at different, tunable frequencies by varying the applied voltage to the varactor of the magnetic- (electric-) meta-atom.

Without loss of generality, we now describe how transmission works (reflection is fully complementary to transmission, and we refer the reader to Appendix A for a rigorous mathematical exposition of both). In Fig. 3(a), we observe that increasing the voltage applied to the magnetic meta-atom U_M from 0 to 8 V (down the three leftmost subplots) shifts its resonance frequency (lowest transmission magnitude point of the red dotted line[1]) to the right (we will analyze how this frequency shifting works in §3.1.2). Similarly, the electric meta-atom induces an electric response and its resonant frequency can be shifted by its own varactor (reading similarly across the three topmost subplots). Together their effects are superposed and we manipulate the collective magneto-electric response that interferes with the incident plane wave.

The key characteristic that allows near-perfect amplitude with full phase coverage appears when the two responses overlap at the same frequency. Otherwise, the phase response undergoes a sharp change of only π and its magnitude dips to nearly zero at its resonant frequency, as we see in Fig. 3(a) and Fig. 3(b) when the voltages applied to the magnetic and electric meta-atom differ by 8 V. However, as the two resonances start to overlap, transmission loss decreases and the phase shift becomes 2π (on-diagonal sub-figures, Fig. 3(a) and Fig. 3(b)). As a result, we achieve 2π phase coverage with near-unity magnitude by increasing the voltage applied to both the magnetic and electric meta-atoms together (Fig. 3(c), at control voltages indicated by the black curves).

While the overlapped resonances can reach a perfect unitary transmission magnitude in theory, the Huygens pattern from our measurement shows a lower transmission magnitude on the area where abrupt phase shifts occur due to various reasons, including the sensitivity at mmWave frequency, fabrication loss, and measurement errors.

3.1.2 Design Process

We now describe challenges we overcame in scaling the resonance of the mmWall unit cell to mmWave frequencies. By definition, the meta-atom behaves as an *LC* circuit with resonant frequency $1/\left(2\pi\sqrt{LC}\right)$, determined by the capacitance or inductance of the meta-atoms. Hence, we must markedly *decrease* the inductance and capacitance of prior microwave designs (§2), if we can hope to achieve a mmWave reso-

[1] In operation we largely avoid the lowest transmission nulls.

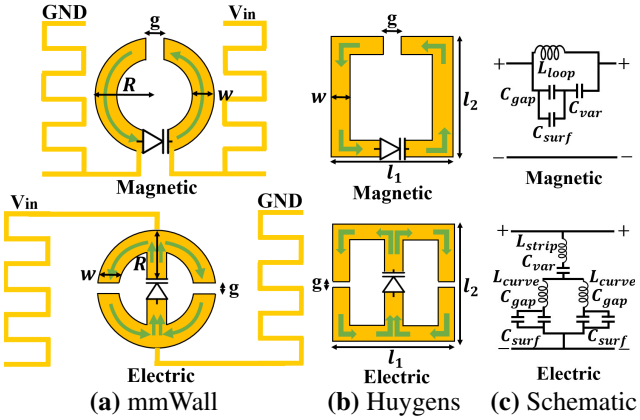


Figure 4: mmWall, prior Huygens unit cell designs (*top:* magnetic; *bottom:* electric side), and equivalent circuits. Green arrows indicate the oscillating current loops, and V_{in} indicates where the input voltages connect.

nant frequency. As we will see next, the smaller the ring is, the higher the resonant frequency becomes. However, the state-of-the-art approach to scale the frequency of a Huygens resonator (Fig. 4(b)) requires a loop width l_1 and loop height l_2 of $\lambda/10$. At mmWave, however, the varactor packaging itself would significantly distort the tailored electromagnetic surface properties when a meta-atom is sized $\lambda/10$, and so the straightforward approach fails. Moreover, the varactor is soldered with heat, causing tighter designs to become more fragile. Changing the rectangular cell shape to a circular one with equal diameter reduces size while preserving varactor placement on the diameter.

We thus instead adopt the design shown in Fig. 4(a), but this is only tenable with a careful tradeoff of meta-atom design parameters *radius R*, *trace width w*, and *trace gap width g* (*cf.* Fig. 4) as we next describe.

Magnetic meta-atom. Fig. 4(a) (*upper*) shows the design parameters that determine inductance L_m and capacitance C_m. L_m (= L_{loop}, the inductance of the physical conductor loop), is largely proportional to R (also $L_{\text{loop}} \propto t^{-1}$, w^{-1}, and g^{-1}). C_m consists of three capacitance values, C_{gap}, C_{surf}, and C_{var}:

$$C_m = \left(\frac{1}{C_{\text{gap}} + C_{\text{surf}}} + \frac{1}{C_{var}} \right)^{-1} \quad (1)$$

Here, C_{gap} is the parallel-plate capacitance induced by the gap in the ring ($\propto g^{-1}$), C_{surf} is a capacitance induced by the metallic surface ($\propto R$ [34]), and C_{var} is the capacitance of the varactor, a voltage-dependent capacitor. While L_{loop}, C_{gap}, and C_{surf} are fixed after fabrication, C_{var} varies with control voltage. Increasing U_M decreases C_{var} (see Fig. 17 in Appendix A for the precise relationship), and thus C_m (Eq. (1)), which in turn increases the resonance frequency, as depicted in Fig. 3.

When tuning the physical loop design parameters, we fix

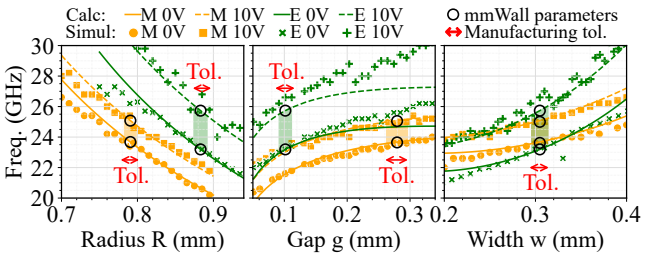


Figure 5: mmWall design parameter sensitivity analysis.

$C_{\text{var}} = 4$ V for both the magnetic and electric meta-atoms since at that voltage, the resonant frequency is at our desired mmWave frequency and an abrupt phase change occurs. Fig. 5 shows our chosen design parameters (denoted with black circles) and its corresponding magnetic side resonant frequency when $U_M = 0, 10$ V. Calculated (curves) and simulated (markers) data in our sensitivity analysis show that among all feature dimensions, decreasing R, followed by increasing g has the greatest effect on increasing resonant frequency for the magnetic meta-atom. We note that after fixing our meta-atom geometry as shown in the figure, 24 GHz lies in the middle of the resulting resonant frequency range. Also, we observe that PCB manufacturing tolerance (±5%) does not greatly shift the resonant frequency (we refer Appendix C for meta-atom sensitivity analysis against fabrication tolerance).

Electric meta-atom. Fig. 4(a) (*lower*) shows the electric meta-atom, in which current oscillates in two different directions, while the current of the magnetic meta-atom oscillates in one direction only (*cf.* green arrows in Figs. 4(a) and 4(b)). Hence, we analyze its inductance L_e as the combination of the inductances of the half-circular loop on the left half (L_{curve}), the inductance of the other half on the right half (L_{curve}, by symmetry), and the inductance from the metallic strip shared by two loops (L_{strip}). Since the two half-loops are arranged in parallel, with the metallic strip arranged in series, $L_e = (L_{curve}/2) + L_{strip}$ [11]. Since inductance generally depends on the surface area of the copper trace, $L_{curve} \propto R$, and $L_{strip} \propto w^{-1}$, L_e largely depends on both R and w, but not g. We see the impact of w on the resonant frequency in Fig. 5: compared to magnetic meta-atom, the resonant frequency of the electric meta-atom increases steeply as w increases due to L_{strip}. To minimize the difference in resonant frequencies between the electric and magnetic sides as desired, Fig. 5 guides us to design an electric meta-atom with equal w as the magnetic meta-atom, greater R and lesser g. The electric meta-atom has two gaps and two surface capacitances, with respective associated capacitances C_{gap} and C_{surf}, all in parallel, and that combination in series with C_{var}:

$$C_e = \left(\frac{1}{2(C_{\text{gap}} + C_{\text{surf}})} + \frac{1}{C_{\text{var}}} \right)^{-1} \quad (2)$$

Because there are many capacitances in parallel, changes in

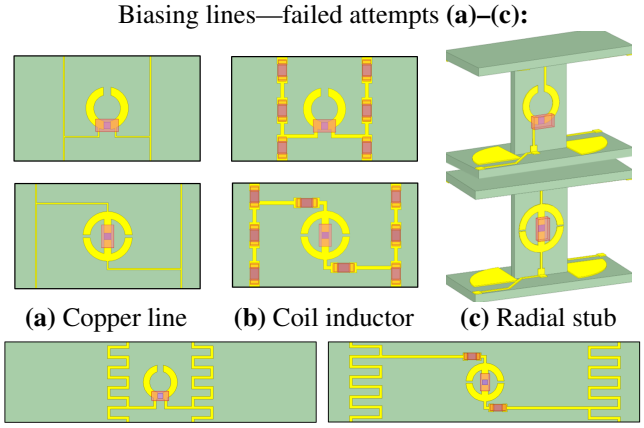


Figure 6: Biasing line designs: notable failed attempts include **(a)** straight microstrip, **(b)** coil inductor, and **(c)** radial stub. **(d)** mmWall uses an inner meander line for magnetic, and an outer meander line for electric meta-atoms.

C_{var} lead to a wider frequency shift than analogous varactor tuning of the magnetic side. Using more precise equation-based analysis (available in Appendix A) that matches our qualitative analysis, we cross-check and finalize design parameters R, g, and w for the magnetic and electric meta-atoms. We refer Appendix A.3 for the values of the design parameters and voltage distributions for different steering angles.

In Fig. 5, we observe that the difference in resonant frequencies between 0 and 10 V for the electric meta-atom are larger than the magnetic meta-atom. Hence, since the effect of C_{var} differs, overlapping of resonance will not always occur when $U_M = U_E$. Rather than of simply finding the area where $U_M = U_E$ as suggested above, we instead need to search for the voltage pair for every desired phase that also maximizes the reflection or transmission magnitude. We do this by running one-time optimization that searches for the voltage pair that maximizes $|T|$ (or $|\Gamma|$) for each phase and generates a static lookup table that will later be used for beam steering.

3.2 Surface Control

To control the meta-atoms, we connect an off-surface control unit via ribbon cables with on-surface *biasing lines*, which altogether comprise the entire *control network* (Fig. 2 on p. 3).

3.2.1 Biasing lines

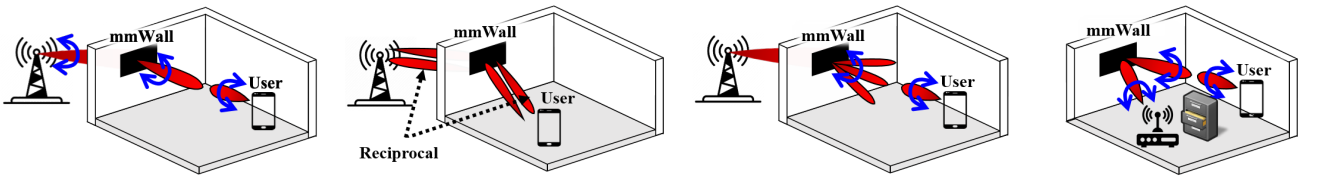
This design process concerns the problem of designing the on-surface control network to interact with mmWave-frequency meta-atoms. Directly connecting a line to the meta-atoms changes the performance of the meta-atom, which causes mmWave signal loss and invalidates the design process described previously (§3.1). To mitigate such adverse effects, we seek to design biasing lines that incorporate radio frequency (RF) chokes, low pass filters that block RF signals within a certain frequency band from propagating on direct current (DC) signal paths. Our primary design goals are to design a biasing network that **1)** minimizes the use of extra components, **2)** avoids a large amount of copper on the panel where the meta-atom is placed, and **3)** is straightforward to fabricate. This is challenging because mmWave meta-atoms are sensitive to the shape and placement of the choke.

Failed attempts. Fig. 6 shows various biasing line structures we have considered. First, we try a straight copper line design **(a)**. We use a narrow width resembling a very high impedance transmission line, to try to attenuate the RF signal while the DC biasing voltage is applied. However, to achieve the desired impedance, a very narrow width transmission line (0.07 mm) is required which is not possible to fabricate by common PCB manufacturing techniques.

Second, we try the use of inductors to create a high-impedance line **(b)**. The impedance of an inductor is determined by the RF frequency and is proportional to its inductance. However, inductance of mmWave inductor components are limited. Hence, we would need to apply at least four inductors in series to achieve the desired isolation, introducing significant surface complexity and also internal resistance that adversely affects unit cell efficiency.

Third, a radial stub which is an open ended transmission line is employed. The length of the stub determines the input impedance of the line, and so thus acts as an RF "choke" that blocks mmWave signals, while a DC biasing voltage is applied to the cell from the control network. The required length of the stub is one-quarter wavelength, which is comparable to the cell size. But if the stub is designed on the same panel, the stub itself would reflect most of the wave, stealing energy to illuminate the cell itself. To avoid this problem, one can put the stubs on a perpendicular panel, as shown in Fig. 6(c). This could potentially solve the wave reflection issue, but would complicate implementation, since there would be one perpendicular panel for each unit cell.

Proposed *meander* structure. To achieve our design goals, we have formulated a meander structure (Fig. 6(d)) that acts as an RF choke, but at the same time connects the vertically adjacent meta-atoms. Longer and thinner traces provide more inductance, so by bending the straight wire vertically and horizontally, we enable the control network itself to be an inductor that outperforms the multiple off-the-shelf inductors. But this increases capacitance between the two meander lines on opposing sides of the unit cell, which also invalidates our meta-atom design process. So mmWall places the meander line of the magnetic meta-atom in a non-overlapping configuration relative to the meander line of the electric meta-atom. To compensate the loss from the microstrip that connects the electric meta-atom and the meander lines, we add two off-the-shelf inductors next to the electric meta-atom.



(a) Refractive establishment **(b)** Angular reciprocity **(c)** Beam tracking **(d)** Reflective establishment

Figure 7: mmWall's refractive link establishment, angular reciprocity property, tracking, and reflective link establishment.

3.2.2 Beam steering and splitting

A conventional phased array transmitter has a net radiation pattern multiplying the radiation pattern of a single element by the *array factor* (AF), the pattern induced by the array. Unlike prior mmWave receive-transmit relay systems which require two phased antenna arrays (one to receive and another to transmit a new phase-shifted signal), mmWall uses only a single array of meta-atoms to directly shift the phase of an existing mmWave signal. For L omni antennas with d separation, each with transmit amplitude A, $\text{AF} = A\sum_{n=0}^{L-1} e^{2\pi jnd(\cos\theta)/\lambda}$ with radio wavelength λ and steering angle θ.

mmWall applies different phase shifts to each meta-atom rib for beam steering. Specifically, by searching over the space of control voltages to maximize reflection or transmission amplitude subject to achieving the desired phase (Fig. 3(c)), we construct a look-up table that maps steering phase φ to the chosen unit cell voltage pair (and without loss of generality) transmission coefficient: $\Phi(\varphi) \rightarrow \langle U_M, U_E, \Gamma \rangle$. The difference with conventional beamforming is that element amplitudes vary, so mmWall's net radiation pattern becomes $\sum_{n=0}^{L-1} \Phi_\Gamma(\phi) e^{j\phi}$ where $\phi = 2\pi nd\cos\theta$.

To transform a single beam into multi-armed beams, we modify the above AF to account for angles θ_1 and θ_2:

$$\sum_{n=0}^{N-1} \left(\alpha\Phi_\Gamma(\phi_1)e^{j\phi_1} + \beta\Phi_\Gamma(\phi_2)e^{j\phi_2}\right) \tag{3}$$

where $\phi_k = 2\pi nd\cos\theta_k$, and α and β are weighting terms that that determine the power of each beam.

3.3 Link Layer Design

Recall that mmWall operates in two different modes, a lens mode and a mirror mode.[2] **1)** In lens mode, a mmWave signal refracts through mmWall allowing, *e.g.*, a user inside the building to communicate with the base station (*ENodeB*) in a cellular network. This requires two beam alignments: one between the ENodeB and mmWall, and another between mmWall and the user. **2)** In mirror mode, mmWall reflects mmWave signals. For example, in wireless LAN settings, it reflects the beam between the AP and user, which requires beam alignment between the AP and mmWall, and again between mmWall and the user.

mmWall electronically switches between the two modes because different users may be located outdoors and indoors. Hence, mmWall sweeps the beam in both lens and mirror mode to align to the user during a beam search.

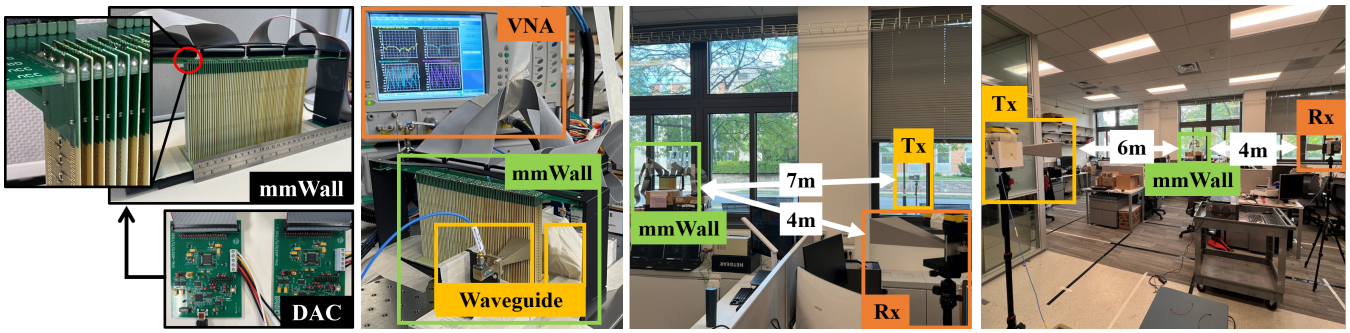
Our development here follows the outline of the existing 5G New Radio (NR) beam management protocol, but adapts it to mmWall's unique capabilities. The current 5G NR beam search proceeds in three steps: **1)** the ENodeB sweeps its beam, the user equipment (*UE*) selects a best direction, and reports it to the ENodeB; **2)** the ENodeB refines its beam (*i.e.*, sweeping a narrower beam over a narrower range), the user detects the best direction and reports it to the ENodeB; **3)** the ENodeB fixes a beam and the UE refines its receiver beam.

To establish a link from a cold start, the ENodeB sweeps different directions such that the user can detect the best beam for an initial link establishment (Fig. 7(a)). If the UE cannot detect the beam or the beam strength is low, it turns mmWall to a lens mode and signal it to simultaneously sweep the beam received from the ENodeB, via sub-6 GHz control. At the same time, the UE scans its receiving beam to various directions. After the search, the UE knows the combination of the ENodeB's transmit beam angle, mmWall's beam refraction angle, and its receive beam angle that maximizes the SNR of downlink signals. Given an initial link, ENodeB and mmWall refine the beam by simultaneously sweeping narrower beams over narrower ranges, and lastly, the user refines its receiving beam.[3] ENodeB-mmWall alignment takes $O(n)$ steps (for n directions), and mmWall-UE alignment takes $O(n^2)$ steps, so cold-start beam alignment as described above takes $O(n^3)$ steps, but only once *ever* when mmWall is installed, because both ENodeB and mmWall are stationary. As long as mmWall remains in the same location, the one-time initial beam alignment is kept constant. Hence, the common case of cold-start beam establishment between mmWall and user in fact requires $O(n^2)$ steps (*cf.* Fig. 7(c)). Also, the above notably does not require modifications to the existing 5G NR protocols.

As illustrated in Fig. 7(b) and demonstrated experimentally in §5.4, mmWall refracts beams in one direction at the same angle as they arrive at the surface from the other side of the

[2] Reflective mode and mirror mode are equivalent.

[3] We note that some full-duplex relays [1] require the relay node's receive direction aligned to the ENodeB, which is not necessary with mmWall.



(a) mmWall hardware **(b)** Near-field testing **(c)** Outdoor-to-indoor scenario **(d)** Indoor reflective scenario

Figure 8: mmWall's hardware implementation, transmissive ('lens') and indoor reflective ('mirror') evaluation scenarios. We placed mmWall at the same location for both scenarios.

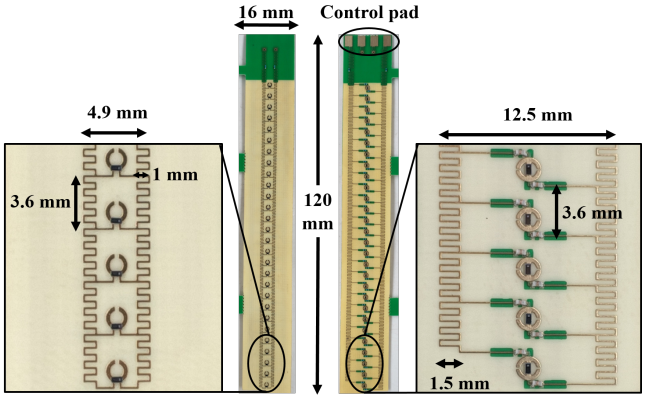


Figure 9: mmWall's *ribs*, comprised of our proposed meta-atom design fabricated on a Rogers printed circuit board.

surface (angular reciprocity), which obviates the need for separate downlink and uplink link establishment. If the downlink has already been established, mmWall does not reconfigure for the uplink. Instead, EnodeB simply switches the direction of its receiving beam to match its transmit beam, and the user transmits in the direction of its receiving beam. This facilitates a quick transition between downlink and uplink.

Since the UE controls mmWall, the user can alternate between the 'lens' mode for outdoor-to-indoor communication and the 'mirror' mode for indoor communication. For example, when the user switches from an outdoor to an indoor ENodeB, it signals mmWall to re-establish the beam estimation process for indoor usage.

Multi-beam search. mmWall can create irregular beam shapes such as multi-arm beams (§5.3), which allows it to leverage state-of-the-art beam searching algorithms that exploit the sparsity of the mmWave channel to accelerate beam search [2, 27] by orders of magnitude improvement (essential for agile and mobile applications such as VR), now for the first time at a surface.

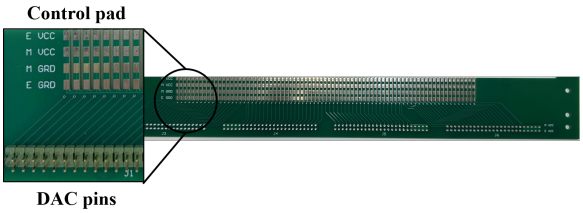


Figure 10: mmWall's FR4 holder/control board.

4 Implementation

We have fabricated and assembled a complete hardware prototype of mmWall, summarized in Fig. 8. mmWall's meta-atoms are fabricated on a 16 by 120 mm *rib* made of Rogers 4003C printed circuit board (PCB) substrate, as shown in Fig. 9. We assemble the PCB and constituent Macom MAVR-000120-1411 varactor diodes[4] and 026011C-1N7 inductors.

In total, we have fabricated 76 ribs, each consisting of 28 vertical meta-atoms. These ribs are mechanically hold together with two perpendicular FR4 panels; one in top and the other in bottom of the structure. The top FR4 also provides control lines as it is shown in Fig. 10. Each rib's control pads are then soldered to the upper holder board, which connects the ribs to a DAC through its microstrip traces and pin headers. The lower holder boards are installed to position and the ribs fixed into these boards. For holding the ribs and FR4 panels steady, a 3D printed enclosure is fabricated that provides a standing support, as shown in Fig. 8(a). The spacing between the adjacent ribs are 2.6 mm, making the dimension of our mmWall prototype 120×197.6 mm. We note that scaling up our prototype with identical ribs and expanded FR4 holder boards is straightforward.

Four 40-channel AD5370 16-bit DACs from Analog Devices allow independent control of both electric and magnetic cells of every mmWall rib. Each DAC supplies a variable

[4]We have modeled this varactor based on its *Simulation Program with Integrated Circuit Emphasis* (SPICE) model (see Appendix, Fig. 17).

0 to 10 V control voltage for each of 40 channels (*i.e.*, one DAC per 20 boards with one channel for U_E and U_M apiece). A laptop is connected to four DACs and listens for control signals from the UE. Once a signal is received, it sends a command to the DACs, which then apply the voltage levels, corresponding to a particular steering angle. Different voltage levels are found from a pre-stored voltage-to-phase look-up table. This control program is written in Microsoft Visual C++, and it can be executed from EnodeB, instead of UE. mmWall hardware, including the DACs, takes 20 μs to reconfigure the beam. The speed of DACs is the key determinant of the total latency, and deploying faster DAC hardware will lower the latency.

5 Evaluation

We begin with field studies that quantify mmWall's SNR gain compared to the best NLoS environment path for both indoor-to-indoor and outdoor-to-indoor links (§5.2). Moreover, we explore the SNR gain and link failure rate under dynamic link conditions. We then evaluate multi-armed beams created by mmWall at various receiver locations (§5.3). We conclude with microbenchmarks to characterize mmWall's steering performance, its support for wide steering angle, angular reciprocity, operation across wide bandwidths, and the impact of the surface size (§5.4).

5.1 Methodology

We conduct evaluations of various indoor and outdoor scenarios. For indoor-to-indoor settings, we place both the receiver (circles in Fig. 11) and transmitter (triangles in Fig. 11) in an office measuring 10×8 m, which includes interior walls, windows, and a server room. Between the three windows, there are two brick walls (black rectangles in Fig. 11). For the outdoor-to-indoor testbed, we locate the transmitter outside the office, approximately $6-7$ m away from the window, while the receiver is inside the office. During the experiments, we place mmWall in front of the window inside the room, and the loss of window is approximately -4 to -5 dB. For each outdoor-to-indoor and indoor-to-indoor experiment, we conduct two sets of experiments, each with a fixed transmitter location and 23 receiver locations. In the first set, the transmitter is perpendicularly facing mmWall and is 6.3 m away (upper subfigures of Fig. 11(a) and left two subfigures of Fig. 11(b)). The second set has the transmitter 6.8 m away from mmWall, and its beam hits the surface at approximately 30° to 40° angle (lower subfigures of Fig. 11(a) and right subfigures of Fig. 11(b)). During the beam search, mmWall steers the angle by the step of 0.5°. For end-to-end performance, we report SNR with a noise floor of 80 dBm.

Near-field experiments. Given that the measured Huygens pattern is likely to deviate from simulated results due to manufacturing tolerances, it is crucial to conduct accurate measurement through near-field experiments and compile a voltage-to-phase look-up table. Specifically, we collect near-field reflection and transmission coefficients of mmWall using two-port Anritsu MS4647B VNA, operating from 70 kHz to 70 GHz, as shown in Fig. 8(b). The Huygens pattern measured from the VNA is shown in Appendix C. To minimize measurement error, we perform a two port calibration before acquiring the data. For data collection, we program the VNA using LabVIEW, which communicates with four DACs through the socket. During the measurement, mmWall is placed in between two waveguide horn antennas that are connected to the VNA. We place two horn antennas closely to mmWall to resemble the near-field simulation. Since the area of mmWall is larger than the aperture of waveguide horns, we collect the pattern on multiple locations of mmWall. In Appendix C, we present a measured Huygens pattern at different locations of mmWall and demonstrate the robustness of mmWall against fabrication variations.

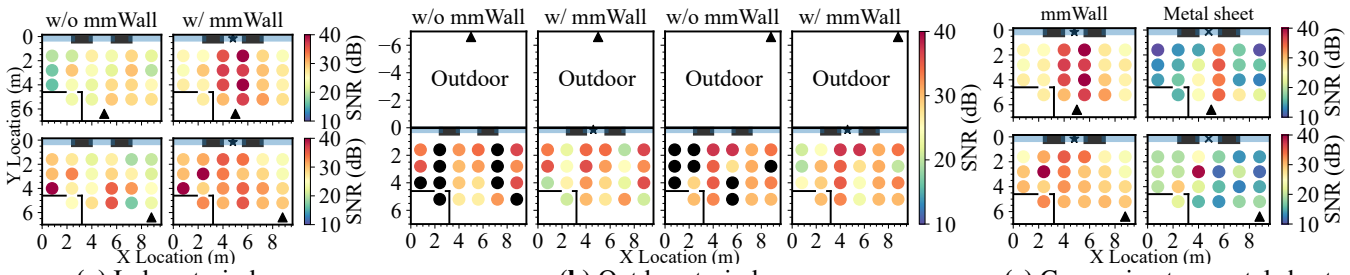
Far-field experiments. A standard mmWave base station is equipped with highly directional phased array antennas and supports an average EIRP range of 55-60 dBm [12, 13] or more. With a 25 dBi transmit horn antenna, the maximum EIRP we achieved is 31 dBm, which is in accordance with FCC rules [14]. We use the same antenna at the receiver but apply a -10 dB correction to reflect typical UE antenna gain. Specifically, to generate mmWave signals, we use off-the-shelf phase-locked loop (PLL) frequency synthesizers ADF4371 with integrated VCO and frequency quadrupler, which quadruples 6.125 GHz VCO signals to 24.5 GHz. At the transmitter, the PLL output power is < -13 dBm, and we use the PLL in conjunction with a variable gain amplifier (VGA) HMC997LC4, which amplifies signals by 18 dBm.

5.2 In-situ Performance

In this section, we evaluate the end-to-end performance of mmWall for indoor and outdoor scenarios.

SNR improvement over the best environment path. To evaluate the effectiveness of mmWall in improving SNR in scenarios with blocked LoS paths, we conduct SNR measurements at multiple transmitter and receiver locations (two locations for the transmitter and 23 locations for the receiver). For each link, the transmitter and receiver (and mmWall if deployed) search for an NLoS path that maximizes the SNR.

Fig. 11 illustrates the measurements taken prior to and following the deployment of mmWall. Specifically, Fig. 11(a) presents the SNRs obtained when the transmitter was positioned towards the window at 0° (upper subfigure) and 30° (lower subfigure) in an indoor testbed. The results of both subfigures show that our indoor testbed has a rich scattering environment, with some receiver locations achieving SNR levels exceeding 25 dB in the absence of mmWall. However,



(a) Indoor-to-indoor. **(b)** Outdoor-to-indoor. **(c)** Comparing to a metal sheet

Figure 11: mmWall's SNR improvement over the best NLoS environment path in two scenarios: (a) when both transmitter and receiver are located indoors (*upper*: transmitter facing mmWall perpendicularly; *lower*: transmitter facing 30° away from mmWall) and (b) when the transmitter is located outdoors (*left*: transmitting perpendicularly; *right*: transmitting at a 30° off-angle). We use the following notations: mmWall ★, transmitter ▲, receiver ◯. ● indicates no signal.

receivers located at either end of the room experience SNR levels below 20 dB. With mmWall, all receivers, including the ones in the corner, achieve SNRs of at least 24 dB. Also, the nodes located within mmWall's steering angle of −45° to 45° has SNRs greater than 30 dB. This improvement in SNR is particularly evident in Fig. 12. In Fig. 12 (*left*), we plot a CDF of the best environment SNRs (black curves) alongside the SNRs of mmWall links at the corresponding receiver location (rectangles). Fig. 12 (*center*) shows the CDFs of maximum SNRs between the environment and mmWall links, while Fig. 12 (*right*) shows the CDFs of the SNR gains over the environment path per receiver location. As shown in Fig. 12 (*upper*), mmWall ensures outage-free communication for 91% of receiver locations at 128 QAM [24] mmWave data rates, while only 40-50% of receivers achieve the same rate in the absence of mmWall. Moreover, among 80% of receivers that experience the gain from mmWall, some receive more than a 15 dB SNR boost.

In Fig. 11(b), we present the SNR improvement in outdoor-to-indoor scenarios. Without mmWall, receivers unable to establish an NLoS link through the window experience complete link failure. With mmWall, on the other hand, all receivers achieve SNRs of at least 19 – 20 dB. Fig. 12 (*lower*) shows the CDFs of outdoor-to-indoor SNR improvement. A single mmWall guarantees 64-QAM for almost all receiver locations and a 30 dB SNR boost for 40% of the links. Our results demonstrate that mmWall is highly beneficial for improving mmWave signals quality in the cases of wall blockage.

Deploying multiple mmWalls. To evaluate more than one mmWall, we place another mmWall (downward triangles in Fig. 12) in front of the window on the right side of the room. Fig. 12 (*upper*) demonstrates the SNR gain from deploying two mmWalls for indoor-to-indoor links. Compared to the gain from a single mmWall, an additional mmWall provides ≤ 5 dB SNR gain for some links. As shown in Fig. 12 (*lower*), there is almost no gain from adding an extra mmWall for outdoor-to-indoor links. The results indicate that a single mmWall is sufficient to provide good coverage (at least 128-

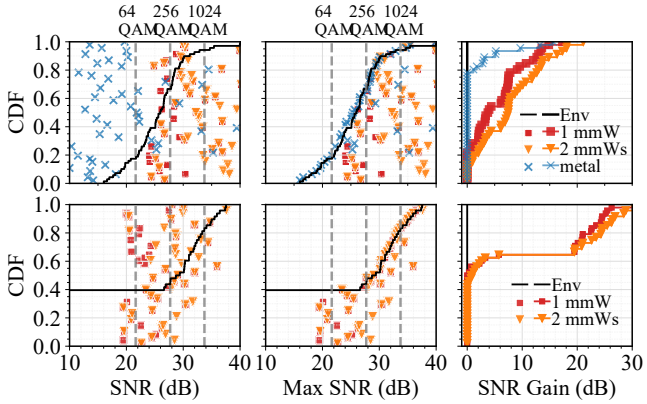


Figure 12: The SNR improvement from the use of one or more mmWalls at various receiver locations in indoor-to-indoor (*upper*) and outdoor-to-indoor (*lower*) scenarios. SNRs collected from a given receiver location are plotted on the same y-axis value (*left*: CDFs of the best environment SNRs in black curves alongside the SNRs of mmWall links at the corresponding receiver location in rectangles. The maximum SNRs between two mmWalls placed in different locations are denoted with downward triangles; *center*: the best available SNRs with or without one or more mmWalls; *right*: the SNR gains attained with one or more mmWalls compared to the best environment path in various Rx locations).

QAM for reflective and 64-QAM for transmissive links) in a 10 × 8 m office room. In a static environment another mmWall will not help *if* a mmWall path is already available.

Improving reliability for dynamic links. While a single mmWall delivers good SNRs throughout all receiver locations, it is still possible for blockages to occur on mmWall links. Likewise, even if there is a robust NLoS path present, it can still be blocked. At mmWave frequencies, the indoor environment typically provides three to four strong paths, including the LoS path [23]. Due to the limited number of available paths, an increase in blockages can easily result

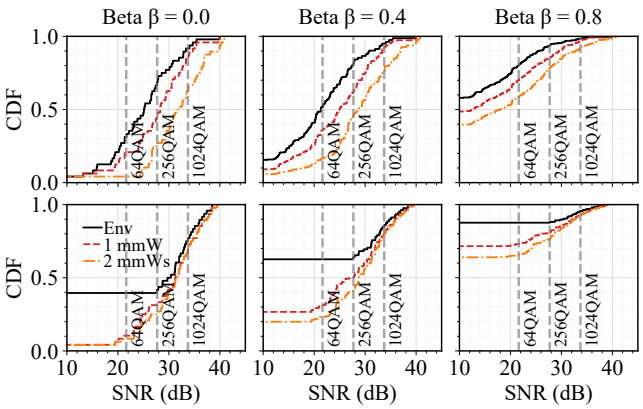
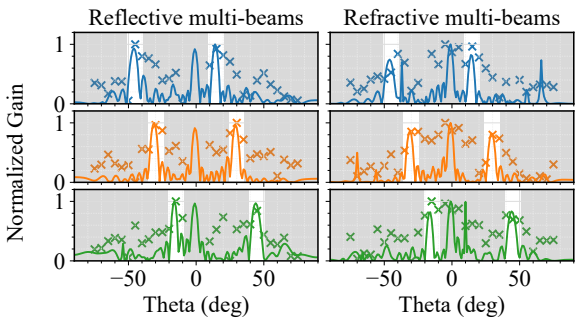


Figure 13: The SNR improvement (multiple mmWalls) for dynamic links (*upper*: indoor-to-indoor; *lower*: outdoor-to-indoor scenarios). β is a blockage probability.

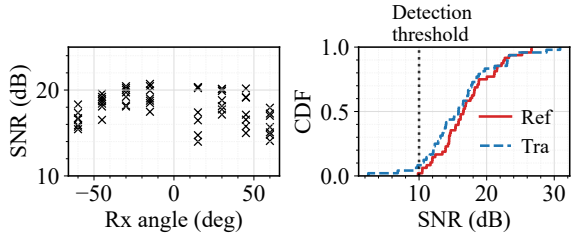
in link failure, which exacerbates when these obstructions are in motion. One of the primary benefits of using one or multiple mmWalls is the enhancement of link reliability. By providing a diverse, strong alternative path, mmWall reduces the probability of link scarcity. In Fig. 13, we demonstrate the SNR gain across various Rx locations as a function of the *blockage probability* β[5] for both environment and mmWall links. In indoor-to-indoor scenarios, a single mmWall and two mmWalls reduce the probability of link failure by a ratio of up to 10% and 20% under 80% path blockage, respectively. For the outdoor testbed, the probability of link failure decreases by 40% for a single mmWall and 45% for two mmWalls under 40% blockage probability. Hence, we conclude that multiple mmWalls are beneficial when channel environments are highly dynamic.

One may argue that deploying a simple reflective metal sheet could help, but mmWall's ability to steer the beam has a significant impact on the extent of coverage. We evaluate the SNRs of links reflected by a 60×60 cm metal sheet, along with the SNRs of links steered by a 10×20 cm mmWall. As shown in Fig. 11(c) (*right*), only 10% of the receivers achieves SNRs above 30 dB, and the remaining 90% have SNRs below 15 dB. It is also worth noting that for a metal sheet, the SNRs depend largely on the location of the receiver and transmitter. In Fig. 11(c) (*right*), only the receivers that are placed and perfectly aligned with the angle of specular reflection achieve a high SNR. In Fig. 12, only 8% of the receivers achieve more than 5 dB SNR gain from the metal sheet. On the other hand, mmWall guarantees at least 25 dB SNRs across all areas. We conclude that, compared to fixed-angle reflection, mmWall links are less sensitive to the location of the transmitter, receiver, and surface, making them much more robust.

[5]A blockage probability is equivalent to a probability of complete link failure for each path. Under various available paths, the blockage probability of one path is independent from the other.



(a) mmWall's multi-armed beam pattern (*upper:* −45/15° degree beam split; *middle:* −30/30° split; *lower:* −15/45° split). Empirical points are denoted ×, with simulation curves.



(b) The SNRs of aligned multi-beams (*left:* a fixed distance between the transmitter and mmWall and between the receiver and mmWall; *right:* various Tx and Rx locations in the office setting.)

Figure 14: Evaluation of mmWall's multi-armed beams.

5.3 Multi-armed Beams

We next evaluate mmWall's capability to generate multi-armed beams. Fig. 14(a) presents our measurements on the multi-armed beams, along with simulation results from HFSS. Here, mmWall splits an incident beam into two beams at −45°/15° and steers these multi-beams to −30°/30° and −15°/45°. To measure the beam pattern, we position the transmitter and receiver three meters away from mmWall and record the gain of mmWall as we move the receiver from a −90° to 90° angle with respect to mmWall. Since we did not measure the beam pattern in an anechoic chamber, the received beam interfered with signals reflected off the indoor environment. Despite the interference, we observe that the gain peaks at the angles where mmWall splits the beam. Furthermore, as mmWall steers its multi-armed beams, the measured peaks change accordingly. Our results show a peak at 0° due to leakage that was directly fed from the transmitter to the receiver. Reducing the distance between the transmitter and receiver and/or increasing the size of mmWall will reduce the peak at 0°.

We then measure SNRs as mmWall generates and steers various multi-beams, the beams that are 15° to 120° apart from each other. The distance between the transmitter and mmWall and between the receiver and mmWall are fixed to 2 m. Fig. 14(b) (*left*) reveals that as the beam is split into a wider angle, SNR drops.

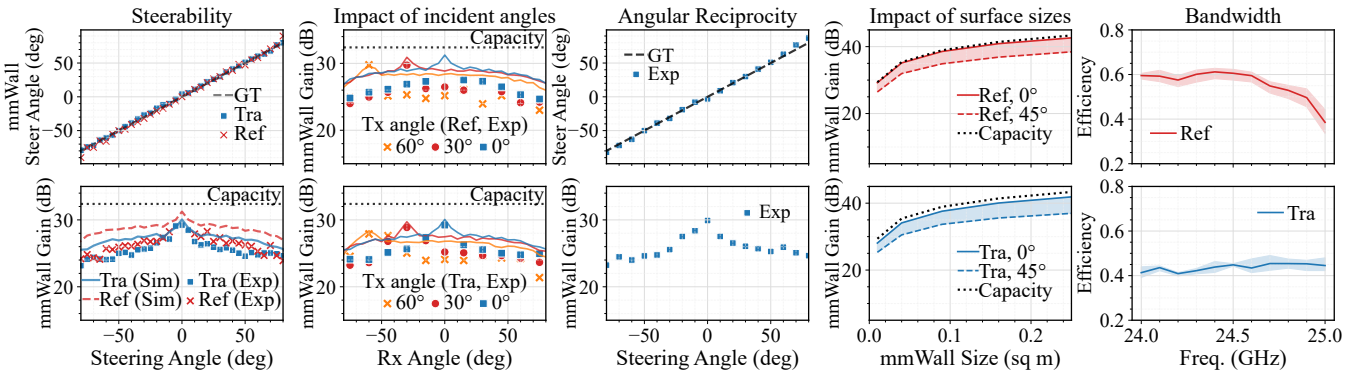


Figure 15: Microbenchmarks evaluating (*left* to *right*:) surface steerability, performance sensitivity of the incident wave angle, angular reciprocity, surface size, and frequency bandwidth. Empirical points are denoted with markers, with simulation curves.

To demonstrate the feasibility of a beam search using multi-armed beams, mmWall again splits the beam into two beams that are 15° to 120° apart from each other. Then it aligns the beam with the receivers at 23 different locations in the room. Fig. 14(b) (*right*) reports the SNRs of mmWall's multi-beam links aligned with various receivers. The results show that more than 90% of multi-beam links achieve SNRs above 10 dB. Considering that no signal is detected in many locations under outdoor-to-indoor settings, 10 dB SNR is enough for the receiver to detect the beam and start the alignment. We conclude that mmWall can generate multi-armed beams that are sufficiently strong to accelerate beam search.

5.4 Microbenchmarks

We now evaluate mmWall's steering performance, its support for wide steering angle, angular reciprocity, operation across wide bandwidths, and the impact of the surface size. The microbenchmark testbed consists of the receiver and transmitter modules that are three meters away from mmWall. Fig. 15 presents both the actual experimental measurements (markers) and simulated results (curves) acquired from HFSS.

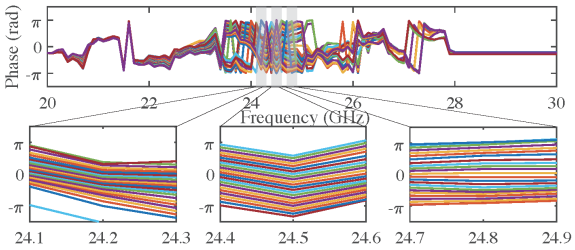
Since mmWall does not have an amplifier, the effective aperture A_e is the primary factor that determines its gain. A well-defined relation for the effective aperture in terms of the aperture gain G is $A_e 4\pi/\lambda^2$. We define the aperture gain as our capacity and compare it against our measured mmWall gains in our microbenchmarks. A rigorous analysis on mmWall gain is available in Appendix B.

mmWall controllability. Fig. 15 (*upper first*) presents mmWall's beam alignment accuracy. We place the receiver at 37 locations in our testbed and find the angle that provides the maximum SNR as mmWall sweeps the beam from −80° to 80° angle. During the experiment, the transmitter is facing mmWall at 0° angle. For both reflection and transmission, mmWall accurately steers the beam with at most 3° difference from the groundtruth (GT). Second, we evaluate the effect of a steering angle on the mmWall gain in Fig. 15 (*lower first*). As mmWall increases the steering angle, the gain slowly decreases. Furthermore, reflection provides a slightly higher gain than transmission.

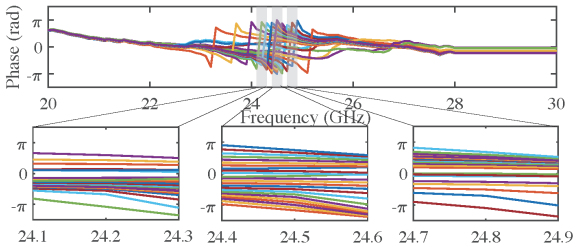
Support for wide steering angle. In this microbenchmark, we evaluate the effect of incident beam angles on the mmWall gain jointly with the steering angle. Here, we move the transmitter to three different locations and the receiver to 37 locations. Fig. 15 (*upper second*) and Fig. 15 (*lower second*) show the impact of incident angles for reflection and transmission, respectively. For both scenarios, increasing the incident beam angle does not greatly reduce the mmWall gain. An important observation is that even with 135° steering angle (*e.g.,* Tx angle at 60° and Rx angle at −75°), mmWall achieves more than 22 dB gain, indicating that mmWall is capable of refracting the beam in a very wide angle.

Angular reciprocity. Once mmWall achieves alignment for the downlink channel, the uplink channel also becomes aligned due to its angular reciprocity. To demonstrate this property, we evaluate the accuracy of uplink beam alignment and the corresponding mmWall gain when downlink alignment is already established. In Fig. 15 (*upper third*), uplink alignment using reciprocity is very accurate and is within an error of 3°. Also, Fig. 15 (*lower third*) shows that all corresponding mmWall gains are above 23 dB using reciprocity.

Operation across wide bandwidths. To demonstrate mmWall's phase coverage across a wide bandwidth, we present our VNA measurements from 20 to 30 GHz. In Fig. 16, each curve indicates the phase response of voltage levels in our lookup table that we compiled at our center frequency, 24.5 GHz. Here, we emphasize three points. First, mmWall provides a full phase coverage from $-\pi$ to π over the 200 MHz 5G mmWave link bandwidth. Second, within 200 MHz bands (highlighted in gray), the phase distributions are mostly constant, allowing improvements over the entirety of these bandwidths. Third, mmWall can operate in the entire 23.5 to 25.5 GHz band, as it provides a wide range of phases there. Hence, mmWall operates over the mmWave 5G band-



(a) Reflection Γ



(b) Transmission T

Figure 16: mmWall's phase coverage and consistency (VNA measurement) across different frequencies. The curves indicate the voltage pairs (U_M, U_E) that provide $-180°$ to $180°$ phase shift with the step of $15°$ at 24.5 GHz. The phases are unwrapped across the mmWall's operating bandwidth.

width. More importantly, our meta-atom design goal is to reduce transmission or reflection *loss level* with a full phase coverage. To quantify both magnitude and phase coverage at the same time, we define *efficiency* as $\sum_{\phi=-180}^{180}(Te^{-1j\phi})/360$ where T is a set of points obtained from the near-field transmissive (reflective) Huygens pattern that provides the maximum magnitude for $-180°$ to $180°$ phases. Fig. 15 (*rightmost*) demonstrates that for both reflection and transmission, the efficiency is consistent and declines after 24.9 GHz. Since targeted operational bandwidth for 5G mmWave is 200 MHz, we conclude that mmWall operates within the 5G bandwidth.

Increasing mmWall size. Fig. 15 (*fourth*) shows an increase of simulated gains as mmWall size increases from 10×10 cm to 50×50 cm with $0°$ (for reflection, it is a specular reflection) and $45°$ steering angle. Also, we compare our simulated results against the effective aperture-based capacity. mmWall gains at both $0°$ and $45°$ steering angle increase with increasing surface size, following the capacity trend.

6 Discussion

In this section, we discuss several limitations of mmWall and our potential solutions.

3D beamforming. 3D beamforming technique is beneficial in massive MIMO communications as it sophisticatedly controls the beam in different directions in spatial domain. With mmWall, meta-atoms on the same rib share the voltage level, and therefore it is structured as a 2D linear array. To achieve 3D beamforming, we can simply separate mmWall control lines in the vertical direction. In the future, we will vertically partition mmWall and separately control them.

Tinted window. While mmWave waves propagate through a glass wall with virtually no loss, the penetration loss increases when the glass is metal-coated. Our window has approximately -4 to -5 dB loss with a light tint. If our window is tinted more, SNRs will drop, and this decrease will be equivalent to the increase of penetration loss from a different level of a tint. There is an existing work [45] that measures reflection coefficients and penetration loss for common building materials at mmWave. According to the paper, the penetration loss may increase by more than 20 dB when the window is heavily tinted. With such windows, we can remove the tint of the small area of the window (approx. 0.02 sq m) for mmWall.

Indoor AP as a 5G mmWave relay. An indoor mmWave AP can serve as a relay when the outdoor cell coverage fails to reach indoors. To accomplish this, cellular operators require indoor infrastructure to install an AP capable of receiving 5G signals. This AP then communicates with an internal modem through an Ethernet cable, and the modem wirelessly transmits the signal to the user through Wi-Fi. This deployment is not only costly and time-consuming but also hard to implement. On the other hand, a single mmWall at a fixed location can achieve all three use cases, including 5G outdoor-to-indoor and outdoor-to-outdoor coverage, and indoor WiFi.

7 Conclusion

This paper presents mmWall, the first Huygens metasurface that can reconfigures itself to relay an incoming mmWave beam as either a non-specular "lens" or "mirror." Our prototype steers single- or multi-armed beams at non-specular directions, arbitrarily in real-time. We conduct an extensive evaluation in various indoor and outdoor settings, demonstrating significant SNR improvement, and describe how scaling to even larger sizes is eminently possible.

8 Acknowledgements

This work is supported by the National Science Foundation under grant CNS-1617161, Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation (CFI) and Ontario Research Fund (ORF).

References

[1] O. Abari, D. Bharadia, A. Duffield, D. Katabi. Enabling high-quality untethered virtual reality. *USENIX NSDI Symp.*, 531–544, 2017.

[2] O. Abari, H. Hassanieh, M. Rodriguez, D. Katabi. Millimeter wave communications: From point-to-point links to agile network connections. *ACM HotNets Workshop*, 169–175, 2016.

[3] Y. Azar, G. N. Wong, K. Wang, R. Mayzus, J. K. Schulz, H. Zhao, F. Gutierrez, D. Hwang, T. S. Rappaport. 28 GHz propagation measurements for outdoor cellular communications using steerable beam antennas in New York City. *IEEE Intl. Conf. on Comms.*, 5143–5147, 2013.

[4] K. Chen, Y. Feng, F. Monticone, J. Zhao, B. Zhu, T. Jiang, L. Zhang, Y. Kim, X. Ding, S. Zhang, *et al.* A reconfigurable active Huygens metalens. *Advanced materials*, **29**(17), 1606,422, 2017.

[5] L. Chen, W. Hu, K. Jamieson, X. Chen, D. Fang, J. Gummeson. Pushing the physical limits of IoT devices with programmable metasurfaces. *USENIX NSDI Symp.*, 2021.

[6] K. W. Cho, M. H. Mazaheri, J. Gummeson, O. Abari, K. Jamieson. mmwall: A reconfigurable metamaterial surface for mmwave networks. *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, 119–125, 2021.

[7] L. Dai, B. Wang, M. Wang, X. Yang, J. Tan, S. Bi, S. Xu, F. Yang, Z. Chen, M. D. Renzo, C.-B. Chae, L. Hanzo. Reconfigurable intelligent surface-based wireless communications: Antenna design, prototyping, and experimental results. *IEEE Access*, **8**, 45,913–45,923, 2020.

[8] X. Ding, Z. Wang, G. Hu, J. Liu, K. Zhang, H. Li, B. Ratni, S. N. Burokur, Q. Wu, J. Tan, *et al.* Metasurface holographic image projection based on mathematical properties of Fourier transform. *PhotoniX*, **1**(1), 1–12, 2020.

[9] A. Epstein, G. V. Eleftheriades. Passive lossless Huygens metasurfaces for conversion of arbitrary source field to directive radiation. *IEEE Transactions on Antennas and Propagation*, **62**(11), 5680–5695, 2014.

[10] A. Epstein, G. V. Eleftheriades. Huygens' metasurfaces via the equivalence principle: design and applications. *JOSA B*, **33**(2), A31–A50, 2016.

[11] B. Esmail, H. Majid, Z. Abidin, S. Dahlan, M. Rahim, O. Ayop, *et al.* New metamaterial structure with reconfigurable refractive index at 5G candidate band. *J. Optoelectron Adv M*, **21**(1-2), 101–107, 2019.

[12] FEDERAL COMMUNICATIONS COMMISSION. Ericsson ab pre-nr and nr base radio akrd901059-1.

[13] FEDERAL COMMUNICATIONS COMMISSION. Sfg-aa100dc 5g access unit test report 1 samsung electronics.

[14] FEDERAL COMMUNICATIONS COMMISSION. Title 47, code for federal regulations.

[15] H. Friis. A note on a simple transmission formula. *Proc. of the IRE*, **34**(5), 254–256, 1946.

[16] H. Hassanieh, O. Abari, M. Rodriguez, M. Abdelghany, D. Katabi, P. Indyk. Fast millimeter wave beam alignment. *ACM SIGCOMM Conf.*, 432–445, 2018.

[17] C. L. Holloway, E. F. Kuester, J. A. Gordon, J. O'Hara, J. Booth, D. R. Smith. An overview of the theory and applications of metasurfaces: The two-dimensional equivalents of metamaterials. *IEEE Antennas and Propagation Magazine*, **54**(2), 10–35, 2012.

[18] I. K. Jain, R. Subbaraman, D. Bharadia. Two beams are better than one: Towards reliable and high throughput mmwave links. *ACM SIGCOMM Conf.*, 488–502. New York, NY, USA, 2021.

[19] S. Jog, J. Wang, J. Guan, T. Moon, H. Hassanieh, R. R. Choudhury. Many-to-Many beam alignment in millimeter wave networks. *USENIX NSDI Symp.*, 783–800. Boston, MA, 2019.

[20] Z. Li, Y. Shu, G. Ananthanarayanan, L. Shangguan, K. Jamieson, P. Bahl. Spider: A multi-hop millimeter-wave network for live video analytics. *2021 IEEE/ACM Symp. on Edge Computing (SEC)*, 178–191, 2021.

[21] M. Liu, D. A. Powell, Y. Zarate, I. V. Shadrivov. Huygens' metadevices for parametric waves. *Phys. Rev. X*, **8**(3), 031,077, 2018.

[22] G. R. MacCartney, T. S. Rappaport, S. Sun, S. Deng. Indoor office wideband millimeter-wave propagation measurements and channel models at 28 and 73 GHz for ultra-dense 5G wireless networks. *IEEE Access*, **3**, 2388–2424, 2015.

[23] M. Nemati, B. Maham, S. R. Pokhrel, J. Choi. Modeling RIS empowered outdoor-to-indoor communication in mmWave cellular networks. *IEEE Transactions on Communications*, **69**(11), 7837–7850, 2021.

[24] K. Nishimori, K. Kitao, T. Imai. Interference-based decode and forward scheme using relay nodes in heterogeneous networks. *International Journal of Antennas and Propagation*, **2012**, 2012.

[25] C. Pfeiffer, A. Grbic. Metamaterial Huygens' surfaces: Tailoring wave fronts with reflectionless sheets. *Phys. Rev. Lett.*, **110**, 197,401, 2013.

[26] K. Qian, L. Yao, X. Zhang, T. N. Ng. MilliMirror: 3D printed reflecting surface for millimeter-wave coverage expansion. *ACM MobiCom Conf.*, 2022.

[27] M. E. Rasekh, Z. Marzi, Y. Zhu, U. Madhow, H. Zheng. Noncoherent mmWave path tracking. *ACM HotMobile Workshop*, 13–18, 2017.

[28] N. Shlezinger, G. C. Alexandropoulos, M. F. Imani, Y. C. Eldar, D. R. Smith. Dynamic metasurface antennas for 6G extreme massive MIMO communications. *IEEE Wireless Comms.*, **28**(2), 106–113, 2021.

[29] D. R. Smith, J. B. Pendry, M. C. Wiltshire. Metamaterials and negative refractive index. *Science*, **305**(5685), 788–792, 2004.

[30] S. Sur, I. Pefkianakis, X. Zhang, K.-H. Kim. WiFi-assisted 60 GHz wireless networks. *ACM MobiCom Conf.*, 28–41. New York, NY, USA, 2017.

[31] X. Tan, Z. Sun, D. Koutsonikolas, J. M. Jornet. Enabling indoor mobile millimeter-wave networks based on smart reflect-arrays. *IEEE INFOCOM Conf.*, 270–278, 2018.

[32] W. Tang, M. Z. Chen, X. Chen, J. Y. Dai, Y. Han, M. Di Renzo, Y. Zeng, S. Jin, Q. Cheng, T. J. Cui. Wireless communications with reconfigurable intelligent surface: Path loss modeling and experimental measurement. *IEEE Trans. on Wireless Comms.*, **20**(1), 421–439, 2021.

[33] W. Tang, X. Chen, M. Z. Chen, J. Y. Dai, Y. Han, M. Di Renzo, S. Jin, Q. Cheng, T. J. Cui. Path loss modeling and measurements for reconfigurable intelligent surfaces in the millimeter-wave frequency band. *IEEE Trans. on Comms. (Early Access)*, 2022.

[34] A. Vallecchi, E. Shamonina, C. J. Stevens. Analytical model of the fundamental mode of 3D square split ring resonators. *J. of Applied Physics*, **125**(1), 014,901, 2019.

[35] E. Violette, R. Espeland, R. DeBolt, F. Schwering. Millimeter-wave propagation at street level in an urban environment. *IEEE Trans. on Geoscience and Remote Sensing*, **26**(3), 368–380, 1988.

[36] X. Wang, P.-Y. Qin, A. T. Le, H. Zhang, R. Jin, Y. J. Guo. Beam scanning transmitarray employing reconfigurable dual-layer huygens element. *IEEE Transactions on Antennas and Propagation*, **70**(9), 7491–7500, 2022.

[37] T. Wei, A. Zhou, X. Zhang. Facilitating robust 60 GHz network deployment by sensing ambient reflectors. *USENIX NSDI Symp.*, 213–226, 2017.

[38] Z. Wu, Y. Ra'di, A. Grbic. Tunable metasurfaces: A polarization rotator design. *Physical Review X*, **9**(1), 011,036, 2019.

[39] Y. Xing, F. Vook, E. Visotsky, M. Cudak, A. Ghosh. Raytracing-based system performance of intelligent reflecting surfaces at 28 GHz. *IEEE Intl. Conf. on Comms.*, 498–503, 2022.

[40] C. Xue, Q. Lou, Z. N. Chen. Broadband double-layered huygens' metasurface lens antenna for 5g millimeter-wave systems. *IEEE Transactions on Antennas and Propagation*, **68**(3), 1468–1476, 2019.

[41] K. Ying, Z. Gao, S. Lyu, Y. Wu, H. Wang, M.-S. Alouini. GMD-based hybrid beamforming for large reconfigurable intelligent surface assisted millimeter-wave massive MIMO. *IEEE Access*, **8**, 19,530–19,539, 2020.

[42] R. I. Zelaya, W. Sussman, J. Gummeson, K. Jamieson, W. Hu. LAVA: fine-grained 3D indoor wireless coverage for small IoT devices. *ACM SIGCOMM Conf.*, 123–136, 2021.

[43] L. Zhang, X. Q. Chen, S. Liu, Q. Zhang, J. Zhao, J. Y. Dai, G. D. Bai, X. Wan, Q. Cheng, G. Castaldi, *et al.* Space-time-coding digital metasurfaces. *Nature Communications*, **9**(1), 1–11, 2018.

[44] H. Zhao, R. Mayzus, S. Sun, M. Samimi, J. K. Schulz, Y. Azar, K. Wang, G. N. Wong, F. Gutierrez, T. S. Rappaport. 28 GHz millimeter wave cellular communication measurements for reflection and penetration loss in and around buildings in New York City. *IEEE Intl. Conf. on Comms.*, 5163–5167, 2013.

[45] H. Zhao, R. Mayzus, S. Sun, M. Samimi, J. K. Schulz, Y. Azar, K. Wang, G. N. Wong, F. Gutierrez, T. S. Rappaport. 28 ghz millimeter wave cellular communication measurements for reflection and penetration loss in and around buildings in new york city. *2013 IEEE International Conference on Communications (ICC)*, 5163–5167. IEEE, 2013.

A Unit Cell Electromagnetic Analysis

We now present a full mathematical analysis of mmWall's unit cells. Since electromagnetic fields are naturally continuous and will not change the propagation characteristics by itself, we artificially introduce electric and magnetic surface currents $(\vec{J}_s, \vec{M}_s)$ from the electric and magnetic meta-atoms, enforcing a field discontinuity:

$$\vec{J}_s = \hat{n} \times [H_t - H_i],\ \vec{M}_s = -\hat{n} \times [E_t - E_i] \tag{4}$$

where $\hat{n}$ is a unit normal. The average tangential field applied on the meta-atom pair induces $(\vec{J}_s, \vec{M}_s)$. To induce suitable surface currents, we need a proper surface impedance for each meta-atom:

$$\begin{aligned} \hat{n} \times [E_{avg}] &= Z_e \vec{J}_s = Z_e \hat{n} \times [H_2 - H_1] \\ \hat{n} \times [H_{avg}] &= Y_m \vec{M}_s = -Y_m \hat{n} \times [E_2 - E_1] \end{aligned} \tag{5}$$

where Z_e is the electric surface impedance and Y_m is the magnetic surface admittance equivalent to $1/Z_m$. In fact, the electric and magnetic meta-atoms are each described by a surface impedance of LC oscillating circuit containing inductance L and capacitance C. Mathematically, we can formulate the surface impedance of the electric and magnetic meta-atom as

$$Z_e = \left(\frac{2\pi f C_e - 1}{(2\pi f)^2 L_e C_e} \right) j,\ Y_m = \left(\frac{1 - (2\pi f)^2 L_m C_m}{2\pi f C_m} \right) j \tag{6}$$

where f indicates the resonant frequency. Each meta-atom behaves as an LC circuit when its resonant frequency f matches the frequency of the incident wave. Mathematically, the resonant frequency is equivalent to $f = (2\pi\sqrt{LC})^{-1}$.

Given Z_e and Y_m, we can formulate the transmission coefficient T and reflection coefficient Γ of a meta-atom pair:

$$T = \frac{4 - Y_m \cdot Z_e}{(2 + Y_m \cdot \eta)(2 + Z_e/\eta))},\ \Gamma = \frac{2(Z_e/\eta - Y_m \cdot \eta)}{(2 + Y_m \cdot \eta)(2 + Z_e/\eta)} \tag{7}$$

where η is the wave impedance in free space. Hence, by changing the surface impedance (Z_e, Y_m), we precisely control the phase of the coefficients, creating an arbitrary phase shift on the incident wave [10].

The excitation of the electric and magnetic surface currents, or, equivalently, the values of Z_e and Y_m is tuned by changing the capacitive or inductive loading of the meta-atoms as shown in Eq. (6). Hence, to make HMS reconfigurable, we load a voltage-controlled capacitor, varactor diode, on each meta-atom. By applying voltage across each varactor, we can arbitrary change the surface impedance, or equivalently, the phase of the transmission or reflective coefficient.

Since the electric and magnetic meta-atoms are superimposed on the surface, we dissect the equivalent circuit model for the electric and magnetic meta-atom individually.

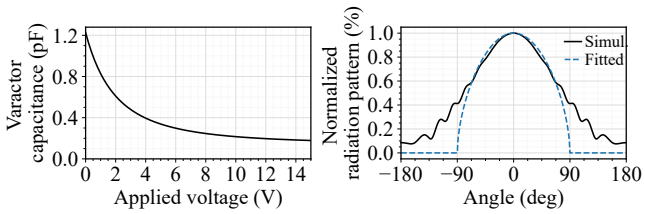


Figure 17: *Left:* C_{var} as the voltage applied to varactor changes, modeled with SPICE simulation; *Right:* mmWall element normalized beam pattern $F(\theta)$ simulated with HFSS and fitted function.

A.1 Magnetic Meta-atom

In this section, we provide the formulas for the magnetic meta-atom's capacitance and inductance discussed in §3.1.2. First, we define the inductance of a circular metallic loop L_{loop} as

$$L_{loop} = \mu_0 R \left(log\left(\frac{8R_m}{t+w} - \frac{1}{2} \right) \right), \tag{8}$$

where R is a mean radius, and μ_0 is free-space permeability. Since there is a gap on the top of a metallic loop, the inductance of our magnetic meta-atom can be calculated as

$$L_m = p_m L_{loop} = \left(1 - \frac{g}{2\pi R} \right) L_{loop}, \tag{9}$$

where g is a length of the gap. Now, we present the calculation of C_m. First, the gap in the metallic loop creates a parallel-plate capacitance as follow:

$$C_{gap} = \varepsilon \frac{wt}{g} + \varepsilon(t + w + g), \tag{10}$$

where w is the width of the loop, and t is the thickness of the copper. Here, $\varepsilon = \varepsilon_0 \varepsilon_{eff}$ where ε_0 is free-space permittivity, and ε_{eff} is effective permittivity, which can be calculated as

$$\varepsilon_{eff} = \frac{\varepsilon_r + 1}{2} + \left(\frac{\varepsilon_r - 1}{2} \right) \left(\frac{1}{\sqrt{(1 + 12t/e)}} \right) \tag{11}$$

where ε_r is the permittivity of the substrate. Second, there is a capacitance induced by the metallic ring itself:

$$C_{surf} = \frac{2\varepsilon(t + w)}{\pi} ln\left(\frac{4R}{g} \right) \tag{12}$$

Lastly, the varactor diode adds the capacitance as discussed in §3.1.2. We have modeled our varactor, of Macom MAVR-000120-1411, based on its *Simulation Program with Integrated Circuit Emphasis* (SPICE) model and demonstrate our simulated C_{var} values in the left subfigure of Fig. 17. Then, we formulate C_m according to Eq. (1). Finally, we model the circuit diagram as a series impedance where the series impedance itself corresponds to the surface impedance $Z_m = 1/Y_m$.

	Radius R (mm)	Gap g (mm)	Width w (mm)
Ele.	0.8831	0.1016	0.3048
Mag.	0.7907	0.2794	0.3048

Table 1: mmWall design parameters.

A.2 Electric Meta-atom

Now, we provide the capacitance and inductance calculation for the electric meta-atom. First, we formulate the inductance of a half-circle ring L_{curve} as follow:

$$L_{curve} = (p_e L_{circle})/2 = \frac{1}{2}\left(\left(1 - \frac{g}{2\pi R_m}\right) L_{circle}\right). \quad (13)$$

Based on [11], we compute the the inductance of the strip as

$$L_{strip} = \mu_0 l/4\pi\left[2\sinh^{-1}\left(\frac{l}{w}\right) + 2\left(\frac{1}{w}\right)\sinh^{-1}\left(\frac{w}{l}\right) - \frac{2(w^2+l^2)^{1.5}}{3lw^2} + \frac{2}{3}\left(\frac{l}{w}\right)^2 + \frac{2}{3}\left(\frac{w}{l}\right)\right] \quad (14)$$

where l is the length of strip, which is equivalent to $2R_m$, and w is the width of the trace. We then combine all inductance values into L_e as

$$L_e = (L_{curve}/2) + L_{strip} \quad (15)$$

The formulas for the gap capacitance and surface capacitance for the electric meta-atom are the same as the magnetic meta-atom, and we define C_e according to Eq. (2). Finally, the surface impedance of the electric meta-atom corresponds to a shunt impedance.

A.3 Design Parameters

We present the exact values for our design parameters, including radius R, gap g, and width w of the magnetic and electric meta-atom, in Table. 1. Also, the voltage levels applied to the magnetic and electric meta-atoms for different phase shifts are shown in Fig. 18. The y-axis indicates the voltage level, and the x-axis is different ribs. Specifically, Fig. 18(a) demonstrates a set of U_M and U_E required for $-30°$, $-15°$, $0°$, $15°$, and $30°$ transmissive steering. Similarly, Fig. 18(b) shows the voltages values required for reflective steering.

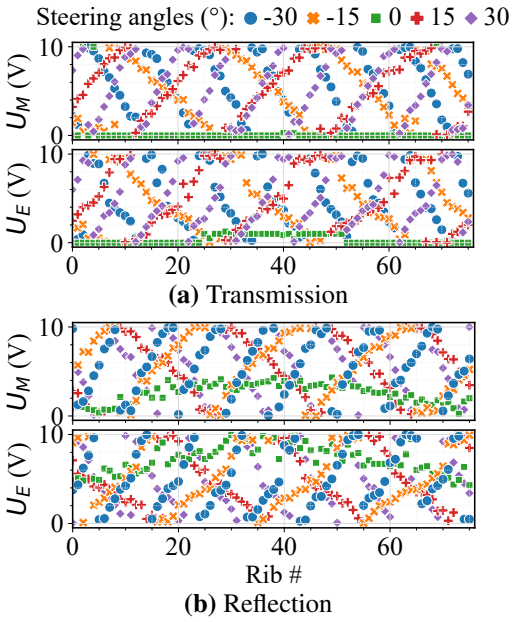


Figure 18: *Upper:* a set of voltage levels applied to the magnetic and electric meta-atoms U_M and U_E for transmissive steering; *Lower:* voltage levels applied for reflective steering.

B Path Loss Model

This section presents a standard path loss model calculation largely following the development in prior similar efforts targeting lower frequencies [32], useful for our purposes to establish the basic feasibility of our design prior to hardware fabrication and full-scale evaluation.

First let us assume that a transmitter directly communicates with a receiver. According to the Friis formula [15], the power intercepted by the receiving antenna with effective aperture Ae_R and distance between transmitter and receiver d is:

$$P_i = S_R Ae_R = \left(\frac{P_T}{4\pi d^2} G_T\right) Ae_R \quad (16)$$

where S_R is the received power density, and G_T is the peak gain of the transmitting antenna. Since the effective aperture $Ae_R = \frac{\lambda^2}{4\pi} G_R$ where G_R denotes the gain of the receiving antenna, we rewrite Eq. (16) as

$$P_i = \left(\frac{P_T}{4\pi d^2} G_T\right)\left(\frac{\lambda^2}{4\pi} G_R\right) = P_T G_T G_R \left(\frac{\lambda}{4\pi d}\right)^2. \quad (17)$$

Now we consider a transmitter communicating with the receiver via mmWall. Given Eq. (17), we formulate the power the nm^{th} meta-atom captures from the transmitter as

$$P^i_{nm} = P_T G_T G_w \left(\frac{\lambda}{4\pi d_{i,nm}}\right)^2, \quad (18)$$

where G_w denotes the gain of the meta-atom in the direction of the transmitter, and $d_{i,nm}$ is the distance between the transmitter and nm^{th} meta-atom. Similarly, we can calculate the power received by the receiving antenna from the nm^{th} meta-atom as:

$$P_{R,nm} = P^s_{nm} G_R G_w \left(\frac{\lambda}{4\pi d_{s,nm}}\right)^2, \quad (19)$$

where G_w is the meta-atom gain scattered in the direction of the receiver, $d_{s,nm}$ is the distance between nm^{th} meta-atom

to the receiver, P^s_{nm} is the power applied by each meta-atom, and $P^s_{nm} = P^i_{nm}\varepsilon$. Here, ε accounts for the limited efficiency of meta-atom and insertion losses associated with components. To simplify the formula, we assume $\varepsilon = 1$. To calculate the power from the transmitter to the receiver, we then combine Eqs. (18) and (19):

$$P_{R,nm} = P_T G_T G_R \frac{G_w G_w}{d^2_{i,nm} d^2_{s,nm}} \left(\frac{\lambda}{4\pi}\right)^4 \tag{20}$$

Here, we emphasize that in the link budget, we must calculate the gain of mmWall twice, one for receiving and another for transmitting. Hence, Eq. (20) has two G_w. Since mmWall consists of a large array of meta-atoms, we can formulate the total received power as a sum of the received powers from all meta-atoms as

$$P_R = \left| \sum_{n=1}^{N} \sum_{m=1}^{M} C_{nm} \sqrt{P_{R,nm}} e^{j\phi_{nm}} \right|^2, \tag{21}$$

where $C_{n,m}$ denotes the transmission or reflection coefficient of the nm^{th} meta-atom, and the phase $\phi_{nm} = 2\pi(d_{i,nm} + d_{s,nm})/\lambda$. In a lens mode $C_{n,m} = T_{n,m}$, and in a mirror mode $C_{n,m} = \Gamma_{n,m}$. We already defined $T_{n,m}$ and $\Gamma_{n,m}$ in eq. Eq. (7). Finally, we write the total received power as:

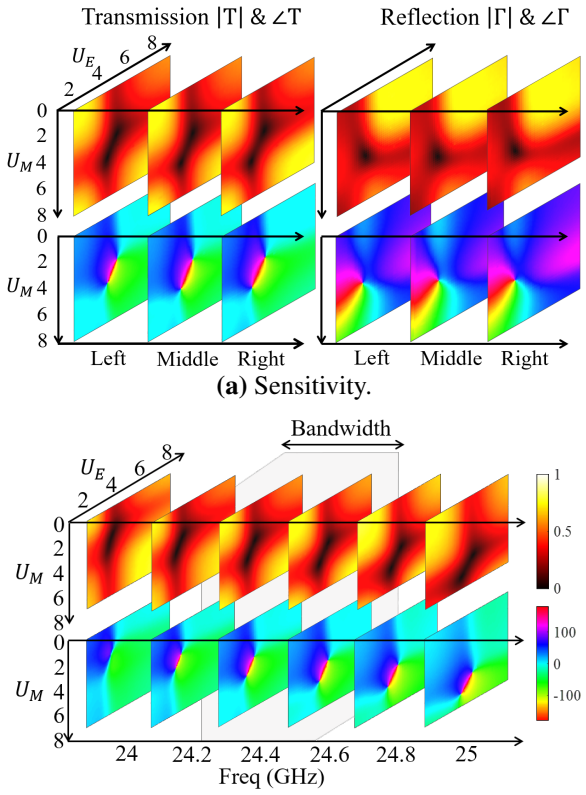
$$P_R = P_T G_T G_R \left(\frac{\lambda}{4\pi}\right)^4 \left| \sum_{n=1}^{N} \sum_{m=1}^{M} C_{nm} \frac{\sqrt{G_w G_w}}{d_{i,nm} d_{s,nm}} e^{j\phi_{nm}} \right|^2. \tag{22}$$

However, the meta-atom gain G_w is unknown. Thus, we re-define G_w as a power radiation pattern from each meta-atom, which is equivalent to $GF(\theta_{nm})$. G is a gain that depends on the physical area (*i.e.* the effective aperture) of the meta-atom, and $F(\theta_{nm})$ is the normalized power radiation pattern. Based on the effective aperture formula, $G = (4\pi/\lambda^2)Ae_{nm} = (4\pi/\lambda^2)(xy)$ where x and y are vertical and horizontal meta-atom spacing, respectively. Unlike traditional antennas with $x = y = \lambda/2$, our meta-atom has $x = \lambda/4.8$ and $y = \lambda/3.4$. Moreover, $F(\theta_{nm})$ defines the variation of the power radiated or received by a meta-atom:

$$F(\theta) = \begin{cases} cos^q(\theta) & \theta \in [0, \pi/2] \\ 0 & \theta \in [\pi/2, \pi] \end{cases} \tag{23}$$

where θ are the angle from the meta-atom to a certain transmitting or receiving direction. In the right subfigure of Fig. 17, we present a simulated mmWall element beam pattern $F(\theta_{nm})$ as well as the curve fitted with Eq. (23). Based on our curve fit, $q = 0.5611$.

Far-field beamforming. In the far-field, we can approximate $d_{s,nm} = d_s$ and $d_{i,nm} = d_i$ since d_i and d_s are much greater than the distance between different meta-atoms. However, we do not approximate $d_{s,nm} = d_s$ and $d_{i,nm} = d_i$ for the phase ϕ_{nm}.



(a) Sensitivity.

(b) Bandwidth.

Figure 19: Meta-atom microbenchmark

Then, we can simplify Eq. (22) as:

$$P_R = P_T G_T G_R \left(\frac{Ae_{nm}}{4\pi d_i d_s}\right)^2 F(\theta_i) F(\theta_s) \left| \sum_{n=1}^{N} \sum_{m=1}^{M} C_{nm} e^{j\phi_{nm}} \right|^2 \tag{24}$$

This indicates that we can maximize the received power by configuring each meta-atom's $\angle C_{nm}$ to $-\phi_{nm}$. Finally, the path loss of a correctly reconfigured mmWall as:

$$L^{-1}_{mmWall} = \left(\frac{xy}{4\pi d_i d_s}\right)^2 F(\theta_i) F(\theta_s) \left| \sum_{n=1}^{N} \sum_{m=1}^{M} |C_{nm}| \right|^2 \tag{25}$$

Since $0 < |C_{nm}| < 1$ for both transmissive and reflective mode, increasing the number of meta-atoms N and/or M reduces the path loss. Assuming $|C_{nm}|$ is close to 1, the path loss of mmWall is proportional to $1/(NM)^2$. While increasing the element spacing x and y seems to reduce the loss, it is not always true because $|C_{nm}|$ decreases when x and y increase due to increasing coupling between adjacent meta-atoms.

C Meta-atom controllability and sensitivity

We present the Huygens pattern measured from the VNA in Fig. 19(a). We measure the near-field Huygens pattern

in three different areas of mmWall to evaluate its sensitivity against fabrication variation. For all three areas, we observe a 360-degree phase variation with high magnitude for both transmission and reflection. Moreover, the patterns do not vary across the different areas of the surface, signifying that manufacturing tolerance do not greatly affect mmWall's near-field performance. We also demonstrate the Huygens pattern across mmWall's operating bandwidth in Fig. 19(b). Within the 200 MHz bandwidth, the pattern is consistent.

Building Flexible, Low-Cost Wireless Access Networks With Magma

Shaddi Hasan[1], Amar Padmanabhan[2], Bruce Davie[3], Jennifer Rexford[4], Ulas Kozat[5], Hunter Gatewood[5], Shruti Sanadhya[5], Nick Yurchenko[5], Tariq Al-Khasib[5], Oriol Batalla[5], Marie Bremner[5], Andrei Lee[5], Evgeniy Makeev[5], Scott Moeller[5], Alex Rodriguez[5], Pravin Shelar[5], Karthik Subraveti[5], Sudarshan Kandi[5], Alejandro Xoconostle[5], Praveen Kumar Ramakrishnan[5], Xiaochen Tian[6], and Anoop Tomar[5]

[1]Virginia Tech [2]Databricks [3]Systems Approach [4]Princeton [5]Meta [6]Independent

Abstract

Billions of people remain without Internet access due to availability or affordability of service. In this paper, we present Magma, an open and flexible system for building low-cost wireless access networks. Magma aims to connect users where operator economics are difficult due to issues such as low population density or income levels, while preserving features expected in cellular networks such as authentication and billing policies. To achieve this, and in contrast to traditional cellular networks, Magma adopts an approach that extensively leverages Internet design patterns, terminating access network-specific protocols at the edge and abstracting the access network from the core architecture. This decision allows Magma to refactor the wireless core using SDN (software-defined networking) principles and leverage other techniques from modern distributed systems. In doing so, Magma lowers cost and operational complexity for network operators while achieving resilience, scalability, and rich policy support.

1 Introduction

Good Internet connectivity has become a basic necessity for people and enterprises all over the world. Yet, more than one-third of the global population does not have access to the Internet [54], and many other users do not have the high-speed connectivity needed for many important applications. The problem is primarily a matter of economics: commercial network operators claim that today's Internet has reached the user footprint that seems commercially viable to serve [29]. To reach the next billion users, we must reduce the *cost* of providing Internet access or enable actors beyond traditional, large-scale commercial operators to build sustainable, scalable network infrastructure. We need effective ways to reduce both capital and operational costs, through less expensive equipment and software, less reliance on highly skilled network administrators, and increased utilization of existing local capabilities. At the same time, providers need ways to manage their limited network resources effectively to enable sustainable network operation. Cellular networks typically achieve these goals with per-user policies, which may include per-user data caps, rate limits, or usage-based charging.

Unfortunately, conventional wireless solutions are not well suited to many scenarios affecting under-served users. WiFi access points operating on unlicensed spectrum cannot generally provide efficient coverage to large geographic regions (e.g., sparsely populated rural areas) due to the propagation characteristics of the radios. Plus, WiFi networks typically do not offer fine-grained policies to manage resources. In contrast, cellular base stations offer wider coverage, support more users, and connect to core networks that support more flexible policies. However, today's cellular access networks rely on expensive equipment, complex protocols, and a highly skilled workforce, limiting their ability to cost-effectively connect the next billion. While cellular networks scale *up* to large user populations, they do not scale *down* well. That is, a small cellular deployment is typically quite expensive. Magma aims to bridge the gap between these two classes of solution: cellular networks with rich policies, large user populations, and long distances, and the simpler but less scalable WiFi networks.

More fundamentally, we observe that choosing to use a cellular radio access network (RAN) today *forces* a network operator to make a series of decisions that deeply impact their network operations that are not inherently related to their choice of access network technology. This choice binds a network operator to: (i) a specific network architecture—namely the 3GPP-defined arrangement of interfaces for network management and on-path devices for policy enforcement, (ii) an ecosystem of vendors that has largely evolved to meet the needs of massive-scale telecom operators, and (iii) a particular set of radio frequencies and associated regulatory requirements. The Magma project aims to change all this, by creating an open-source, carrier-grade wireless networking platform that supports a wide range of deployment scenarios. Magma deployments can leverage whatever radio access technology is readily available and most appropriate for their density of subscribers or deployment scenario. Magma achieves this goal through *access gateways* that terminate the radio-specific protocols as close to the radios as possible. As a result, Magma

allows carriers to augment an existing cellular deployment with WiFi hotspots in popular locations (e.g., athletic venues), or use LTE base stations to serve homes in rural areas, using a single core network and management platform.

Ideally, new deployments could start small and grow over time. Magma achieves a "scale as you go" design through horizontal scaling of software components that run on commodity hardware, as is common in cloud-computing environments. Magma also leverages open-source software components (e.g., Open vSwitch, gRPC, Kubernetes, Prometheus) commonly used in cloud settings. Magma simplifies network management by adopting software-defined networking concepts, so that a central point of control can be used to set network policies, manage subscribers, etc. Magma adopts a hierarchical control plane to improve scalability. Magma supports only the essential features for efficient Internet access (e.g., authentication, accounting, and per-user policies), and forgoes some complex features. For example, while Magma supports both mobility (within the area served by an access gateway) and roaming, it does not yet support seamless user mobility *between* access gateways; this is because mobility has not been a requirement for the use cases that commercial deployments of Magma support (e.g., home broadband or backhaul to WiFi hotspots). As other work has observed [38], modern end-host protocols and applications can perform well without in-network mobility support.

In this experiences paper, we present the lessons learned in designing and deploying Magma. We discuss how the goals of supporting heterogeneous radio and backhaul technologies and flexible policies, all at low cost, lead to a novel software architecture. Magma is used in real-world deployments that vary significantly in geographic scope, number of users, technology choices, and the business models that make them financially sustainable. In Section 2 we motivate Magma's central tenet that the radio access technology should not dictate the network architecture. Then, Section 3 discusses how the design of the access gateways enables Magma to support diverse technologies, a scalable control plane, fault tolerance, and more. Next, Section 4 presents an experimental evaluation that demonstrates that Magma design and implementation achieves good performance and scalability along with a discussion of two production access networks. We have seen cost savings in one deployment of 43% compared to traditional approaches due to lower operational, hardware, and software costs. Our deployment experience also illustrates how Magma scales both up and down, with one deployment supporting more than 800 eNodeBs (base stations) in 45 US states at the time of writing. Section 5 presents related work. The paper concludes in Section 6 with a discussion of ongoing work on Magma and future challenges.

Ethics. This paper raises no ethical concerns. For the deployments discussed in Section 4, we only consider operational data and did not have access to any user data or traffic.

2 The Radio Access Technology Should Not Drive the Network Architecture

Traditionally, the choice of radio access technology dictates a raft of other decisions about the network architecture. In contrast, Magma starts with the premise that each radio access technology has a role to play in reaching diverse user communities and that network operators should be able to use the radio and backhaul technologies most suited for a deployment scenario. In short, wireless network architectures should, like the Internet itself, abstract away the link layer.

2.1 WiFi vs. Cellular Access Networks

The two main classes of radio access technologies emerged as extensions to existing wireline networks with different design philosophies. WiFi extended IP networks, whereas modern cellular data networks began as extensions to voice telephony networks. Many of the differences between these two classes of access networks follow directly from this early distinction.

WiFi: WiFi allows inexperienced users to run simple low-cost local-area networks on their own. These networks use unlicensed radio spectrum (typically at 2.4 GHz and 5 GHz) that do not require WiFi network operators to get advance regulatory approval. At the same time, anyone can access the same spectrum, subject to limits on transmission power. As a result, WiFi networks share their bands with devices including baby monitors, cordless phones, and smart power meters, so the WiFi MAC layer must assume that a WiFi access point (AP) operates in the presence of physical-layer interference. Combined with power restrictions that limit transmit distance, WiFi is most suitable for dense coverage in small areas. WiFi service is best-effort, consistent with the Internet design philosophy—and realistic given the likelihood of interference. Enterprise WiFi deployments, such as those on college campuses and in corporate office buildings, perform more centralized management of interference across multiple overlapping access points. Still, the risk of interference means that the service remains best-effort.

Cellular: Cellular access networks allow telecommunication providers to offer wireless service to their subscribers, typically using licensed spectrum that is owned or leased by the carrier for long periods of time at high cost. Since the radio has exclusive access to spectrum over a geographic region, cellular waveforms are designed for wide-area coverage and high spectral efficiency, with deployments by well-resourced actors that can acquire land, build and connect towers, and hire skilled staff.

Regardless of access technology, any network of significant scale requires substantial investment in equipment, staffing, and, in the case of cellular networks, regulatory licenses. Thus, beyond very small networks, operators implement policies to manage limited spectrum, ranging from access control; charging for service based on time, usage, or more sophisticated

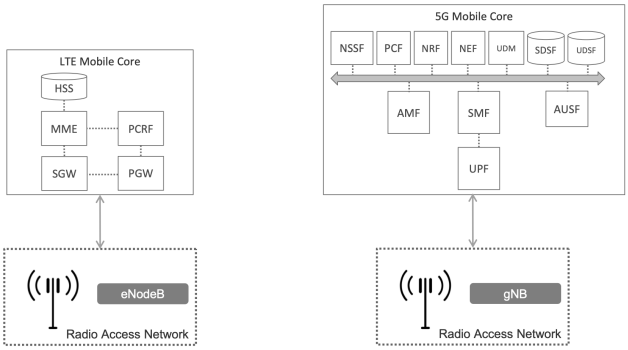


Figure 1: Differences between the LTE and 5G architectures. See Appendix for explanation of acronyms.

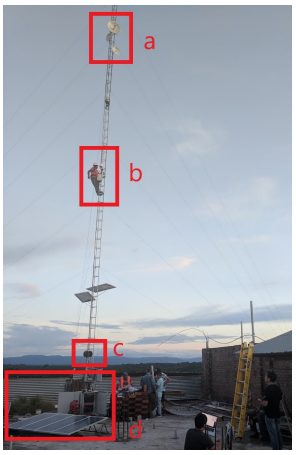


Figure 2: An early Magma deployment with a small rural ISP in Peru (their first cellular site). Components (top to bottom) include (a) point-to-point wireless backhaul, (b) LTE radio and antenna, (c) ruggedized embedded PC serving as Magma AGW, and (d) solar power and battery backup for site.

techniques that incorporate community values [36]; usage caps; and throttling. The policy specification for LTE, for example, runs to almost 300 pages [14]. A simple example policy would be: "rate limit customer C to X Mbps until they have sent Y GB in interval t_1, then limit to Z Mbps for interval t_2." Supporting flexible policies can help carriers reach under-served users in a financially sustainable manner: even networks operating for social reasons still incur costs and must efficiently manage limited resources.

These capabilities are implemented by a sophisticated packet core network that connects multiple base stations to the Internet. In contrast to how the Internet architecture changes incrementally, each generation of cellular network has been an opportunity to rethink everything from authentication to the modularity of the control and data planes.

As such, different generations of the 3GPP standards [15] have different packet core architectures. UTMS ("3G") differs from LTE ("4G"), which differs from 5G, and all of the generations differ from enterprise WiFi. The differences between LTE and 5G are illustrated in Figure 1, adapted from [46]. The different radio technologies require differences in the base stations (eNodeB versus gNB) but note also the change in modularity of the mobile core. WiFi would be different again, and less standardized, with functions such as Authorization, Authentication, and Accounting (AAA) corresponding roughly to Mobility Management Entity (MME) and Home Subscriber Server (HSS) components in LTE.

Today, the boundaries between cellular and WiFi are increasingly blurry, with operators deploying each technology in scenarios more classically served by the other. In recent years, large WiFi deployments have adopted more sophisticated methods for user authentication, power control, seamless mobility, and more [23,24,60], with efforts like Eduroam [58] and OpenRoaming [57] bringing cellular-like wide area roaming to users of WiFi access networks. Similarly, some cellular access networks now use "lightly licensed" spectrum, such as Citizen's Band Radio Service (CBRS) [17] that supports dynamic allocation of radio spectrum to give radios exclusive access to some portion of the spectrum (on the timescales of tens of minutes). Enterprises are deploying *private* cellular access networks for a range of use cases—such as industrial automation, medical applications, and Internet access at hotels and sporting events—that need better radio efficiency, authentication, and performance than WiFi traditionally offers.

2.2 Lowering the Barriers

Magma aims to lower the barrier to connecting under-served populations via wireless networks. We argue that operators should be able to choose the appropriate access technology for any deployment without then being locked into a core architecture that is compatible only with that access type. A single design that supports heterogeneous technologies amortizes the substantial engineering effort for creating software and the costs of training and supporting those who operate the networks. Plus, the design enables a single carrier to use multiple radio technologies (e.g., WiFi in shopping malls and cellular elsewhere) on a single core.

Cellular access has high barriers to *entry*. Network operators deploying cellular access technologies must make large capital investments (CapEx) in infrastructure: whereas a WiFi access point can cost under US$100, even a low-cost cellular deployment would cost at least 1-2 orders of magnitude more. Traditionally, cellular core network equipment is designed for large deployments with hundreds of base stations and does not "scale down" to small initial deployments at reasonable cost; these networks also have high operational costs (OpEx), relying on highly skilled staff to manage the equipment. In addition, remote communities may not have affordable access to the high-quality, low-latency backhaul (e.g., fiber) links cellular networks typically rely upon. Instead, these networks may use satellite or wireless backhaul links with lower reliability and performance.

In contrast, WiFi deployments can start small, but present high barriers to *scale*. WiFi networks do not typically require skilled staff to deploy. Because WiFi is an inherently best-

effort access technology, these networks can leverage any available backhaul, even ad-hoc mesh backhaul using the same physical WiFi radios. Yet WiFi networks do not typically offer scalable ways to implement network policy or (beyond proprietary and vendor-specific solutions) to manage large networks. Thus, it is difficult for a WiFi-oriented operator to offer financially sustainable service over a wide area along the lines of large (usually cellular) operators.

Despite the differences between WiFi and cellular, these barriers are *not* fundamental. The building blocks of network policies are common in each; what is missing is architectural support. Software-defined networking can help address these gaps by enabling network-wide control over a distributed infrastructure, and adopting "scale out" techniques based on commodity components can reduce cost. In short, adopting and extending successful Internet and cloud approaches to scalability and management can make it possible to create a wireless access network that is both flexible and affordable.

3 Magma Architecture

Magma cannot overcome the shortcomings of existing solutions simply by reimplementing a standard, 3GPP-compliant mobile core. Instead, Magma terminates the radio-specific protocols as early as possible, in *access gateways* (AGWs) connected directly to the radio access network, as shown in Figure 3. These access gateways are instrumental in handling a variety of radio technologies in a single design. The Magma architecture goes beyond the traditional RAN/core split of 3GPP to place additional functionality in the access gateway, with a goal of making the packet core more scalable, including scaling down. Notably, Magma adopts the architecture of software-defined networking (SDN) systems, using a hierarchical control-plane design where a local controller in each access gateway interacts with a centralized *orchestrator*. The orchestrator is the central point of control for the system and maintains authoritative state related to system-wide configuration (*config* state). Runtime state, which relates to the activity of user equipment (UEs), is localized to the AGW that serves the appropriate base station for a given UE.

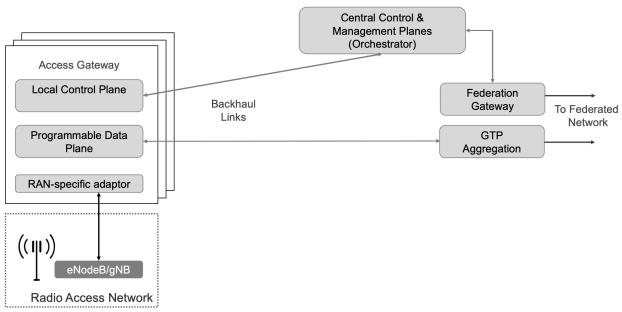


Figure 3: Simplified Magma architecture

Each AGW is a small fault domain, ensuring that the failure or upgrade of any one component affects relatively few users. In this way, Magma's architecture is similar to modern cloud systems designed to run on low-cost hardware that is prone to failure [26]. Magma adopts other ideas from cloud architectures, including the use of gRPC for communication among components, a "desired state" model for state synchronization, and software-based, programmable data plane. While common in cloud computing deployments, these decisions deviate significantly from the way typical 3GPP networks are designed and managed.

3.1 Abstracting the Radio Access Technology

As Figure 1 illustrates, the details of the radio access technology traditionally "leak" into the core network. To counter this, Magma identifies a core set of functions that the AGW must implement for any radio technology (e.g., finding the appropriate policy for a given subscriber) and provides them in an access-technology-independent way. These functions form the heart of an AGW, as illustrated on the right side of Figure 4. Control protocols, which are specific to a given radio technology, are terminated *early* in technology-specific modules close to the radio. These modules, on the left of the figure, communicate with the generic functions (e.g., subscriber management, access control and management) on the right using messages that are RAN-agnostic.

Consider the example of "attaching" a newly active UE. The UE communicates with a nearby base station over a temporary (unauthenticated) radio link. In traditional 4G implementations, the base station forwards the request to the Mobility Management Entity (MME), which initiates an authentication protocol with the UE. The MME consults a subscriber database, authenticates the UE, creates an entry in a session table, and informs the other components of the parameters needed to serve the UE including: (a) assigning an IP address to the UE and setting the appropriate QoS parameters in the data plane; (b) instructing the base station to establish an encrypted channel to the UE; and (c) giving the UE the symmetric key for the encrypted channel. At the end of this sequence of events, the UE has an active *session* established with the mobile core and is able to send and receive data.

These functions are performed in the Magma AGW in a way that abstracts the details of the radio technology, as illustrated on the right-hand side of Figure 4. For example, Magma's subscriber database has the union of all capabilities across the radio access types, even if some fields in a given database row are valid only for some technologies. QoS policies, for example, are less rich in WiFi than in 4G networks, while 4G policies are in turn less rich than those of 5G. Similarly, UE authentication and session establishment are done in a common way by generic functions that cover 4G, 5G, and WiFi procedures. The data plane, which is implemented in different devices across 4G, 5G, and WiFi, is implemented in

Magma	LTE	5G	WiFi
Access Control/Management	MME	AMF	RADIUS AAA
Subscriber Management	HSS	UDM/AUSF	RADIUS AAA
Session/Policy Management	MME/PCRF	SMF/PCF	RADIUS AAA
Data Plane Configuration	SGW/PGW	SMF	WiFi data plane
Data Plane	SGW/PGW	UPF	WiFi data plane
Device Management	per-box configuration		
Telemetry and logging	no equivalent defined		

Table 1: Magma abstractions vs. RAN-specific versions

a common, programmable data plane for Magma.

Table 1 shows how the various components of 4G, 5G, and WiFi are all mapped onto a common set of Magma abstractions. The key observation here is that there are a certain set of functions that need to be performed to authenticate users, establish session state, control the data plane, and so on. Magma does all of these in a generic way that is agnostic to the radio technology in use, thus providing an implementation in which the radio-specific details are abstracted from the core and limited to protocol termination close to the radio itself.

Additionally, Magma adds some generic functions that are not part of the 3GPP standards: device management and telemetry. Coupled with the SDN architecture, this simplifies the management of a large number of devices spread over a wide geographical area. Rather than logging into a specific device to configure it or check its statistics, Magma provides central management and monitoring from the orchestrator, where it can be leveraged by other systems that consume the northbound API. We have found that considering device management and telemetry as first-class responsibilities of Magma significantly reduces the operational complexity and cost of operating access networks (Section 4.3.1).

We do not claim that Magma's decomposition of functionality (Figure 4) is fundamental, but our operational experience shows that it is useful both from an engineering perspective and for a wide range of use cases (as discussed further in Section 4). The modularity between components allows Magma's internal interfaces to evolve independently of the RAN, aligning with an iterative development approach and in stark contrast to rigid 3GPP interface definitions. This has enabled the team to perform major changes to AGW functionality, such as adding new features (e.g., 5G support) or refactoring internal services without exposing these changes southbound toward the RAN or northbound toward the orchestrator API.

All communication between the RAN-specific modules on the left of Figure 4 and the generic functions on the right use gRPC [5], an open-source Remote Procedure Call (RPC) framework, as does all long-distance communication (e.g., from the AGW to the orchestrator). Although this is a typical approach for building modern distributed systems, it differs substantially from the protocols defined for communication among 3GPP components, which leak endpoint (e.g., UE and MME) consistency requirements into a network-level protocol. By running over HTTP, gRPC inherits the resilience to loss and delay of TCP/IP, which is absent from some 3GPP

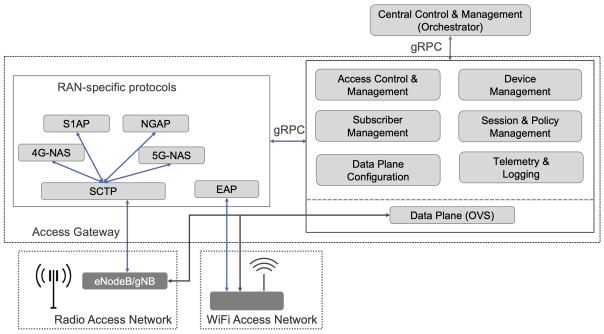


Figure 4: Common functions and RAN-specific protocols in the Magma architecture.

protocols designed for more benign, controlled environments (e.g., leased lines). A concrete example is GTP (GPRS Tunneling Protocol), which is sensitive to loss and latency to the point that it struggles to operate over lower quality or congested backhaul links, such as satellite or shared microwave links. Thus, adopting gRPC allows Magma more latitude to implement alternative consistency models without breaking UE state machines in a wider range of backhaul network conditions. In practice, this tolerance helps mitigate poor error handling on devices: while UEs should reconnect after experiencing a 3GPP protocol-level failure, we find that UEs with low-end baseband processors do not do so reliably. When a UE fails to reconnect, the failure manifests as a confusing lack of coverage to people using these devices, and the failure typically only resolves after power cycling the UE. Since Magma terminates GTP locally in the AGW without traversing the backhaul link, a UE never sees a dropped GTP connection and does not have to handle the error.

While agnostic to the radio technology, Magma necessarily makes practical choices about the order of feature development. Many early Magma deployments used LTE, so we have prioritized support for LTE features, with 5G support coming later. A good example is the support of QoS policies. Simple policies to impose rate limits and usage caps, as outlined in Section 2.2, are supported today for both LTE and 5G. More complex policies could be expressed, particularly in 5G, but full support for richer policies is currently under development.

3.2 Hierarchical SDN Control Plane

Magma adopts software-defined networking (SDN) to reduce operational complexity and minimize reliance on skilled staff. Rather than configuring a distributed collection of devices, providers specify network-wide policies at the orchestrator. The orchestrator provides a central point of control and exposes a northbound API for integration with other systems (e.g., for metrics, alerting, and monitoring). However, running the entire control plane in a central controller would impose limits on the scalability of the system. Hence, practical SDN

systems like Network Virtualization Platform (NVP) [37] and Open Virtual Network (OVN) [42] adopted a hierarchical control plane, and this is the model used by Magma.

In a hierarchical control-plane design, we identify those elements of the control plane that have network-wide scope; these are candidates for the central controller. For example, to add a new subscriber to the network, the long-lived information about the subscriber is network-wide information that is created and maintained by the central controller. Conversely, much of the runtime state associated with a UE can be localized to a single AGW. For example, upon becoming active, a UE is associated with a single AGW. The UE's session state can be created and managed by the local control plane of that AGW. Thus, much of the control plane is able to scale out with increasing numbers of base stations and subscribers, rather than increasing the processing in the central controller.

This division between central and local control planes roughly corresponds to the timescale of changes to the control-plane state. The addition of a new subscriber happens on configuration timescales, and that state is managed centrally. The creation of session state—when a UE becomes active and attaches to an AGW—happens more frequently. This runtime state is handled by the local controller on an AGW.

As with any SDN architecture, we must consider "headless" operation, i.e., the situation where a data plane node is disconnected from the central control plane. In a traditional SDN approach, the goal is to ensure that the data plane continues to operate without the control plane, even as updates to the data plane may be impossible while the control plane is disconnected. With a hierarchical control plane, many local operations are still possible even while the central controller is unreachable. For example, an AGW can still establish a session for a UE that attaches to a base station, because the local control plane has enough information (e.g., cached subscriber profiles) to process the session establishment. Conversely, network-wide actions like the addition of users or changes to user policies must wait until the central control plane becomes available again. Magma makes trade-offs for availability versus consistency as the CAP theorem [22, 28] implies. It is generally possible for state stored in an AGW to be stale during times of disconnection, which might, for example, allow a UE to temporarily consume resources beyond its quota.

This design helps to achieve the scaling goals of Magma, in allowing both a small minimum footprint (scaling down) as well as scaling up. A minimal Magma deployment would be a single AGW and an orchestrator. The orchestrator is typically three virtual machine instances in a cloud, while the AGW itself is a small (4-core) x86 commodity server. This is dramatically less hardware than a conventional cellular packet core. Scaling up is essentially a matter of adding more AGWs, which increases the number of base stations and UEs, without much increase in the load on the orchestrator. We discuss our experiences in scaling up in Section 4.3.

The decision to place local control-plane functions on the AGWs, while beneficial for scalability, does introduce trade-offs. In particular, it complicates the picture for some features that require coordination among AGWs. Notably, while Magma supports mobility across radios served by a common AGW, seamless mobility *between* AGWs would require communicating some control-plane state from one AGW to another during hand-offs. While many use cases can be supported without this feature, we expect to add it in the future.

3.3 Fault Tolerance Via Small Fault Domains

The desire to build a low-cost solution for Magma has a significant effect on how the architecture approaches fault tolerance. Low-cost hardware is prone to failure, and so Magma adopts the view common to most modern cloud systems: it is expected that individual components will fail. A failure of a component must affect as few users as possible (i.e., fault domains must be small) and must not affect other components. This approach also has a positive impact on operations such as software upgrades, as it is possible to upgrade small components independently without taking down the whole system. This is in stark contrast to traditional 3GPP implementations.

The SDN-like architecture of Magma localizes state more fully than a typical 3GPP implementation. In a standard implementation, the runtime state of a UE is spread among several large components (e.g., the PGW, SGW, and MME in the LTE case). In contrast, Magma localizes the runtime state of a UE to a single AGW. This simplifies failure handling. The runtime state stored in an AGW is checkpointed regularly and may be copied to a backup instance of the AGW running as a cloud service. When an AGW fails, the backup cloud instance is brought into service, and can manage connections for the affected set of UEs until the primary AGW is restarted. As noted above, an AGW may continue to establish sessions to UEs even when disconnected from the orchestrator. The state synchronization approach described in Section 3.4 mitigates the long-term effects of such failures.

While it is common for a traditional cellular packet core to serve millions of subscribers, Magma distributes much of the functionality to a large number of Access Gateways. Each AGW is thus a fault domain that holds state for a relatively small number of UEs served by a small number (typically less than ten) of base stations. The failure of a single AGW would impact the set of UEs currently served by the attached base stations, but has no impact on the rest of the network or its customers. This contrasts with the much larger fault domains typical of a standard mobile core implementation.

3.4 State Synchronization

State in a mobile core needs to be communicated among components. Generally, one component is the authoritative owner of some piece of state, and it needs to synchronize state

with another component. In Magma, state can take one of three forms, for which Magma makes different guarantees.

The first is *runtime state* associated with a UE and its network activity. Backwards compatibility with existing user devices and RAN equipment requires Magma to implement standards-defined state machines to support operations like connecting to the network; modifications to runtime state can occur due to events in the UE itself, the RAN equipment, or Magma's "core" network elements. Importantly, runtime state within Magma is encapsulated within the AGW, which as discussed in Section 3.3 is the failure domain for Magma, and we assume a crash-recovery failure model for AGWs.[1] Further, most runtime state is both ephemeral and recoverable in the event of failure: a UE can simply reconnect.

The second is the *configuration state*, associated with the configuration of a Magma network element, such as an AGW. This is only ever written by the orchestrator and pushed asynchronously to the AGW. Examples include classes of network policy to be applied to classes of user or radio configuration to be applied by an AGW to connected RAN equipment. AGWs recover configuration state after a crash, and the source of truth for configuration state is stored durably in the orchestrator (Postgres); we only permit modification to configuration state through the orchestrator. Configuration state generally changes on human timescales (i.e., minutes or hours).

Finally, Magma also manages *metrics state*, telemetry from Magma elements. This operational data, while useful, is captured on a best-effort basis.

Like many cloud-native systems, Magma adopts a "desired state" model for runtime and configuration state. By this we mean that to communicate a required state change (e.g., the addition of a new session in the data plane), the desired end state is set via an API. This contrasts with a "CRUD (Create, Read, Update, Delete)" interface, which is common in 3GPP specifications. Magma replaces the CRUD model with the desired state model to simplify reasoning about changes across elements of the system in the case of partial failures. This is a common case in challenged environments, where portions of the end-to-end system (e.g., backhaul) are far less reliable than others (e.g., the link between the UE and the RAN). This is best explained via a simple example.

Consider the case of establishing data-plane state in an AGW for a set of active sessions. Suppose there are two active sessions, X and Y. Then a third UE becomes active and a session Z needs to be established. In the CRUD model, the control plane would instruct the data plane "add session Z". The desired state model, by contrast, communicates the entire new state: "the set of sessions is now X, Y, Z". The CRUD model is brittle in the face of failures. If a message is lost, or a component is temporarily unable to receive updates, the receiver falls out of sync with the sender. So it is possible that the control plane believes that sessions X, Y and Z have been established, while the data plane only has state for X and Y. By sending the entire desired state, we ensure that the receiver comes back into sync with the sender once it is able to receive messages again.

This approach is hardly a novel idea in the cloud-native world, but it differs from typical 3GPP systems. It allows Magma to tolerate occasional communication failures (caused by poor quality backhaul, for example) or component outages due to software restarts, hardware failures, etc. Limiting the scope of 3GPP protocols to the very edge of the network gave us the flexibility to rethink state synchronization to improve fault tolerance (in addition to other benefits noted above).

We close by considering how Magma manages state for one particularly salient policy: billing users based on data volume, and the possibility of double-spending. Volume-based billing policies are typically implemented using a third-party online charging system (OCS) that integrates with both the network operator's existing business support systems (BSS) as well as Magma. In this arrangement, *billing and charging* are handled by the OCS, while Magma handles *metering and accounting*. The OCS tracks a user's account balance (e.g., in US$) and then authorizes small quotas of data (e.g., 1MB) to the user via Magma; when the user nears completion of their quota, Magma requests another quota on the user's behalf from the OCS, which makes the decision on whether to grant or deny the request. Whether or not a user has been allocated a quota is *configuration* state from Magma's perspective, while the amount remaining in a user's current quota is *runtime* state. Thus, while it is possible for a malicious user to double-spend by moving between AGWs strategically, the maximum amount of double-spend permitted is capped as a business decision by the quota size. Operators for whom this is a particular concern could also adopt techniques for volume-based accounting in a distributed context [31].

3.5 Software Data-Plane Implementation

The data plane is responsible for (i) recognizing the flows for active sessions (traffic to and from active UEs); (ii) collecting statistics for those flows; (iii) adding and removing tunnel headers; and (iv) enforcing policies such as rate limits per subscriber. Magma's data plane is implemented using Open vSwitch (OVS) [47]. OVS provides a programmable data plane that is controlled by OpenFlow [39]. While OpenFlow and OVS are convenient implementation choices, they are not fundamental to the architecture. Other options may be used in the future. The important points are that the data plane is highly programmable and implemented entirely in software.

The software implementation of the data plane enables Magma to operate on commodity hardware. While throughput, latency, and jitter of the data plane are important for cellular networks, we have found OVS to offer entirely adequate performance. OVS performance has been well studied and optimized for many years [47]. In Section 4 we evaluate

[1] We generally assume the same for individual AGW software components; per-process state is held externally for most critical services.

the performance of OVS in the Magma context. It is worth noting that other aspects of the system such as backhaul and RAN capacity are likely to have a larger performance impact overall than the data plane within the access gateway.

The "data plane configuration" box in Figure 4 generates the commands necessary to program the data plane with a set of rules to handle the flows of current sessions. Currently, those commands are OpenFlow commands. If OVS were replaced with a different forwarding engine, only the "data plane configuration" component would be affected.

3.6 Federation With Other Networks

To this point we have described standalone deployment of Magma, but it can be deployed in one of three modes:

- *Standalone*: Magma supports an independent network, with all 3GPP control and user plane traffic terminated in the AGW.
- *Local breakout roaming*: Magma *federates* with an existing cellular network, with control-plane traffic terminated externally but user-plane traffic still handled by the AGW and routed directly to or from the Internet.
- *Home roaming*: Magma federates with an existing cellular network, with both control and user-plane traffic terminated in an external network.

Much as the AGW terminates access-specific protocols from the radio network, Magma introduces additional elements to terminate access-specific protocols with an external core network, using a component referred to as a *Federation Gateway* (FeG). The FeG implements 3GPP-defined interfaces to support "home roaming" as well as "local breakout roaming". The latter is made possible in Magma by the fact that rich policy enforcement is provided in the AGW. As an example, an AGW can obtain the policy to apply to a UE by querying the subscriber data base in the federated network, then enforce that policy in the AGW. Signalling traffic between UEs and the MNO core is handled by the FeG service in the orchestrator[2]. User data-plane traffic is tunneled to an analogous component, the GTP Aggregator (GTP-A) which in turn connects to the MNO's existing P-GW.

Unlike the AGW, the FeG and GTP-A are centralized, on-path devices. This serves a practical purpose: traditional MNOs prefer a single point of interconnection between their sensitive core network and "extension" networks [31]. This has scaling implications as discussed in Section 4.3.2.

4 Evaluation

Magma makes a number of fundamental design choices that differ from traditional core network software to improve flexibility and scalability, while supporting rich network policies. The aim is to support *practical* cellular access deployments. To evaluate Magma, we first consider system performance in an emulated environment, and then discuss a large-scale commercial Magma deployment.

4.1 Supporting Typical Deployments

Emulation Testbed Although evaluating Magma's performance in a real deployment is possible at small scale, evaluating scenarios with hundreds of UEs and RAN elements is impractical. Further, extracting data from commercial deployments is challenging due to privacy and commercial considerations. Thus, we instead evaluate Magma using a commercial emulation system, Spirent Landslide [53], which allows us to emulate arbitrary configurations of virtual UEs and RAN elements in a replicable fashion.

For our evaluation, we deployed the most recent stable release of Magma, v1.6.1. We deployed the orchestrator on a cluster of AWS EC2 instances and two AGWs in our lab. The first AGW was a bare-metal AGW on an Intel J3160 quad-core 1.6GHz CPU with 8GB of RAM and four Intel I210 1Gbps NICs. The second was a virtual AGW running with Intel Xeon 6126 2.60GHz, 8GB of RAM, and 2x10G Mellanox ConnectX-3 NICs; we assigned a variable number of vCPUs to the virtual AGW as defined in our experiments below. Both the bare-metal and virtual AGWs were connected directly to the Landslide emulator as well as to the Internet via 1Gbps and 10Gbps links, respectively. We also verified that memory was not a bottleneck for the AGW during our experiments and that all machines in the orchestrator deployment were running well under capacity. Finally, the emulated SIM cards for the emulated UEs were pre-provisioned into the orchestrator and AGW in advance of all experiments, as is typical for network operator deployments of Magma.

Unlike traditional core networks, Magma's AGW is co-located with RAN equipment (for example, at a tower site), and the unit of scaling for Magma is the AGW itself: as operators grow their network, they add both additional RAN capacity (i.e., radio equipment) but also additional "core" capacity (i.e., AGW instances). Since the AGW is an on-path device for all traffic associated with the cell site, the AGW should be provisioned such that site is limited by the capacity of the RAN as the site, rather than the AGW. This is a notable observation that, in part, motivates Magma's design: when co-locating core network functionality with RAN elements, the *RAN is the bottleneck* for performance on a per-site basis.

The recommended (and typical) deployment scenario has roughly one AGW per "cell site", which in practice consists of 1-3 eNodeBs in the case of an LTE network. A typical eNodeB (such as those described in Table 2 or depicted in Figure 2) can support at most 96 simultaneously active users[3] and radio channels of at most 20MHz; this channel capacity, in turn, corresponds to a peak aggregate throughput of

[2]This is necessary to coordinate low-level network state between the UE and the MNO's traditional core, such as GTP bearer identifiers.

[3]More users may be attached but not actively transmitting data.

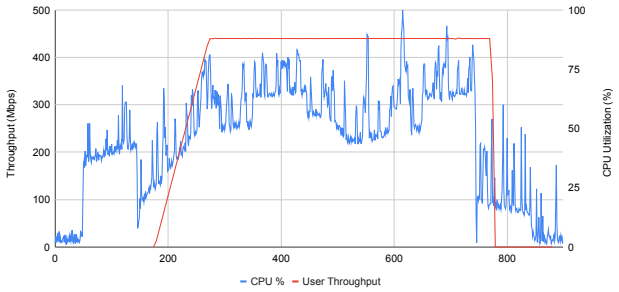


Figure 5: AGW CPU utilization under maximum "typical" workload for a cell site. Aggregate throughput is limited by radio capacity, not the AGW.

Item	Unit Cost	Qty	Total	Notes
LTE eNodeB	US$4,000	3	US$12,000	Baicells Nova 223: 1W, 3.5GHz, 96 user, 2x2 MIMO.
AGW	US$450	1	US$450	Same as used in experiments.
Accessories	US$450	3	US$1,350	18dBi sector antenna, RF cables, connectors, grounding.
RAN CapEx (per site)			US$18,760	

Table 2: Cost breakdown of active RAN equipment for a typical Magma deployment. Excludes site-specific passive infrastructure and backhaul costs.

126Mbps [16] under ideal conditions, for a typical cell site maximum capacity of 378Mbps. We note that the additional cost of an AGW is modest in comparison to the cost of a cell site, similar to the site cost breakdown observed in related work [31]. Although LTE site costs can vary widely and are, in our experience, dominated by non-networking costs such as land, power, and tower (also known as "passive infrastructure"), a representative deployment could consist of the hardware in Table 2; AGW cost represents less than 3% of the cost of active equipment for the site. Power costs can be especially significant in "off-grid" locations, but these are largely driven by the power needs of the radio equipment and hence not greatly influenced by the mobile core implementation. Note the use of solar and battery power in Figure 2.

Magma must be able to support this type of workload. We evaluate this by emulating the peak load of a the cell site described above: a total of 288 UEs connect (or "attach") to the network for the first time at a rate of 3UE/sec, and each then performs a short HTTP download at a rate of 1.5Mbps, for an aggregate total offered load of 432Mbps. Figure 5 demonstrates our results, focusing on the total CPU utilization as well as achieved throughput of the AGW. At a high level, the AGW accepts attach requests from all new users over the course of approximately 1.5 minutes, after which the AGW enters a steady state for the duration that UEs are making HTTP requests. In this experiment, average sustained UE throughput reaches the expected throughput of 432Mbps throughout the duration of the experiment, indicating performance is limited by the RAN, rather than Magma's AGW, as expected.

We acknowledge that other RAN configurations can exist (including vRAN/cRAN arrangements) where many RAN elements are effectively "co-located" to a single point within the operator's network. Magma can be used effectively in these deployments, with one (or more) AGWs allocated to support this range of RAN equipment. However, no deployments at scale of Magma to date have used that configuration (to our knowledge), and Magma can be deployed on any general-purpose compute (e.g., VM or container) alongside this RAN infrastructure to support it. Similarly, a radio vendor could integrate an AGW into the same physical enclosure as a traditional eNodeB for a combined RAN and AGW element.

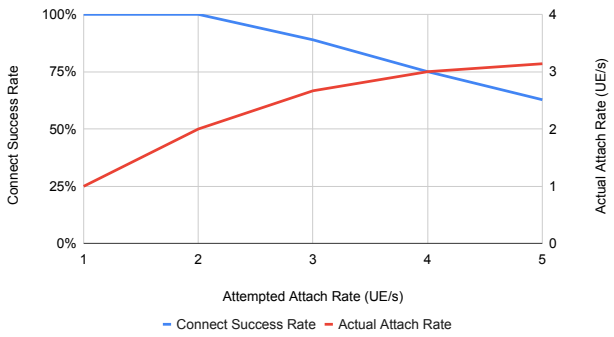


Figure 6: Maximum supported attach rates are limited by the AGW (specifically, the MME component). Results depict a physical AGW.

4.2 Control and User Plane Separation

Different usage patterns of a network stress user plane or control plane elements of the network core: a common example of the former would be human users accessing video content while the latter would be an IoT workload consisting of large numbers of devices that only exchange occasional small messages. This presents a major dimensioning challenge in traditional cellular core networks and motivates efforts to separate control and data plane elements so operators can scale them independently (statically or dynamically); this is known as "control/user plane separation" (CUPS) in LTE and 5G.

Magma's distributed design naturally facilitates a CUPS architecture. By default, every AGW implements a data plane at the network edge, and all control plane functions are implemented as user-space processes at the AGW, with configuration state managed by the orchestrator.

From Figure 5, we observe that the AGW operates in two distinct and characteristic domains. At the start of our experiment, while UEs are attaching to the network, the AGW's CPU workload is dominated by the *control plane* workload associated with handling attach requests, including performing cryptographic operations necessary to authenticate users as well as setting up per-user, per-session state in the data plane and control plane to implement the desired policy for each UE; in our experience, this is the most computationally-intensive control plane procedure. After UEs attach, the CPU workload is dominated by *user plane* workload associated with forwarding UE traffic.

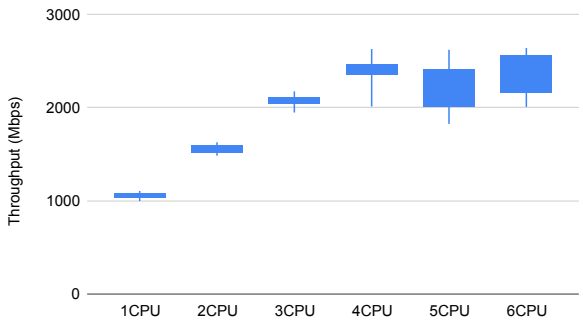


Figure 7: Steady state throughput vs. CPUs allocated to user plane. Note our traffic generator was unable to saturate the virtual AGW's user plane in the 5CPU case and above.

Figure 6 illustrates how our bare-metal AGW copes with a "worst case" control plane workload, a surge of new UEs attaching then saturating the data plane. We define the *connection success rate* (CSR) to be the number of connection attempts that succeed over the total number of connection attempts made, for each five second bin during the experiment. We observe that above 2UE/s, the bare-metal AGW is unable to service all connection attempts, with the connection success rate (CSR) falling linearly beyond this point. On a per-AGW basis, Magma's control-plane performance is relatively limited; improving this is an active area of engineering effort. Attach rate is a function of hardware as well: a 4 vCPU instance of our virtual AGW supports 16 attaches per second, which would saturate the RAN capacity of the "typical" site described above in 18 seconds.

Lastly, we consider per-AGW allocation of resources to the control and user plane. To do this, we statically limit the number of cores available to the user plane and evaluate steady-state throughput and median connection success rate. These results are shown in Figures 7 and 8; note that these experiments use the VM AGW, and as such the absolute throughput numbers are not comparable with earlier experiments. We observe that increasing the cores available to the user plane improves steady-state throughput at the cost of decreased connection success rate (i.e., control-plane performance), but allowing the kernel scheduler to allocate resources flexibly between user plane and control plane tasks provides both high throughput and good connection success rates. We note that we expect raw user-plane performance to increase beyond what is shown here; the commercial test equipment we used was unable to generate more than 2.5Gbps aggregate load.

Taken together, these emulation results demonstrate that Magma can handle typical workloads using low-cost commodity hardware. For more intensive workloads, Magma's control and user plane capacity scales with additional hardware. We finally note that these results provide an upper-bound on the performance of a *single* Magma AGW; the *network* capacity of a Magma network scales linearly with AGWs.

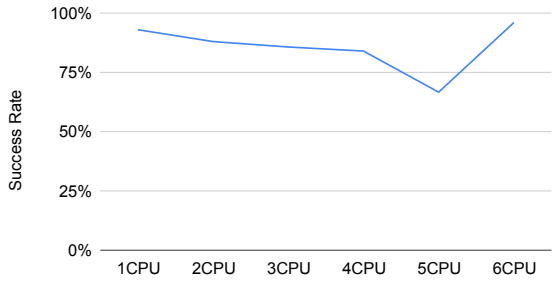


Figure 8: Median connection success rate vs CPUs allocated to user plane.

4.3 Deployment

We now turn to large commercial deployments of Magma. We first note that Magma is an open-source project governed by the Linux Foundation, and as such the core development team (including the authors of this paper) do not directly operate any production deployments; as such, we draw our examples from partners within the project's ecosystem.

Magma adoption. To understand how Magma is used in practice, we interviewed two people working in product management and marketing for the Magma open-source project; in their roles, they speak regularly with operators as well as other commercial entities within the Magma ecosystem. Based on our discussion, as of February 2022, twenty commercial networks were operating using Magma across eight countries in Africa, Asia, North America, and South America. These networks support a range of access modalities and policies. For example, Magma has been used in networks providing backhaul for WiFi hotspots, fixed wireless broadband to homes and businesses, "carrier" WiFi to extend a traditional mobile operator's service to indoor WiFi, and traditional mobile broadband service. Today, Magma has approximately 100 active committers to its codebase.

Magma deployments. To demonstrate how Magma is used, we worked with one of the largest commercial entities, FreedomFi, that provides support to operators deploying Magma. FreedomFi provided data to characterize two significant deployments they help operate. This data was provided to the authors in de-identified form, and only operational data (not user data) was used in our analysis.

4.3.1 Fixed Wireless Hotspots

One of FreedomFi's first commercial deployments was AccessParks [1], a US-based operator that provides public WiFi hotspot networks in large outdoor areas; their deployment locations require multiple WiFi access points (APs) to provide consistent service. With the availability of CBRS spectrum, AccessParks sought to use LTE to provide backhaul to their WiFi hotspots in some of their larger deployments. End users connect to AccessParks's WiFi access points via traditional WiFi mechanisms and an existing captive portal system, and

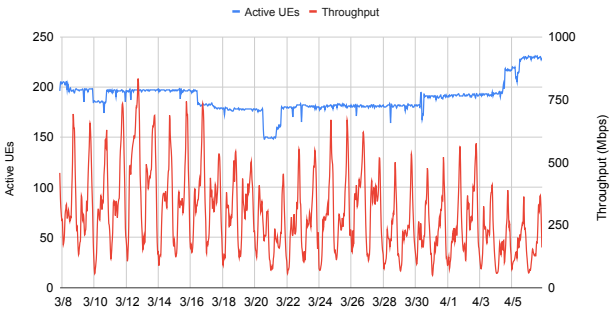


Figure 9: Per-hour AccessParks usage during Mar-Apr 2022.

the UEs in the Magma network are fixed wireless modems that connect the WiFi APs to the Internet via Magma. The setup is illustrated in Figure 10.

AccessParks's deployment began in December 2020 with a ten site pilot to evaluate Magma. Today, the network consists of fourteen sites providing backhaul to over 200 access points, with plans to continue expanding. Figure 9 depicts active subscribers and hourly throughput of the network.

Network policies for the AccessParks networks are very simple: because the LTE network simply serves as backhaul, all UEs simply have unrestricted access. Per-user policies are implemented by AccessParks's pre-existing captive portal and pre-paid billing software, which is implemented using standard techniques (i.e., RADIUS for AAA at the WiFi AP).

Operational complexity. AccessParks's original Magma pilot was motivated in part by their poor experiences with the operational complexity of other commercial and open-source cellular core software in their previous two years of deployment. Although operational complexity is subjective, one quantifiable way in which it manifests is in an operator's labor costs: simpler systems should require less staff time and support to manage. Table 3 shows the results of this comparison for AccessParks. For identical access network infrastructure, AccessParks achieved a 43% reduction in per-site deployment costs using Magma compared to traditional architectures, largely driven by a reduction in support costs and engineering time for site configuration and planning.[4]

4.3.2 Franchised MNO Extension

A second (and, to our knowledge, the largest) deployment of Magma is an early-stage deployment to provide a franchised, neutral host network.[5] This network is unique in that the physical deployment of network infrastructure is not managed by any single network operator. Instead, "micro network operators" (which include individuals, small ISPs, and enterprises) deploy LTE and 5G RAN equipment alongside Magma AGWs that have been customized by FreedomFi to support their proprietary traffic accounting and settlement system.

Item	Traditional	Magma	Difference (%)	Notes
RAN	$7,950	$7,950	-	Identical RAN and backup power.
Core HW	$1,200	$300	-$900 (-75%)	
Core SW	$2,000	$600	-$1,400 (-70%)	Licenses/support.
Field Eng.	$200	$200	-	Installation.
LTE Eng.	$5,000	$330	-$4,670 (-93%)	Planning, core config.
Cost/Site	$16,350	$9,380	-$6,970 (-43%)	

Table 3: Comparison of per-site installed costs for AccessParks's traditional cellular system compared to Magma. Total cost per site decreased by 43%, driven primarily by Magma's reduction in operational complexity for deployment.

Services and Policy. The neutral host network is operated by FreedomFi and allows customers of incumbent MNOs to use this network for service. The core "policy" supported by this network is tunnelling all user traffic back to the appropriate MNO; a user's MNO, in turn, applies their standard network policies for billing, charging, and throttling within their existing core network. The FreedomFi network provides access on a best-effort basis, with each micro network operator leveraging shared CBRS [17] spectrum in the 3.5GHz band (as done in the previous deployment). This service requires integrating the thousands of distributed AGWs with a partner MNO's centralized core network, leveraging the federation capabilities described in Section 3.6.

Scale. As of this writing, this network is still in early testing, so does not have significant user traffic. However, it still provides a useful example of how the Magma *control plane* scales with network size: even without users, Magma still manages device configuration, network monitoring, and supports interconnection with partner MNO core networks.

The FreedomFi network began initial deployments in November 2021, and as of April 2022 consists of 5370 AGWs and 880 eNodeBs (FreedomFi reports the discrepancy between AGWs and eNodeBs is due to supply-chain issues: while AGWs are commodity x86 PCs, cellular radios are specialized equipment with fewer vendors and the ones used in this network only began shipping in January 2022). The network is currently adding on average 150 new AGWs and 90 new eNodeBs per week, all of which are deployed on an ad-hoc basis by micro-network operators; these AGWs are deployed in 45 states across the United States.[6]

Supporting this network is a dedicated orchestrator running on six AWS virtual machines managed by Kubernetes (EKS) [18]. Three instances are dedicated towards "heavy" tasks: operation of the FeG, device configuration, and metrics reporting; these systems are each equipped with 16 vCPUs and 32GB RAM. Remaining orchestrator services run on a

[4] Unfortunately, we do not have data on ongoing maintenance costs from AccessParks; however, AccessParks' decision to use Magma for future deployments suggests it compared favorably.

[5] A neutral host network describes a business model in which a mobile network is operated by an entity for the sole purpose of providing wholesale capacity to third-party retail MNOs and MVNOs; the neutral host network operator does not have its own users, but instead enables users of its customers to use the neutral host network on a shared basis.

[6] The network only operates in the United States for regulatory reasons.

collection of smaller VMs (4 vCPU/16GB RAM). The GTP-A runs on a single bare metal server with a 3.4GHz 8-core Xeon E2278G CPU, 32GB RAM, and 2x10G NICs, and is physically co-located near the facilities of a partner MNO's core network. In total, this costs FreedomFi approximately $4,000 per month to operate.

We view the rapid deployment of this network as cautious evidence for Magma's ability to support large-scale networks with unique business models. We hope to further investigate the operational dynamics of this network in future work.

5 Related Work

Open-source LTE/5G core networks: Several projects share our goal of creating an open-source LTE/5G cellular core network [4, 9, 10, 13]; these were preceded by similar efforts to build open 2G and 3G networks [11, 12]. With the exception of OpenBTS [11] (a GSM-to-VOIP bridge), each of these focuses on implementing traditional, 3GPP-compliant, core networks.[7] Aether [2, 43] is an open-source 5G-connected edge platform, which brings together 5G connectivity and edge-cloud servers. Like Magma, Aether adopts cloud design principles. However, Aether does not refactor the network design to break the coupling of the radio access technology with the core, and Aether does not focus on low-cost equipment to reach under-served users.

Expanding connectivity access: Many efforts have proposed or described novel solutions for expanding Internet access to under-served people [25, 30, 44, 45, 48, 55, 56]. Similarly, small(er)-scale network operators have a rich history providing service to especially rural communities [27], such as community networks [3, 6, 7, 21] and small ISPs [32]. Of this extensive literature, Magma is most closely related to work on community cellular networks [19, 33, 51].

NextG cellular core architecture. The networking research community is actively rethinking the design of next generation networks. PEPC [50] refactors the packet core by consolidating user state into one location, similar in spirit to Magma's AGW. ECHO [40] refactors an EPC to run on less-reliable public cloud infrastructure. SCALE [20] explores an elastically scalable cellular control plane, and KLEIN [49] describes a similarly elastic control and data plane. Although these works all focus on (logically) centralized core networks, the techniques described are complementary to Magma.

Other work takes a more "clean slate" approach to reimagining the cellular core. CellBricks [38] contemplates a highly federated cellular network and moves support for mobility, authentication, and billing into end hosts; it is implemented as an extension to Magma. dLTE [35] makes 4G networks more like WiFi through a decentralized design, including a global registry for peer discovery. SoftCell [34] uses SDN principles to improve the scalability and flexibility of the packet core network. Magma draws on this body of work for inspiration while maintaining a backwards-compatible, standards-compliant edge to facilitate production deployment.

Magma directly builds on recent work exploring core architectures for under-served communities. CCM [31] presents a distributed cellular 2G core that enables semi-disconnected operation over unreliable rural backhaul connections; this work served as an early inspiration for Magma, which extends these concepts to modern wireless access technologies. Similarly, CoLTE [52] provides a lightweight core which—like an AGW—is co-located with RAN elements, but unlike Magma focuses on small, independent community networks.

Open radio access networks: Several recent initiatives focus on opening up the radio access network (RAN). For example, the OpenRAN project [59] and the O-RAN alliance [8, 41] develops standards that disaggregate 3GPP RANs, with open interfaces between the layers. These efforts are complementary to Magma, as they focus on the cellular interface—the part of the network *before* reaching Magma's access gateway.

6 Conclusion

We have presented our experiences in designing and deploying Magma, an open-source platform for building access networks. The most important design decision was to terminate the RAN-specific protocols in access gateways close to the radio. This simple design decision brings many benefits: supporting diverse radio technologies, tolerating disruptions in backhaul links, using a low-cost software data plane, and scaling naturally with a hierarchical SDN control plane. Magma also adopts modern cloud-computing design patterns (e.g., desired-state synchronization, tolerance to failure of individual components) and open-source software components (e.g., gRPC, Open vSwitch, Kubernetes, Prometheus). In line with Magma's goal to enable practical networks, we demonstrated that Magma can support typical deployment scenarios and discussed two large-scale commercial networks that use Magma. Importantly, Magma also scales *down*, with a small minimum footprint that supports incremental deployment, thus filling a gap between traditional WiFi and cellular. All software artifacts for Magma are available on GitHub[8].

Magma was designed with the primary goal of reaching under-served communities, by supporting heterogeneous radio and backhaul technologies and reducing capital and operational cost. We believe that Magma is a good fit for other deployment scenarios, including enterprise 5G networks. Future work on Magma can expand the set of supported features, including seamless mobility between access gateways as well as network virtualization. We look forward to extending the Magma code base, and the community of contributors to the software, so the platform can evolve to serve more users.

[7] We note that the Magma AGW's LTE-specific portion was originally based upon OpenAirInterface [9], as it was the most mature open-source core available at the inception of Magma's development.

[8] https://github.com/magma/magma

Acknowledgements

We thank our shepherd Ranveer Chandra and the anonymous reviewers for their helpful feedback. We thank Boris Renski, Matthew Mosesohn, and Joey Padden for their assistance gathering deployment data for this paper. We also thank the Magma developer and user community for their important contributions, as well as Meta Connectivity for supporting the early development and deployments of Magma.

References

[1] AccessParks. https://accessparks.com/. Retrieved 7/2022.

[2] Aether. https://www.aetherproject.org/. Retrieved 7/2022.

[3] AlterMundi. https://altermundi.net/. Retrieved 7/2022.

[4] free5GC. https://www.free5gc.org/.

[5] gRPC. https://grpc.io/. Retrieved 7/2022.

[6] Guifi.net. https://guifi.net/. Retrieved 7/2022.

[7] NYCMesh. https://www.nycmesh.net/. Retrieved 7/2022.

[8] O-RAN Alliance. https://www.o-ran.org/.

[9] Open Air Interface. https://openairinterface.org/.

[10] Open5Gs. https://open5gs.org/. Retrieved 7/2022.

[11] OpenBTS. http://openbts.org. Retrieved 7/2022.

[12] Osmocom. https://osmocom.org/projects/cellular-infrastructure. Retrieved 7/2022.

[13] srsRAN. https://www.srsran.com/.

[14] Universal Mobile Telecommunications System (UMTS); LTE; Policy and Charging Control (PCC); Reference points. https://www.etsi.org/deliver/etsi_TS/129200_129299/129212/15.03.00_60/ts_129212v150300p.pdf. (3GPP TS 29.212 version 15.3.0 Release 15).

[15] 3GPP: The mobile broadband standard. https://www.3gpp.org/.

[16] 3GPP: Evolved universal terrestrial radio access (E-UTRA); physical layer procedures. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2427.

[17] CBRS Alliance. https://www.cbrsalliance.org.

[18] Amazon Elastic Kubernetes Service (EKS). https://aws.amazon.com/eks/.

[19] Abhinav Anand, Veljko Pejovic, Elizabeth M Belding, and David L Johnson. VillageCell: Cost Effective Cellular Connectivity in Rural Areas. In *International Conference on Information and Communication Technologies and Development*, pages 180–189, 2012.

[20] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. Scaling the LTE Control-Plane for Future Mobile Access. In *ACM SIGCOMM CoNEXT Conference*, pages 1–13. ACM, 2015.

[21] Luca Belli, Sarbani Banerjee Belur, Peter Bloom, Anriette Esterhuysen, Nathalia Foditsch, Maureen Hernandez, Erik Huerta, Mike Jensen, Meghna Khaturia, Michael J Oghia, et al. *Community Networks: The Internet by the People, for the People*. FGV Direito Rio, December 2017.

[22] Eric Brewer. CAP Twelve Years Later: How The "Rules" Have Changed. *Computer*, 45(2):23–29, 2012.

[23] Jyh-Cheng Chen, Ming-Chia Jiang, and Yi-wen Liu. Wireless LAN security and IEEE 802.11i. *IEEE Wireless Communications*, pages 27–36, February 2005.

[24] T. Charles Clancy. Secure handover in enterprise WLANs: Capwap, Hokey, and IEEE 802.11R. *IEEE Wireless Communications*, pages 80–85, October 2008.

[25] Michaelanne Dye, David Nemer, Josiah Mangiameli, Amy S Bruckman, and Neha Kumar. El Paquete Semanal: The Week's Internet in Havana. In *CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[26] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. Cluster-based Scalable Network Services. In *ACM Symposium on Operating Systems Principles*, pages 78–91, 1997.

[27] Hernan Galperin and François Bar. The Microtelco Opportunity: Evidence from Latin America. *Information Technologies and International Development*, 3(2), 2006.

[28] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, jun 2002.

[29] GSMA. Unlocking Rural Coverage: Enablers for commercially sustainable mobile network expansion, 7 2016.

https://www.gsma.com/mobilefordevelopment/resources/unlocking-rural-coverage-enablers-commercially-sustainable-mobile-network-expansion/.

[30] S. Guo, M. H. Falaki, E. A. Oliver, E. A. Oliver, S. Ur Rahman, S. Ur Rahman, A. Seth, M. A. Zaharia, and S. Keshav. Very Low-Cost Internet Access Using KioskNet. *ACM SIGCOMM Computer Communication Review*, 37(5):95–100, 2007.

[31] Shaddi Hasan, Mary Claire Barela, Matthew Johnson, Eric Brewer, and Kurtis Heimerl. Scaling Community Cellular Networks with CommunityCellularManager. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 735–750, 2019.

[32] Shaddi Hasan, Yahel Ben-David, Max Bittman, and Barath Raghavan. The Challenges of Scaling WISPs. In *Annual Symposium on Computing for Development*, pages 3–11, 2015.

[33] Kurtis Heimerl, Shaddi Hasan, Kashif Ali, Eric Brewer, and Tapan Parikh. Local, Sustainable, Small-Scale Cellular Networks. In *International Conference on Information and Communication Technologies and Development*, ICTD '13, pages 2–12, Cape Town, South Africa, 2013. ACM.

[34] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *ACM SIGCOMM CoNEXT Conference*, pages 163–174. ACM, 2013.

[35] Matthew Johnson, Spencer Sevilla, Esther Jang, and Kurtis Heimerl. dLTE: Building a more WiFi-like Cellular Network (Instead of the Other Way Around). In *ACM Workshop on Hot Topics in Networks*, pages 8–14. ACM, 2018.

[36] Matthew William Johnson, Esther Han Beol Jang, Frankie O'Rourke, Rachel Ye, and Kurtis Heimerl. Network Capacity as Common Pool Resource: Community-Based Congestion Management in a Community Network. *ACM Human-Computer Interaction*, 5(CSCW1):1–25, 2021.

[37] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, et al. Network Virtualization in Multi-tenant Datacenters. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 203–216, 2014.

[38] Zhihong Luo, Silvery Fu, Mark Theis, Shaddi Hasan, Sylvia Ratnasamy, and Scott Shenker. Democratizing Cellular Access with CellBricks. In *ACM SIGCOMM*, August 2021.

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[40] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. ECHO: A Reliable Distributed Cellular Core Network for Hyper-Scale Public Clouds. In *Annual International Conference on Mobile Computing and Networking*, pages 163–178. ACM, 2018.

[41] O-RAN Alliance. O-RAN: Towards an Open and Smart RAN, October 2018. White paper, https://www.o-ran.org/s/O-RAN-WP-FInal-181017.pdf.

[42] OVN Architecture. https://www.ovn.org/support/dist-docs/ovn-architecture.7.pdf, 2021.

[43] Guru Parulkar. Aether: An Open Source Platform for Private 5G Connected Edge Cloud-as-a-Service, 2020. Keynote presentation at ONF Spotlight on 5G Connected Edge Cloud for Industry 4.0 Transformation, https://www.youtube.com/watch?v=Zn7FZyiw5KM&t=4s.

[44] Rabin Patra, Sergiu Nedevschi, Sonesh Surana, Anmol Sheth, Lakshminarayanan Subramanian, and Eric Brewer. WiLDNet: Design and Implementation of High Performance WiFi Based Long Distance Networks. In *USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[45] Alex (Sandy) Pentland, Richard Fletcher, and Amir Hasson. DakNet: Rethinking Connectivity in Developing Nations. *Computer*, 37(1):78–83, 2004.

[46] Larry Peterson and Oguz Sunay. *5G Mobile Networks: A Systems Approach*. Systems Approach, June 2020. https://5g.systemsapproach.org/.

[47] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 117–130, 2015.

[48] Thomas Pötsch, Salman Yousaf, Barath Raghavan, and Jay Chen. Zyxt: A Network Planning Tool for Rural Wireless ISPs. In *ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 1–11, 2018.

[49] Zafar Ayyub Qazi, Phani Krishna Penumarthi, Vyas Sekar, Vijay Gopalakrishnan, Kaustubh Joshi, and Samir R Das. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core. In *ACM Symposium on SDN Research*, pages 1–12. ACM, 2016.

[50] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A High Performance Packet Core for Next Generation Cellular Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 348–361. ACM, 2017.

[51] Rhizomatica. http://rhizomatica.org/, 2013. Retrieved 4/2013.

[52] Spencer Sevilla, Matthew Johnson, Pat Kosakanchit, Jenny Liang, and Kurtis Heimerl. Experiences: Design, Implementation, and Deployment of CoLTE, a Community LTE Solution. In *ACM MobiCom*, pages 1–16, 2019.

[53] Spirent Landslide Core Network Testing. https://www.spirent.com/products/mobile-network-testing.

[54] Statista. Internet usage worldwide—statistics & facts. https://www.statista.com/topics/1145/internet-usage-worldwide/#dossierKeyfigures.

[55] Sonesh Surana, Rabin K Patra, Sergiu Nedevschi, Manuel Ramos, Lakshminarayanan Subramanian, Yahel Ben-David, and Eric A Brewer. Beyond Pilots: Keeping Rural Wireless Networks Alive. In *NSDI*, volume 8, pages 119–132, 2008.

[56] William Waites, James Sweet, Roger Baig, Peter Buneman, Marwan Fayed, Gordon Hughes, Michael Fourman, and Richard Simmons. RemIX: A Distributed Internet Exchange for Remote and Rural Networks. In *Workshop on Global Access to the Internet for All*, pages 25–30, 2016.

[57] WBA. OpenRoaming. https://wballiance.com/openroaming/, 2021.

[58] Klaas Wierenga and Licia Florio. Eduroam: Past, Present and Future. *Computational Methods in Science and Technology*, 11(2):169–173, 2005.

[59] Mao Yang, Yong Li, Depeng Jin, Li Su, Shaowu Ma, and Lieguang Zeng. OpenRAN: A Software-Defined RAN Architecture via Virtualization. *ACM SIGCOMM Computer Communication Review*, 43(4):549–550, 2013.

[60] H. Zhu, M. Li, I. Chlamtac, and B. Prabhakaran. A Survey of Quality of Service in IEEE 802.11 Networks. *IEEE Wireless Communications*, pages 6–14, August 2004.

Appendix

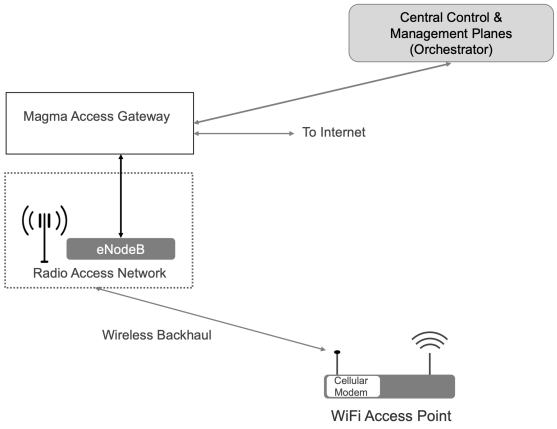


Figure 10: Wireless backhaul to WiFi hotspots provided by Magma. This is the network architecture used by AccessParks in their deployment: end users connect to WiFi access points via standard mechanisms, and traffic is backhauled from the hotspot via a co-located cellular modem to the LTE RAN supported by Magma. Note that nothing in this design precludes an end user from directly connecting to the LTE network, if appropriately configured and allowed to do so by the network operator.

Acronym	Definition
MME	Mobility Management Entity
HSS	Home Subscriber Server
PCRF	Policy and Charging Rules Function
SGW	Serving Gateway
PGW	Packet Gateway
AMF	Access and Mobility Function
SMF	Session Management Function
PCF	Policy Control Function
UDM	Unified Data Management
AUSF	Authentication Server Function
S1AP	S1 Access Protocol
NGAP	Next Generation Access Protocol
SCTP	Stream Control Transmission Protocol
NAS	Non-Access Stratum
RAN	Radio Access Network
LTE	Long Term Evolution
3GPP	Third Generation Partnership Project
UE	User Equipment (a phone or other cellular client)
eNodeB	The “access point” for an LTE network
gNodeB	The “access point” for an 5G network
AGW	Access Gateway
AAA	Authentication, Authorization, and Accounting
RADIUS	Remote Authentication Dial-In User Service

Table 4: Acronyms used in the paper

LinkLab 2.0: A Multi-tenant Programmable IoT Testbed for Experimentation with Edge-Cloud Integration

Wei Dong[†], Borui Li[†], Haoyu Li, Hao Wu, Kaijie Gong, Wenzhao Zhang, Yi Gao[✉]
College of Computer Science, Zhejiang University,
Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China
{dongw, libr, lihy, wuh, gongkj, zhangwz}@emnets.org, gaoyi@zju.edu.cn

Abstract

In this paper, we present **LinkLab 2.0**, a completely programmable and controllable IoT testbed with the support of edge devices and cloud infrastructures. To be more specific, LinkLab 2.0 leverages a tiered architecture for the programmable devices and the management system to achieve scalability. To better support the integrated experiment among IoT, edge and cloud, LinkLab 2.0 provides one-site programming support and leverages the customizable offloading with serverless functions. Moreover, LinkLab 2.0 proposes a device-involved multi-tenancy approach to ensure responsiveness for concurrent requests. Furthermore, targeting 24/7 availability for experimenters, LinkLab 2.0 leverages proactive and reactive anomaly detection to improve the reliability of the testbed. Finally, we describe the supported research experiments and the outreach usage by external users. We also report lessons learned from the four-year operation. LinkLab 2.0 has supported experiments for 2,100+ users. The accumulated usage time across all the devices exceeds 17,300 hours.

1 Introduction

Many modern IoT systems are deeply integrated with edge and cloud platforms. New edge computing platforms like NVIDIA Jetson, and new computing technologies like computational offloading [41, 47] and serverless computing [27, 51] greatly enhance the capabilities of IoT systems [9, 26, 30, 49] and will eventually usher in an era of *Internet of Everything*. For example, in an industrial machine power monitoring scenario [34], hundreds or thousands of IoT devices monitor and collect energy data at a high frequency. To reduce the bandwidth usage and improve the real-time performance, edge devices are usually required to perform data prepossessing and analytics before forwarding the data to the cloud.

However, a major challenge is the lack of a fully programmable testbed for allowing the community to deeply explore new cloud/edge technologies and their "sweet spot" with a *large* number of IoT devices and *highly heterogeneous* computing platforms.

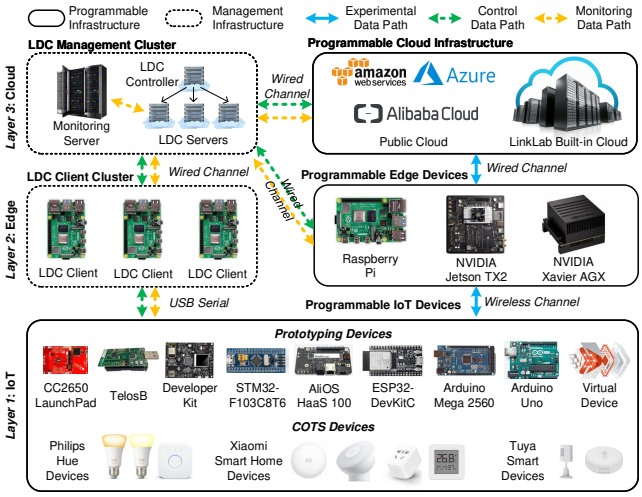


Figure 1: Overview of LinkLab 2.0.

We notice that there exist multiple sensor network testbeds such as MoteLab [67], Indriya2 [7], and FIT IoT Lab [1], allowing the research community to experiment with various sensornet/IoT hardware and IoT software. *Unfortunately*, they do not fully address the aforementioned challenge. Specifically, these testbeds do not natively support edge/cloud integration. Most testbeds do not support the device-edge-cloud communication path and do not allow programming on the edge devices. Moreover, they do not have good support for multi-tenant and high concurrent online experiments with a growing need for teaching and research purposes in the COVID-19 era. In this paper, we present **LinkLab 2.0**, a multi-tenant IoT testbed with edge-cloud integration, aiming to address the following systems and engineering challenges:

(1) How to support device-edge-cloud integrated experiments? Towards this, LinkLab 2.0 enables *one-site integrated programmability and control* for IoT, edge and cloud devices with several front-end and back-end supports. LinkLab 2.0 also supports new computation paradigms by supporting *customizable offloading with serverless functions*. Moreover, the

[†]Co-primary authors

Table 1: Functionality comparison of existing IoT testbeds.

Testbeds	Remote Develop	Edge Support	Cloud Support	Virtual Devices	Web IDE
MoteLab [67]	✓	✘	✘	✘	✘
Indriya2 [7]	✓	✘	✘	✘	✘
FIT IoT [1]	✓	✘	✓*	✘	✘
LinkLab 2.0	✓	✓	✓	✓	✓

*FIT IoT Lab must work with FIT Cloud for cloud-IoT experiments

above programming support requires a separate and reliable channel to deploy experiments onto the devices. Hence, LinkLab 2.0 employs a *vNIC-based bandwidth reservation mechanism* on edge and cloud devices to guarantee the timeliness for controlling the devices (§3.2).

(2) How to ensure dynamic and dedicated usage with a high level of concurrency and multi-tenancy? The *Kubernetes-based architecture* makes LinkLab 2.0 adaptive to highly fluctuating usages (§3.1). Furthermore, considering the concurrent programming requests in the online education scenario, LinkLab 2.0 leverages a *device-involved multi-tenancy* technique to divide a proportion of services and devices as a tenant for dedicated usage. LinkLab 2.0 also provides a nimble configuration interface for administrators to manage the tenants (§3.3).

(3) How to ensure a high level of reliability, especially for the IoT devices? In accordance with the complicated potential root causes of IoT devices, LinkLab 2.0 uses a *proactive and reactive problem detection approach* to detect whether the devices are broken and locate the error as soon as possible. For the whole testbed, LinkLab 2.0 detects anomalies by automatically analyzing the *multi-model logs* during the operation of the testbed (§3.4).

Figure 1 shows the overall architecture of LinkLab 2.0. LinkLab 2.0 consists of three layers: device layer, edge layer and cloud layer. In each layer, there are various programmable devices to facilitate different levels of programmability and control for users. Besides programmable devices, there are dedicated devices and services to manage the programmable devices, namely LinkLab 2.0 Device Center (LDC). There are three different data paths among different layers. The experimental data path is used in the experiments conducted by users. The control data path is used for programming and controlling the devices of LinkLab 2.0, while the monitoring data path is used for experimental data collection and system monitoring of LinkLab 2.0 which is important for guaranteeing 24/7 availability.

Currently, LinkLab 2.0 is equipped with 420+ real IoT/edge devices of 14 different types. Furthermore, LinkLab 2.0 supports theoretically unlimited virtual devices with device-level simulation and a web-based IDE for easier access to the devices. The controller-server-client architecture of LDC allows LinkLab 2.0 to scale easily to accommodate substantial IoT devices at different physical sites. Table 1 compares the functionality of LinkLab 2.0 with other well-known testbeds.

LinkLab 2.0 (`https://linklab.emnets.cn`) facilitates researchers to conduct a broad range of experiments to explore new system designs. It incorporates various embedded computing platforms (e.g., Arduino, ESP32), IoT protocols (e.g., LoRa, MQTT, COAP) and techniques for edge computing (e.g., container-based service composition, edge AI). Furthermore, during the four-year operation, we extend LinkLab 2.0's ability to better serve the community, especially for educational purposes. In §5, we exemplify the supported experiments and outreaches of our testbed to showcase the various capabilities of LinkLab 2.0.

2 Basics and Usage of LinkLab 2.0

Building and managing a testbed with numerous heterogeneous devices from the device, edge and the cloud layer at the same time need a prudent design. In this section, we first present the bird's-eye view and the usage of LinkLab 2.0, then we compare LinkLab 2.0 with the existing testbeds.

LinkLab 2.0 in a nutshell. Towards the aforementioned goal, as Figure 1 shows, LinkLab 2.0 exhibits a three-layer architecture for both hardware and software, namely the IoT device layer, edge layer and cloud layer. LinkLab 2.0 supports both *real* and *virtual* devices for users to program with.

Currently, LinkLab 2.0 includes over 420 real devices for IoT, edge and cloud programming. The IoT devices are deployed in various environments (e.g., multi-hop scenario) as shown in Figure 2. These devices also incorporate various sensing peripherals and networking technologies for users. Another key hardware building block is the programmable edge devices and cloud server (also listed in Table 2), which is currently not well-supported by other testbeds such as FIT [1] and CloudLab [18]. For the programmable cloud server, LinkLab 2.0 provides users with a built-in cloud infrastructure with general-purpose computing resources (i.e., CPU, GPU). Moreover, LinkLab 2.0 supports users to use the public cloud service such as Microsoft Azure or the users' own server as the cloud node in their experiment.

In addition to the real devices, LinkLab 2.0 also introduces *virtual devices* to support the experiments that are *large-scale* or *trace-driven*. We propose two kinds of virtual devices: code-level and message-level. The code-level virtual device accepts the same code as the real node and simulates all behaviors of the device. The message-level virtual device only simulates the network behavior such as MQTT publish, which enables a theoretically unlimited number of devices for large-scale experiments. Furthermore, LinkLab 2.0 also supports binding time-stamped datasets to virtual devices to reproduce an experiment with a pre-recorded trace.

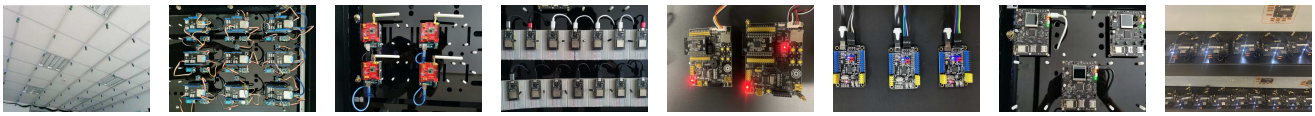
Lifecycle of an experiment. LinkLab 2.0 provides comprehensive support for users to carry out experiments. Conducting experiments contains the following steps:

(1) *Project Creating and Resource Claiming*: Users should first create a project, select the hardware and claim the occupation time via our web portal. The experiment automatically terminates when the requested time quota runs out.

(2) *Programming and Provisioning*: For IoT devices, users

Table 2: List of deployed programmable devices in LinkLab 2.0.

Cat.	Device	ISA	#	Operating System	Wireless	Peripherals/Characteristics
IoT	TelosB	MSP	30	Contiki/RIOT	Zigbee	Temperature, Humidity etc.
	Arduino Mega 2560	AVR	26	Bare-metal/RIOT	WiFi/BLE	Temperature, Humidity, SD Card, etc.
	Arduino Uno	AVR	16	Bare-metal/RIOT	LoRa	LoRa Shield
	ESP32-DevKitC	Xtensa	180	Zephyr/RIOT/etc.	WiFi/BLE	LED
	nRF52840	ARM32	10	Zephyr/RIOT/etc.	Zigbee/BLE/Thread	LED
	STM32 F103C8	ARM32	54	FreeRTOS/RIOT/etc.	LTE	Temperature, Humidity, Light, etc.
	AliOS Things DevKit	ARM32	8	AliOS Things	WiFi/BLE	9-axis IMU, pressure, Mic., etc.
	HaaS100	ARM32	29	AliOS Things	WiFi/BLE/Ethernet	SD Card, LED
	COTS IoT devices	/	11	Philips/Xiaomi/Tuya	WiFi/BLE/Zigbee	Water/Temp./PIR sensor, Plug, Bulb, etc
Edge	Raspberry Pi 4B	ARM64	47	Raspbian Buster (Linux)	WiFi/BLE/LoRa	with 8GB RAM
	NVIDIA Xavier AGX	ARM64	3	Ubuntu 18.04 (Linux)	Ethernet	with AI accelerator
	NVIDIA Xavier NX	ARM64	1	Ubuntu 18.04 (Linux)	Ethernet	with AI accelerator
	NVIDIA Jetson TX2	ARM64	1	Ubuntu 18.04 (Linux)	WiFi/Bluetooth	with AI accelerator
	NVIDIA Jetson Nano	ARM64	8	Ubuntu 18.04 (Linux)	Ethernet	with AI accelerator
Cloud	LinkLab 2.0 Built-in Server	x86_64	1	Ubuntu 20.04 (Linux)	WiFi/Ethernet	with 36-core CPU and GPU



(a) TelosB (b) Arduino Mega (c) Arduino Uno (d) ESP32 (e) nRF52840 (f) STM32 F103 (g) AOS DevKit (h) HaaS100

Figure 2: IoT devices deployment in LinkLab 2.0.

could simply upload the experiment binary via our web portal. Furthermore, LinkLab 2.0 also provides a web-based IDE and online compiling services to allow users to conduct experiments anytime and anywhere. For programming the edge and cloud, LinkLab 2.0 supports both programming with serverless functions and Docker-based development.

(3) *Data Collection Configuring*: The experimental data collection of LinkLab 2.0 is based on logging "channels". Each channel represents a specific category of experimental data. Currently, LinkLab 2.0 provides three channels: console logs, network traffic and energy measurement (for some devices that are connected to a Monsoon Power Monitor).

(4) *Experiment Execution*: Once finished the provisioning, users could start the experiment by clicking the "start" button. During the experiment, users could select one or more devices to view the serial/console outputs instantly and adjust the configurations from the web portal.

(5) *Report Acquisition and Data Processing*: After the execution, users could download the experimental report from the web portal and use data processing tools such as `Python` or `R` to perform further analysis.

Comparison to other testbeds/infrastructure. We compare the development process of LinkLab 2.0 with the one using FIT IoT Lab (for IoT device deployment) and FIT Cloud Lab/Microsoft Azure (for edge/cloud development) in Figure 3 and summarize the differences as follows. (1) LinkLab 2.0 is the only testbed that includes the IoT, edge and cloud, which facilitates users to do a one-stop development. Users are not asked to login with multiple credentials and the inter-device network is automatically configured. Users of FIT IoT Lab will spend a long period configuring the border router and setting up the connectivity between IoT and the cloud. (2) Furthermore, thanks to our integrated programming support (§3.2), users could have a bird's-eye view when selecting devices and the networking parameters are shown when programming, which could shorten the development time. (3) The Web-based IDE and built-in online compilation environment enable the one-key provision of developed experiments. However, using FIT and Azure, developers could only separately write code for the IoT and cloud and manually deploy the program, which both need much coordination efforts between platforms.

3 Designs of Management Services

Managing such a multi-tiered testbed with various heterogeneous devices faces several non-trivial challenges. In this part, we will present the challenges and solutions from the basic architecture design to programming and reliability issues.

3.1 Overview of Management Architecture

As shown in Figure 4, all the IoT, edge and cloud devices are managed by a three-tier architecture named LinkLab 2.0 Device Control (LDC). LDC contains three building blocks: the controller, the server and the client. The LDC controller is the top-level management service, which is responsible for gathering the programming and controlling tasks from the users. The LDC server takes the experiment tasks as input, assigns the tasks to the devices and forward the binaries or configurations to the LDC client (for IoT devices) or directly to the programmable devices (for the edge/cloud). The LDC client is at the lowest level, which directly interacts with the IoT devices and provides device control interfaces (e.g., program, reset, and keep-alive) to the LDC server via the network.

Kubernetes-based management services. The architectural design of LinkLab 2.0 does not happen overnight. We next elaborate on the alternatives and our considerations dur-

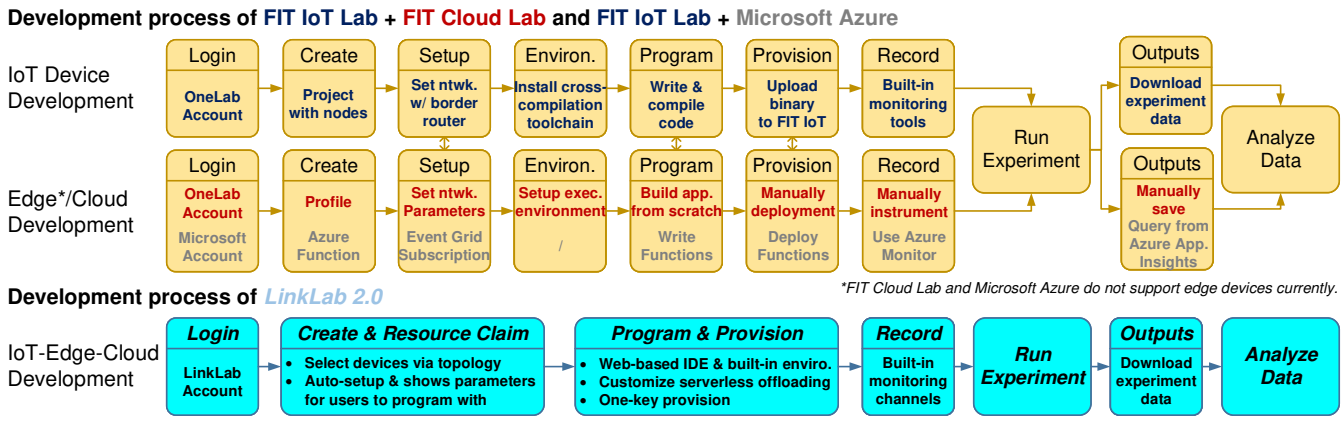


Figure 3: Development process of FIT IoT Lab (for IoT), FIT Cloud Lab or Microsoft Azure (for cloud) against LinkLab 2.0.

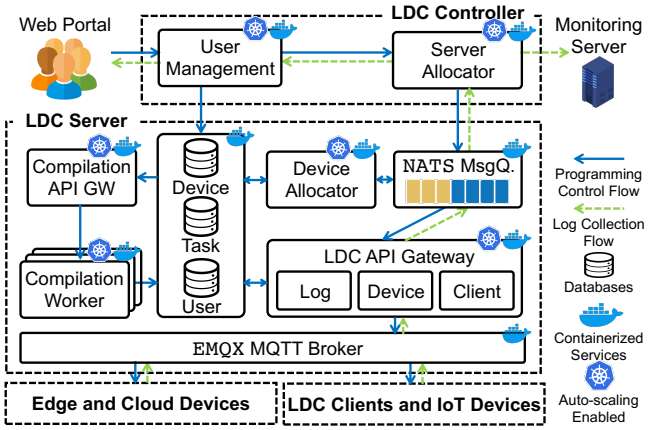


Figure 4: Kubernetes-based management services.

ing the development of such a heterogeneous and large-scale testbed.

Monolithic or cloud-native? In retrospect, LinkLab 2.0 is originally built in a native, monolithic way, which means the management functionalities are centralized in a handful of monolithic services and bare-metally deployed on the server with binaries. Nevertheless, after a few irritating experiences of migrating the services between servers or establishing sub-sites, we decided to adopt a cloud-native architecture (see Figure 4), which means decoupling functionalities to microservices and deploying them with containers. The rationale is we frequently build sub-sites to extend LinkLab 2.0's coverage and community, which makes us value the ease of service management and migration brought by cloud-native more than the extra overhead brought by containerization.

Be adaptive to highly fluctuating workloads. Since one of LinkLab 2.0's usage scenarios is online education, we observe a highly fluctuating workload during the operation of LinkLab 2.0, i.e., many concurrent requests during classes while few users are active at night. Simply over-provisioning the management services is too conservative and uneconomical, hence LinkLab 2.0 leverages Kubernetes for adaptivity.

Kubernetes [53] is an open-source system that enables the automated scaling of containerized services by instantiating service replicas. As Figure 4 shows, all the key services are containerized and managed by Kubernetes (except `NATS`, `EMQX` and databases because they have their own scaling policy). The addition of a *server allocator* makes LinkLab 2.0 scalable for building LDC servers in multiple remote sub-sites.

Benefits. This Kubernetes-based management architecture is scalable to user requests. Once the user request bursts, services shown in Figure 4 will automatically scale up to handle the requests and scale down to save the resources when there are few requests.

Tiered management of devices. Due to the different programming approaches between IoT and edge/cloud devices, LinkLab 2.0 employs tiered management of the devices.

IoT devices. The real IoT devices are connected to a Raspberry Pi (RPI), which the LDC client deployed on, via USB serial. LDC client includes a programming service based on the burning tool provided by the manufacturers of the devices (e.g., `avrdude` for Arduino-series boards). An alternative is using the Over-The-Air (OTA) technology to update the binaries, while we gave up this idea due to the scarce storage space on IoT devices and the wireless interference.

The management of virtual devices is akin to the real nodes. The only difference is the addition of creating and deleting interfaces for users to adjust the number and type of simulated devices, and the managing commands are transmitted via the network rather than USB serial.

Programmable edge and cloud. The management of the programmable edge/cloud devices differs from IoT devices in two ways: (1) LinkLab 2.0 leverages network-based controlling instead of USB serial because the transmission speed of USB serial (115.2Kbps) is generally much slower than the network (>100Mbps), while the data size for provisioning edge device is mainly in gigabyte magnitude. (2) The LDC client is directly deployed on the programmable edge/cloud devices because they have enough computing ability to handle the management commands.

Hence, LinkLab 2.0 develops a runtime for the edge/cloud (Figure 5) to support the programming, controlling and moni-

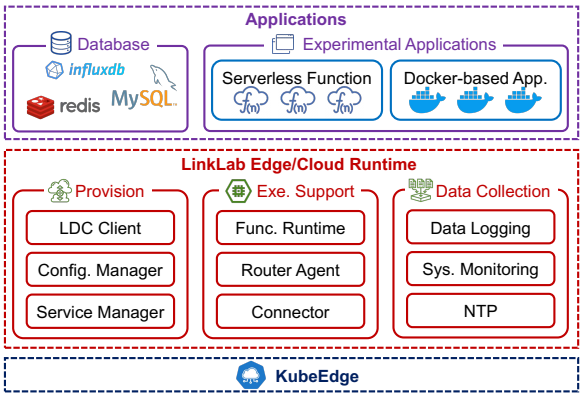


Figure 5: LinkLab 2.0 edge/cloud software stack.

toring tasks. The LDC client is part of the edge/cloud runtime and responsible for handling the commands sent by the LDC server and reports the status of the edge device. Once the LDC client receives an experiment deployment configuration, the *configuration manager* parses it and deploys the experiment. After provisioning, the lifecycle of the services of the experimental application is managed by a *service manager*, including starting, restarting and destroying the instances.

Benefits. This tiered management system as well as the distributed design of LDC client and server dramatically reduces the efforts for adding the devices to LinkLab 2.0. Users who intend to add new IoT devices only need to deploy an LDC client and plug the device into the client. Without these designs, adding devices requires plugging them into the same hardware device that the LDC server lies on, which reduces the scalability for deploying the IoT devices in the wild or physically remote from the LDC management cluster.

3.2 Achieving IoT-Edge-Cloud Integration

We now introduce how LinkLab 2.0 achieves the integrated programming for IoT, edge and cloud and how to guarantee the reliable programming of the multi-layered devices.

Integrated programming for IoT, edge and cloud. The most important feature of LinkLab 2.0 is to facilitate one-site programming for IoT devices, edge and cloud. Moreover, LinkLab 2.0 also supports serverless functions and computation offloading to lower the threshold for experiencing cutting-edge programming paradigms.

One-site programming for the three layers. In order to facilitate the one-site programming for IoT, edge and cloud, LinkLab 2.0 optimizes each step of conducting an experiment. (1) During the selection of devices, LinkLab 2.0 provides users with the hardware network topology. With this view of topology, users could choose devices from different layers with fully aware of the connectivity between devices. (2) For the programming step, users could use a Web-based IDE of LinkLab 2.0 to write code directly in the web browser and leave the compilation and deployment to LinkLab 2.0 backend services. LinkLab 2.0 also automatically handles the network between devices and shows the networking parameters (e.g., IP address) for each device during users' programming. (3) During the experiment execution, LinkLab 2.0 supports the customization of configurations, especially the parameters for inter-layer communication, which could not be easily configured in other testbeds such as FIT IoT Lab. The programmable configurations include connectivity settings (e.g., round-trip time, bandwidth, packet loss rate) and performance settings (e.g., resource quota of services, docker priority).

Customizable offloading with serverless functions. Recent advances in serverless computing [60, 61, 74] allow users to focus on the application logic other than wasting time on the configuring environment and parallelism from scratch. Hence, besides the basic Docker-based development, users of LinkLab 2.0 could decompose their experiment logic into serverless functions and deploy them on edge or cloud devices. Moreover, to further simplify the serverless programming in the IoT-edge-cloud scenario, LinkLab 2.0 provides *device-interaction APIs* for serverless functions to read data from an IoT device.

Based on the serverless functions, LinkLab 2.0 provides systematic support for function offloading. Primarily, users could use `@remotable` annotation to mark the function that could be offloaded. Moreover, a large amount of related research [36, 50] concentrates on the offloading policy. Therefore, LinkLab 2.0 presents a *customizable offloading framework* to enable users to define and test their own offloading policies. Towards this, LinkLab 2.0 first decouples the offloading policy module from the offloading handler, which is responsible to intercept the function execution and transmit it to the offloading destination. Second, LinkLab 2.0 provides customization interfaces for users to define their own policies.

Timely edge control based on vNICs. As we stated before, LinkLab 2.0 uses the network to manage the programmable edge and cloud, which is the same data channel that most experiments use. Suppose an experiment intends to occupy as much bandwidth as it can (which most experiments do), the management delay of this device will dramatically increase, or even the device will stop responding to management commands. Existing cloud testbeds could alleviate this by employing a dedicated network interface card (NIC) for management and control. Nevertheless, most of the COTS edge devices only have one NIC and are not customizable after manufacture.

Hence, we develop a software-based tool, `resGuard`, to ensure the responsiveness of the management service under any circumstances. (1) For outbound traffic, the `resGuard` first uses `cgroup` to categorize the key management tasks (processes) and other user tasks. Then, `resGuard` leverages `tc rate` to guarantee the minimum bandwidth of management tasks. (2) For inbound traffic, however, the aforementioned approach is not applicable because `tc` only implements an egress packet queue and `cgroup` could not classify the inbound traffic to its destination process. Hence, `resGuard` takes advantage of the `ifb` virtual NIC mechanism of Linux.

```yaml
tenant:
  name: NSDI2023
  user: "alice", "bob"
  hardware_exclusive: "AMega-1", "AMega-2", "ESP32-1", "RPI-1"
  hardware_shared: "ESP32-2", "ESP32-3", "RPI-2"
  services: "$all$" # enable all services
  service_quota:
    - compiling: 10 # unit: req/s
    - burning: 5 # unit: req/s
```

Figure 6: Example configuration to create a tenant.

The `ifb` vNIC is a message queue by implementation and any packet that is redirected to `ifb` will return to its original NIC automatically after traffic shaping. Therefore, `resGuard` redirects all the inbound traffic to `ifb` and uses `tc` to prioritize the packet from the IP addresses which host the management services. Note that `resGuard` only prioritizes the management service, the experiments could only utilize all the resources if there is no management task. In addition, `resGuard` also reserves CPU quota for management tasks via `cgroup`.

3.3 Achieving Multi-tenancy

During the COVID-19 pandemic, we make LinkLab 2.0 available to teachers and students for educational purposes. However, this scenario poses new concurrency and multi-tenancy challenges for LinkLab 2.0, which is guaranteeing a dedicated and responsive usage of services and devices during a certain period of time. This is because the programming requests are expected to be handled immediately in a limited class time.

The containerized architecture of our management services (Figure 4) is a good start for achieving multi-tenancy for its scalability and isolation property. However, the programmable devices, which are one of the building blocks of LinkLab 2.0, are not included in this isolation framework. Hence, LinkLab 2.0 proposes *device-involved multi-tenancy*. With this technique, operators could easily create a new tenant for a dedicated usage, and LinkLab 2.0 assures the programming responsiveness of services and devices within a preset quota.

Device-involved multi-tenancy. Services in Figure 4 are reconstructed to support multi-tenancy. First of all, each database owns a `TenantID` field for other services to look up. To avoid the interference of the tasks from different tenants, the device allocator creates waiting queues for each tenant and launches an appropriate number of instances to handle the requests. Other services will subscript the task from `NATS` once they are assigned to a tenant.

Once a new request of a user is received, LinkLab 2.0 will first query which tenant the user belongs to and put the request to the corresponding queue. Then, the device allocator searches for if there are idle devices that meet the user request and belong to the requesting tenant. If so, the request will be assigned to the device. Furthermore, this device-involved multi-tenancy is also a lightweight approach to IoT device virtualization. When creating a new tenant, administrators could assign devices to the tenant in the "exclusive" or "shared" manner. "Exclusive" means the device could only be accessed by the users of the tenant, while "shared" means the device is shared with other tenants. There are primarily sensing and actuation IoT devices in LinkLab 2.0. If a sensing device is set to "shared", we can multiplex its sensing data to multiple tenants if necessary. On the other hand, the actuation devices cannot be shared after being allocated to avoid conflicting operations.

We also leverage an accounting mechanism for the containerized services in Figure 4 to record the resource usage of each tenant. If a tenant exceeds its preset quota, the user management will reject the new request from this tenant.

Tenant management. We build a command-line interface (CLI) for administrators to manage tenants, such as creating a tenant or to moving a device/user to an existing tenant. As Figure 6 shows, administrators could allocate users, hardware resources, and software services with quota in a `YAML` configuration file when creating a new tenant. Then use our management CLI with `LinkLab 2.0-manage tenant -c <config>.yaml` to create (`-c`) a new tenant to LinkLab 2.0. Besides adding new tenants, LinkLab 2.0 also supports modifying (`-m`) and deleting (`-d`) tenants.

3.4 Achieving Reliability

Reliability is the principal requirement of a testbed that is used for both academic and educational purposes. To achieve 24/7 availability, LinkLab 2.0 employs an anomaly detection system. The goal of the monitoring system is to keep two groups of reliability problems away from LinkLab 2.0: (1) Device-related problems, such as unexpected offline and abnormal sensor readings; (2) Service-related problems, such as improper resource occupation and service breakdown.

Proactive and reactive device anomaly detection. Towards the device-related problems, we use both proactive and reactive detection to catch the exceptions as soon as possible.

Reactive detecting. Basically, the devices in LinkLab 2.0 are monitored reactively, which means we only use the data that is non-intrusively collected from the IoT device. Such as we monitor the working state of the peripherals using the readings piggybacked from users' experiment applications.

Proactive probing. To monitor the devices that are used infrequently and improve the coverage of device problem detection, LinkLab 2.0 also employs proactive probing. We create a benchmark set for each kind of IoT device that covers most of the functionalities of the device. Once a device is idled for a period of time (empirically set to 2 hours), our monitoring system programs the benchmarks to the devices and analyzes the output. Note that we make the regular user requests could preempt the execution of probing benchmarks for a better device utilization. Once abnormal behavior occurs, our monitoring system will send an alarm to the operators.

Benefits. Besides alarming the maintainers of device failures, our device anomaly detection approach is also beneficial to the reproducibility of experiments conducted on LinkLab 2.0. To ensure the correctness and accuracy of the experiments, we pay attention to the result consistency and abrasion

of our devices by correlating the piggybacked sensor data in the reactive detection with the environmental data collected by the maintainers and the outputs of proactive probing with our pre-defined ground truth.

Service anomaly detection via multi-model log fusion. In this part, we introduce how LinkLab 2.0 detects service abnormalities through log analysis during system operation.

Multi-model system runtime data collection. The runtime log is an implicit indicator of the system's health. LinkLab 2.0 collects both structured and semi-structured runtime data for further anomaly detection.

(1) Structured key performance indicators (KPIs). KPIs are the quantitative metrics that reflex the operating status of the system. For each layer of LinkLab 2.0 (IoT, edge and cloud), we collect different KPIs according to the intrinsic difference of each layer. (a) For the edge/cloud layer, we collect the network throughput, CPU, memory, etc., as the KPIs. (b) For devices of the IoT layer, the connectivity of each device (indicated by its heartbeat message to the LDC client) is recorded. The KPIs are periodically collected by our monitoring system. The interval is empirically set to 2min in the current deployment.

(2) Semi-structured event logs. Each building block of LinkLab 2.0, especially the management services in Figure 4, reports the underlying behavior of the component via the semi-structured event logs. These logs are mainly service-specific outputs with timestamps and labels of severity levels such as `FATAL` and `ERROR`. Our monitoring system collects these logs from each layer of LinkLab 2.0 for further analysis.

Multi-model log fusion based anomaly detection. The collected KPIs are time-series data, which are eligible for further processing. Nevertheless, the collected semi-structured event logs also need to be structured for anomaly detection. We propose the simFDT approach, which extends the widely-used fixed depth tree (FDT) model with the ability to parse logs in variable lengths using the similarity between logs and templates, to cope with diverse logs generated by heterogeneous devices. After parsing by simFDT, we employ a sliding window to count the different events in a period of time as vectors and feed the vectors to our anomaly detection algorithm.

We employ Autoencoder (AE) to detect abnormal events of our entire system using the KPIs and the vectors parsed by simFDT. AE is widely used for anomaly detection on time-series data [13, 42, 55]. In order to reduce the overhead of transmitting monitoring data, we leverage multi-level detection by separately training anomaly detection AEs for each layer and deploying the models on the nearest upper layer of the devices/services being monitored. For example, the AE for IoT devices is deployed in the LDC client cluster and the AE for the LDC client cluster is on LDC servers.

Operational findings of anomaly detection. We have implemented and deployed the aforementioned anomaly detection approaches to our production environment to achieve 24/7 reliability. Up to now, LinkLab 2.0 achieves 98.2% hardware available time across the IoT, edge and cloud layer since the introduction of this system, and Table 3 shows the detected anomalies. We can see from the table that our anomaly detection approaches could recognize the abnormal states for services, devices and the entire system.

Table 3: Detected anomalies during the operation of LinkLab 2.0 (sorted by the occurrence in descending order). Dev. and Svc. mean the device- and service-related problems, respectively.

#	Type	Root causes of detected anomalies
1	Svc.	Docker images of services are recycled by Kubernetes.
2	Svc.	Failed deployment of a newly developed feature.
3	Dev.	The power supply of an edge/IoT device is broken.
4	Dev.	Loose connection between devices and peripherals.
5	Svc.	Disconnection of the reverse proxy to the cluster.
6	Dev.	The peripheral/pinout of an IoT device is broken.
7	Svc.	Network fluctuations for hours or days.
8	Svc.	Deletion of key files caused by maintainer's misoperation.
9	Svc.	Power outage in the equipment room of LinkLab 2.0

Nevertheless, we observe false positives (FPs) and true negatives (TNs) during the long-term deployment. Here, we will elaborate on the occurrence of FPs and TNs, and discuss how LinkLab 2.0 evolves to cope with the problems.

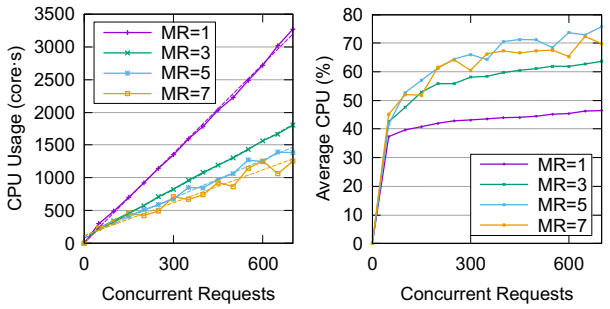
The observed FPs include: (1) The number of testbed usages increases sharply, which leads to the false warning due to unusual high resource usage. (2) The regular operational actions by the maintenance team run out of the resources (i.e., deploying a new version, too many concurrent proactive tests for anomaly detection). (3) The false warning on unprecedented low resource usage after a server upgrade. The rationale behind these FPs is the data sources of the anomaly detection are inadequate. Hence, we subsequently improve the detection system by correlating the detection results with the operational data (e.g., active user, deployment status) and the domain knowledge of operators (e.g., server upgrade).

The TNs are mainly short-time service unavailability (usually for a few to tens of seconds) incurred by the network fluctuation. These TNs could be easily addressed by shortening the interval of KPI collection and proactive probing. We argue that the selection of the interval in the current deployment exhibits a good tradeoff between detecting accuracy and overhead, and we will adjust the interval for the time-sensitive situations such as supporting live classes and exams.

4 System Performance

We test our system to answer the following questions: (1) Is LinkLab 2.0 scalable? (2) Whether the `resGuard` of LinkLab 2.0 is reliable for programming edge/cloud devices and what is the performance? (3) How does the multi-tenant design of LinkLab 2.0 perform?

Effectiveness of the scalable architecture. In this part, we evaluate the scalability of LinkLab 2.0's architectural design by simultaneously issuing hundreds of concurrent requests and see how many resources that used by LinkLab 2.0.



(a) CPU usage. (b) Average CPU utilization.

Figure 7: Resource usage of management services for different concurrencies. MR is the maximum number of service replicas (i.e., the number of containers instantiated to serve the requests of a service) when auto-scaling.

Figure 7(a) illustrates the usage of the CPU. Note that we use the product of CPU utilization and time as the metric of CPU usage in order to illustrate the overall usage during a period of time rather than the instant usage, and we omitted the illustration of memory usage because it shares almost the same distribution to the CPU and page limit. We can observe in Figure 7(a) that the relationship between CPU usage and the concurrent request is *linear* across all the settings, which is scalable because the per-user resource is stable even under extreme concurrencies. Another observation is that the CPU usage seems to be decreasing when the maximum number of replicas (MR) increases. To investigate more on the reason for the decrease, we go into the average CPU utilization of each setting, as Figure 7(b) shows. We can see that the CPU is under-utilized with lower MR, which may be because the tasks have to wait longer when there is no worker replica to process them. Nevertheless, the benefit of auto-scaling also has its upper bound. In our setting, the upper bound is 5 replicas, which we can see in both Figure 7(a) and 7(b), because the management services of LinkLab 2.0 are deployed on a cloud cluster with *five* nodes (each has a 2.5GHz CPU core).

Effectiveness of `resGuard`. In this part, we evaluate the programming reliability of the IoT devices and edge/cloud devices with our three-tiered management system.

For the IoT devices, there are only 504 (1.9%) failed trials in 27,216 programming tasks from May 2020 to January 2021. For edge/cloud devices, we conduct an experiment to evaluate the effectiveness of the `resGuard` we introduced in §3.2. The evaluation methodology is that we attempt to deploy a new experiment on the edge device while another experiment is still running on that device, and we adjust the bandwidth occupation of the existing experiment to see how the deployment time varies under different situations. We use two jobs as new deployment tasks: (1) the user deploys an `InfluxDB` and views the logs, and (2) the user deploys a `Kafka` logging service to the edge device. Figure 8 shows the results. We can see that the deployment time increases rapidly without `resGuard` while the deployment times stay still with `resGuard` when the existing experiment takes up more than 3MB/s bandwidth.

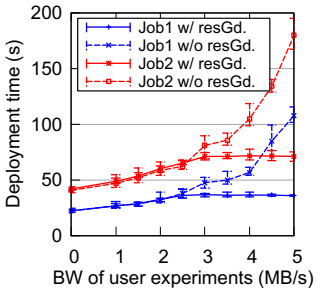


Figure 8: Provisioning time of new experiments when another experiment is running on the edge device with and without `resGuard`.

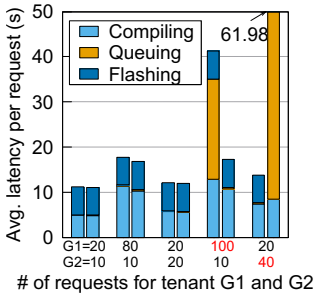


Figure 9: Average programming latency different concurrent request for tenant G1 (80 devices maximum) and G2 (20 devices maximum).

Effectiveness of multi-tenancy. To evaluate the effectiveness of multi-tenancy, we use 100 devices of LinkLab 2.0 and separate them into two tenants: G1 and G2. Then we emit concurrent requests on behalf of the two tenants in different distributions and record the latency per request of each stage in LinkLab 2.0. We set the device quota as: G1 has 80 devices and G2 has 20 devices. Figure 9 shows the results. We can observe that once the request number exceeds the device quota of the tenant (e.g., G1=100>80, G2=10<20), its per request latency increases greatly while the latency of the other tenant is not affected. Moreover, the increased time is mainly spent on waiting for allocation, which is a piece of direct evidence that the tenants would not preempt the resources of others.

5 Representative Use Cases

LinkLab 2.0 facilitates various new usages in addition to the basic wireless and embedded experiments supported by existing IoT testbeds [3, 7, 28, 57]. Furthermore, during the evolution of LinkLab 2.0, we extend it from an academic testbed to an innovative learning and examining platform for schools and individuals. We summarize the representative use cases of LinkLab 2.0 in Table 4 and Table 6 and will elaborate on both the research usages and outreaches in the rest of this section, respectively.

5.1 Supported Research Experiments

Potential research domains. We summarize the potential research domains that LinkLab 2.0 could support in Table 4. Besides traditional wireless and embedded experiments, researchers could conduct experiments with respect to serverless computing, edge AI and other research topics easily and holistically with the integrated architecture, heterogeneous devices and various deployment methods of LinkLab 2.0. Furthermore, with the involvement of edge and cloud, researchers could extend the networking protocol experiments to the IoT domain and obtain data from both the server and client.

While implementing all the research above is beyond the scope of this paper, we internally developed three represen-

Table 4: Research usages of LinkLab 2.0. F1-F4 are LinkLab 2.0's features: F1: distributed and scalable architecture (§3.1), F2: serverless/docker-based development (§3.2, S for serverless, D for docker), F3: offloading (§3.2), F4: multi-tenancy (§3.3).

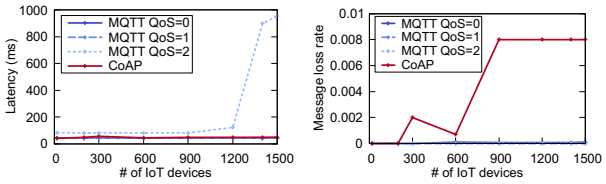
Category	LinkLab 2.0's representative use cases	Scale	LinkLab 2.0's components/features used in researches						
			Cloud	Edge	IoT	F1	F2	F3	F4
Potential Research Domains	Cloud-Edge-IoT integrated application [21,31,73,76]	2+ devices	●	●	●		S, D	◗	
	Wireless and embedded experiments [39,64,77]	2+ devices	◗	●	●		D	◗	
	Offloading algorithms [23–25,40,43]	2+ devices	◗	◗	●		S	◗	
	FaaS and serverless computing [2,5,6,14,17,60,61]	2+ devices	◗	◗	●		S	●	
	Container-based service composition [35,62,63,75]	1+ devices	◗	◗	●		D	◗	
	Edge AI [29,38,45,54,71,72]	1+ devices	◗	◗	●	N/A	N	◗	N/A
	IoT networking protocols [15,37,59]	3+ devices	◗	◗	●		D	❍	
	Industrial Internet of Things [11,32,44,65,69]	3+ devices	◗	◗	●		D	◗	
Example Researches	dSpace: Composable abstractions for smart spaces [21]	5~10 devices	●	●	●		D	❍	
	HRank: AI model pruning for edge computing [45]	3+ devices	❍	●	●		S, D	❍	
	Measurement of different IoT messaging protocol [59]	3+ devices	●	❍	●		D	❍	

[1] ●=all of the cases use the component/feature, ◗=many but not all of the cases use the component/feature, ❍=the component/feature is not used.
[2] N/A means feature F1 and F4 is not applicable for individual usages.

Table 5: Collected data in Edge AI experiment with LinkLab.

Device	Action	Time (s)	Size	Acc.¶
NVIDIA Xavier AGX (w/ accelerator)	Rank generation	590.7	Before: 115MB ⇓ After: 15MB	92.9% (93.9%)
	Model pruning	4,213.5		
	Inference	15.2		
Raspberry Pi (no accelerator)	Rank generation	180.1		93.9%‡ (93.9%)
	Model pruning	–†		
	Inference	385.4‡		

¶ Values in the parentheses are the results presented by the authors of [45].
† Model pruning on Raspberry Pi (RPI) is too long to finish.
‡ Measured using HRank authors' model since pruning on RPI is too long.



(a) End-to-end latency. (b) End-to-end message loss rate.

Figure 10: Collected performance of IoT protocols.

tative experiments that could evaluate the functions and lead the further usage of LinkLab 2.0. Code and tutorials of these experiments are publicly available[1].

Experiment 1: Cloud-Edge-IoT integrated application. We use a programming framework for smart spaces named dSpace [21] as an example to show the potential of LinkLab 2.0 for deploying cloud-edge-IoT integrated research.

Implementation. We directly deploy the open-source dSpace runtime [20] on the cloud. A home automation framework (Home Assistant in our implementation) is deployed on the edge to manage the IoT devices locally and provide device-controlling APIs for the dSpace runtime.

Findings. (1) The original version of dSpace only takes the COTS IoT devices into consideration. With the help of LinkLab 2.0, could explore a larger design space by obtaining more detailed data on network traffics and energy consumption by deploying instrumented codes on the prototyping devices of LinkLab 2.0. (2) The integrated architecture of cloud-edge-IoT and the docker-/serverless-based programming approach of LinkLab 2.0 facilitate users to explore the "sweet spot", and this dSpace case is an ideal example. To be more specific, users could easily move the Home Assistant module between edge and cloud to evaluate the tradeoff between deploying the Home Assistant module on the edge (unreliable edge-cloud connection but shorter local control latency) and cloud (responsive device-controlling APIs but higher device control latency).

Experiment 2: Edge AI. Recently, Edge AI is recognized as one of the emerging technologies by Gartner [22] since it could provide lower latency and better privacy for users. In order to illustrate how LinkLab 2.0 facilitates Edge AI experiments, we conducted an example experiment based on HRank [45] which prunes the AI model to make it feasible to be executed on the resource-constrained edge devices.

Implementation. For model pruning, HRank first generates the rank of each layer of the model by performing the training process with a very small portion of the dataset. Then, HRank prunes the less important filter by retraining the original model with the generated rank. With LinkLab 2.0, users could simply apply for an edge device, deploy the rank generation and pruning algorithm, and evaluate the accuracy degradation of inference with the serverless functions or native dockers. Table 5 shows the comparison of the execution time, model size and inference accuracy of HRank on heterogeneous edge devices, namely Xavier AGX (AGX for short) and Raspberry Pi (RPI for short).

Findings. (1) During our reproduction of HRank, we encountered an abnormal result. It is known that AGX's CPU is more powerful than RPI's, and AGX is also featured with a GPU-like accelerator. Nevertheless, the rank generation time on AGX is much longer than on RPI (see Table 5), which is unusual. After our investigation, we finally found the reason as follows. The HRank code leverages a GPU-based matrix algebra library named `MAGMA` to accelerate the rank generation, but it does not have an ARM distribution for AGX. Hence, on the AGX, HRank must move the intermediate variables during training from GPU to CPU to calculate rank and move back to the GPU, which leads to massive performance degradation. We attribute this interesting finding to LinkLab 2.0's various and heterogeneous edge devices that could be used to obtain a

[1] https://linklab.emnets.cn/tutorials

Table 6: Outreaches of LinkLab 2.0. Similar to Table 4, F1~F4 represent LinkLab's features and S/D represent serverless/Docker.

Category	Outreaches of LinkLab 2.0	Scale	LinkLab 2.0's components/features used in cases						
			Cloud	Edge	IoT	F1	F2	F3	F4
Educational Institutions	E1: IoT curriculum of undergraduates in College A	~40 users/yr.	●	●	●	❍	D	❍	●
	E2: IoT fieldwork courses for students in College B	~90 users/yr.	●	❍	●	❍	S, D	◗	❍
	E3: Joint construction of an IoT Laboratory with University C	~400 users	●	●	●	●	S, D	◗	●
Commercial Cooperations	C1: Online IoT device playground of Merchant D	~150 users	❍	❍	●	●	D	❍	●
	C2: IoT Engineer Certification Exams of Merchant D	~500 users	●	●	●	❍	D	❍	●
Third-party Individuals	T1: Geek developers	1000+ users	●	●	●	❍	S, D	◗	❍
	T2: Self-learners of rudimentary IoT developments		●	❍	●	❍	S, D	❍	❍

●=all of the cases use the component/feature, ◗=many but not all of the cases use the component/feature, ❍=the component/feature is not used in the case.

more thorough view of the performance of the algorithms. (2) In addition to the above experiment, LinkLab 2.0 also allows users to build edge AI applications with the real-world sensing data which could be acquired from our IoT devices and test the performance brought by unstable wireless networks, which are currently not supported by other testbeds.

Experiment 3: Measurement of IoT protocols. Different from the dominance of HTTP for web applications, there is no unique protocol that could serve all the diverse application scenarios of IoT. Hence, conducting measurement studies on IoT protocols under different scenarios is worthwhile, especially before the actual deployment of IoT applications.

Implementation. As [59] illustrates, we compare the two most-used IoT protocols, MQTT and CoAP, using LinkLab 2.0. To make the measurement closer to the real-world application, we leverage the widely-used `EMQX` message broker on the cloud for both CoAP and MQTT. With respect to the clients, we develop the client based on `libcoap` and `paho`. Figure 10 shows the measured end-to-end latency (device-edge-cloud-device) and message loss rate against the increasing number of devices.

Findings. (1) By virtue of the LinkLab 2.0's theoretically unlimited number of virtual IoT devices, users could easily simulate large-scale, real-world IoT applications to evaluate the performance of edge cases. (2) With the help of LinkLab 2.0's bandwidth management, users could measure the performance of protocols under different network conditions.

5.2 Outreaches

We now report the three types of external users for non-academical usages as shown in Table 6.

Educational institutions. The most valued and long-acting outreach of LinkLab 2.0 is supporting the IoT programming practices of the relevant courses in schools, especially after the outbreak of the COVID-19 pandemic.

As shown in Table 4, the usage of LinkLab 2.0 covers regular IoT curriculum (E1) to fieldwork courses (E2). A and B are both technical colleges that teach students both theoretical expertise and practical skills, and students of these colleges mainly choose to serve the local economy for non-academic jobs after their graduation. To cope with this teaching objective, LinkLab 2.0 decides to establish a series of experiments to better train students from various knowledge and socioeconomic backgrounds ready for job markets.

(a) Online virtualization (b) Tabletop model (c) Tabletop schematics

Figure 11: Online-offline integrated smart elderly care education toolkit for College B.

We use our cooperation with college B, which is proficient in serving the local smart elderly care industry, to exemplify how LinkLab 2.0 works with the institutions and builds curricula for their students. To initialize the cooperation, we set up seminars with teachers in college B to introduce the functionalities of LinkLab 2.0 and co-create a basic version of the experiment series that fulfills the knowledge points of the courses. Then we support students for one semester's usage, collect operational data for each experiment (e.g., average completion time, retry counts) and discuss again with the teachers to adjust the difficulty of each experiment or extend the experiments to cope with the changing demands of the job market. The above process is performed recursively in the last three academic years. Up to now, the outcome of our collaboration with College B includes (1) a set of lab experiments for the technical school students who major in IoT, and (2) an online/offline integrated education toolkit for smart elderly care (see Figure 11). The toolkit includes an offline tabletop model containing necessary sensors and actuators for students to realize their ideas on smart elderly care, as well as an online simulated environment to remotely test their programs in a visualized way with the identical program to the offline realization.

Furthermore, thanks to the cloud-native architecture, LinkLab 2.0 also jointly built an IoT laboratory (E3) with University C in a short time, which contains a cabinet of devices with LinkLab 2.0's containerized management software pre-installed. This laboratory is used for the experiments, exams and research of all the IoT-related curricula in University C.

Commercial cooperation. LinkLab 2.0 also draws the attention from commercial institutions. LinkLab 2.0 facilitates an online IoT device playground named HaaS Lab (C1) for Merchant D, which releases a new IoT development board named HaaS (Hardware-as-a-Service). HaaS Lab allows users to try the features of the HaaS board online before buying it

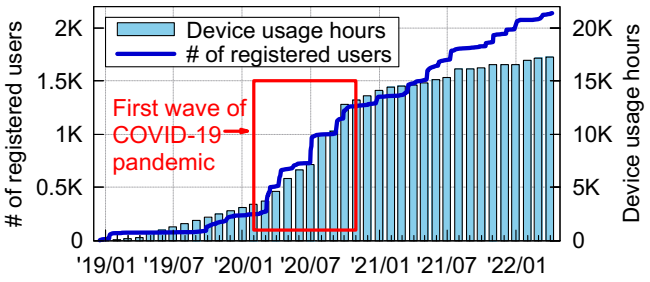


Figure 12: Illustration of the number of registered users and accumulated working hours of devices of LinkLab 2.0.

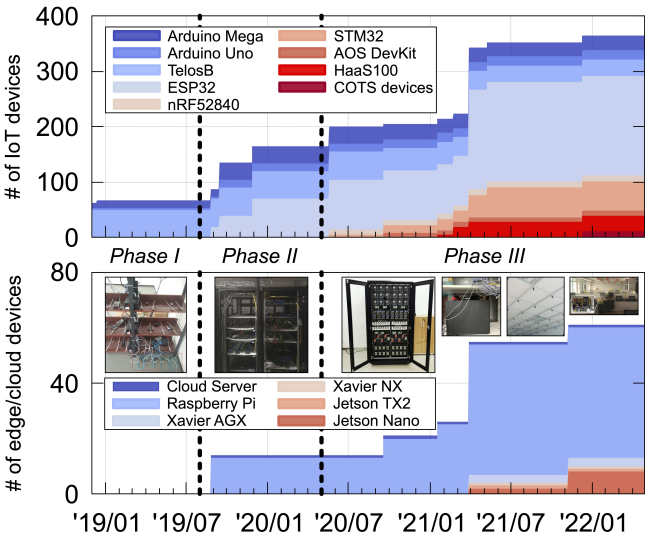


Figure 13: Evolution of LinkLab 2.0.

and Merchant D initiates HaaS Lab to promote its sale. Moreover, LinkLab 2.0 also supports an IoT engineer certification program (C2), which contains a qualifying exam and a lab exam, by creating a dedicated examination tenant.

Third-party individuals. There are 1,000+ *third-party individuals* that use LinkLab 2.0. According to the questionnaire we collected when users are registered, geek developers and self-learners take a large proportion of individual users. The geek developers use our testbed to prototype their new ideas without buying the actual hardware (T1), and the self-learners study rudimentary IoT developments (T2) with LinkLab 2.0.

6 Evolution and Lessons Learned

We are excited to see a broad use of LinkLab 2.0 in diverse scenarios since December 2018. Figure 12 illustrates the number of registered users and the device usage hours of LinkLab 2.0. We can see from the figure that LinkLab 2.0 has over 2,100 registered users and they have conducted 17,300 hours of experiments in total. We also noticed that during the first wave of the COVID-19 pandemic in the first half of 2020, both the new user registration and experiment hours of LinkLab 2.0 increased rapidly. Before January 2020, the concept of online education of IoT is not widely recognized and most of the experiments are occasionally conducted for research purposes.

We have progressively extended the functionality of LinkLab 2.0 since its first public release. According to the different focuses of the LinkLab 2.0's development, we divide the four-year operation into three phases, as shown in Figure 13.

Phase I (2018/12~2019/8): *IoT device testbed.* In this phase, we focus on the basic building blocks of LinkLab 2.0 as an IoT device testbed, such as device management, online compiling, and the web-based programming interface. Because of the diversity of the IoT hardware used in IoT applications, we investigated popular online IoT forums and academic projects, then selected five far-reaching development boards that cover the mainstream IoT ISAs (i.e., `ARM`, `AVR`, `MSP` and `Xtensa`).

Phase II (2019/8~2020/5): *Integration with cloud and edge.* Based on our experience in IoT application development and the requests from LinkLab 2.0's user community, we realized that the cloud and edge were indispensable for a complete IoT application. Therefore, we started to bring cloud and edge devices to LinkLab 2.0 and refactored the management software to cope with this architectural change. In this phase, we continue to increase the number of IoT devices. To better cope with the community's demand and avoid the waste of investments, we leverage an expenditure utilization metric with the device usage per day and the cost of a device to assess how to allocate the purchase budget of hardware.

Phase III (2020/5~now): *Cloud-native and multi-tenancy.* Advancing to the third phase, we extended LinkLab 2.0 with Kubernetes-based scalability and multi-tenancy. Moreover, we continued to enrich the list of supported IoT devices in this phase, such as the nRF52840 and COTS IoT devices.

During this four-year evolution, we have learned a lot while developing and operating LinkLab 2.0 on a broad scale. Hence, we report five lessons learned as follows.

***Lessons Learned* ①: It is not easy to support heterogeneous devices for online experimentation.** Before LinkLab 2.0, we experienced a smooth development process using a sensor network testbed with about 100 TelosB motes. However, it is not a plain sailing when we try to bring more heterogeneous IoT devices to LinkLab 2.0.

For example, ESP32 boards can not automatically enter the programming mode because the flashing signal issued by the uploading software is too short. We finally came up with the idea of adding a 10μF electrolytic capacitor to the `EN` port of ESP32 to lengthen the flashing signal, which works properly. Unfortunately, we failed to attach some boards that need to manually switch to the programming mode or restart the device by pushing an on-device button (e.g., TI CC2650 DK and HiSilicon Hi3861). This manual intervention leads to the human-in-the-loop problem and does not conform to the design principle of LinkLab 2.0.

According to our operational experience, we summarize *a checklist on what device could be attached*, which is: the

devices (1) do not need a manual restart, (2) owns the interface to get operational logs, and (3) support compilation and upload via a command-line interface. We believe this checklist is favorable for the operators of existing or blueprinting testbeds that use IoT devices, and the researchers that intend to conduct large-scale experiments automatically.

Lessons Learned ②: **It is difficult but important to support CI/CD (continuous integration/deployment) as well as online monitoring for providing 24/7 continuous services**. We planned to adopt the CI/CD process at the beginning of developing LinkLab 2.0. However, building a proper testing environment for CI/CD is challenging for LinkLab 2.0. Specifically, a complete testing environment of LinkLab 2.0 includes various devices and cloud servers with the proper software. It is not easy to build such an environment for each developer, especially those who worked remotely in a different city during the pandemic. Therefore, CI/CD was not supported in LinkLab 2.0 during Phase I. However, after experiencing problems like software incompatibilities and devices offline, we added CI/CD support to LinkLab 2.0 in Phase II.

Specifically, we set up a testing environment with dedicated cloud servers and a representative subset of devices for CI/CD. Note that compared with the cloud servers, the IoT/edge devices are much more problematic. Therefore, using a subset of operating devices instead of using dedicated testing devices can significantly increase the similarity between the testing environment and the operating environment. To further avoid the interference of automatic testing to the operating devices, a device will only be selected to act as a testing device without any adjacent tasks.

Even with CI/CD, online monitoring is also important for quality assurance. Device malfunctioning, wireless connection instabilities, and bursty usages are inevitable for an IoT testbed. Hence, LinkLab 2.0 performs online monitoring for all its software components and devices constantly. In case of any problems, alerts will be sent to maintainers automatically.

Lessons Learned ③: **Cloud-native is not optional, but necessary.** As shown in Figure 12, the number of LinkLab 2.0's users surges during the COVID-19 pandemic. Without the containerized architecture, resource management became painful due to the busty usages (e.g., concurrent experiments of a class of students) and various QoE requirements (e.g., time constraints for online exams). Table 4 gives such examples including classes (E1) and exams (C2). Therefore, during Phase III, LinkLab 2.0 became fully cloud-native by containerizing all services and using Kubernetes for on-demand auto-scaling, service provisioning, etc.

An unexpected benefit of becoming cloud-native is that it is one of the necessities to support multi-tenancy. After using LinkLab 2.0 for experiments or teaching, some users started to ask for dedicated devices instead of shared ones, for performance and privacy considerations. Therefore, we extended LinkLab 2.0 to support multi-tenancy (e.g., E3 and C1 in Table 4 are two typical tenants), which was straightforward with cloud-native. With multi-tenancy, we could divide a certain proportion of services and devices into a dedicated tenant to meet various requirements of the usage scenarios.

Lessons Learned ④: **Incorporating open-source projects can not always accelerate the development process.** LinkLab 2.0 builds on top of a large body of open-source projects, such as the Kubernetes for service management and EMQX for message dispatch. However, open-source software is not always ready for out-of-the-box usage, and an active community is profoundly important. For example, we build our web-based IDE based on Eclipse Theia [19], but we have to extend it for interacting with remote IoT devices and many other features, which means massive modifications. With an active community like Theia's, some of the modifications may be inspired by the existing discussions or solved by submitting issues. Unluckily, some modifications are unprecedented to the community and should be conducted by a developer who knows this open-source project well. Nevertheless, gigantic projects like Theia are too complicated to know everything about, especially for student developers.

Lessons Learned ⑤: **Plan for obsolescence.** For a long-term project, the availability of underlying open-source software could change unexpectedly. The first example is an IoT OS named AliOS Things gives up the support of ESP32, STM32 and other boards after a version iteration due to their adjustment of commercial strategy. To make things worse, they also stopped the maintenance of the compilation toolchain for the old version, which influences LinkLab 2.0's online compilation and flashing service. To this end, we managed to rebuild the obsolete tools with public documentation and older versions, which costs much labor work. The second one is the API obsolescence in open-source projects, which occurs when we update Theia from v0.8 to v1.1. After the update, most of our WebIDE components work improperly. Afterward, we check our entire code base for external dependencies, persist the current version to avoid the discontinuance impact, and be cautious when adopting newer versions.

Hence, according to the above five lessons learned, our suggestions for future testbed stakeholders are four-fold. (1) The devices which satisfy the checklist in lessons learned 1 are easier to attach into the testbed. (2) Take fully advantages of the cloud-native micro-service software architecture to better cope with bursty requests and minimize the maintenance overhead. (3) Try to incorporate the open-source software which owns an active community and has been tested by time. (4) Always remember to leave a copy of external resources such as compiling tools to avoid the obsolescence.

7 Future Directions

In this section, we envision the potential future directions and our plans for Phase IV of LinkLab 2.0.

Firstly, we plan to attach more types of devices in LinkLab 2.0. Recently, more kinds of heterogeneous devices are used in edge computing scenario. For example, the FPGA-based edge devices are widely used [68, 70] because they

exhibit excellent performance on accelerating specific tasks while keeping high energy efficiency. However, FPGA devices have limited support for multi-tenancy, which is a key feature of LinkLab 2.0. We plan to borrow the partial reconfiguration approach [33] to provide concurrent programming support and design a shim [16] for our LDC client to manage the FPGAs. Another important group of devices are the boards featured with the trusted execution environment (TEE) such as Arm Trustzone [8]. Such devices are used for secure AI model training [52], sensor data collection [48, 58], etc. Nevertheless, developing a TEE-based application is different from existing development process because it requires to develop the trusted and untrusted part of the application separately and deploy them individually. Hence, we plan to embrace the new programming pattern of TEE-based applications and attach more devices facilitated with TEE to serve a broader research community.

Secondly, we also consider improving the granularity and performance of our monitoring system as the next milestone of LinkLab 2.0. As the number of supported devices and core services grow continuously, configuring our monitoring system becomes more difficult and requires more human-on-the-loop efforts. This is mainly because the current metrics-based and log-based monitoring system is not fine-grained enough to track the micro-service invocation of each request. Recently, the micro-service observability based on eBPF (extended Berkeley Packet Filter) is widely studied both in academia and industry [10, 12, 46]. It can provide more fine-grained monitoring by leveraging user-space programmable kernel extensions. Hence, we plan to incorporate eBPF with our monitoring system to achieve more fine-grained and low-overhead anomaly detection.

8 Related Work

With the proliferation of IoT technology, applications and research of IoT grow rapidly. Several testbeds are proposed to ease IoT research and application developments.

Device-edge/cloud integrated testbeds. FIT IoT Lab testbed [1], proposed by the FIT consortium, is a large-scale wireless sensor network testbed deployed across France. Except for the full programmability of the sensor devices, FIT IoT Lab also provides researchers with energy monitoring and network sniffing infrastructure to obtain the experiment data from multiple perspectives. Another testbed that consists of experiments for both edge/cloud and IoT devices is COSMOS [56], deployed in New York. COSMOS focuses on supporting the experiments of advanced wireless technologies such as mmWave and dynamic spectrum sharing.

LinkLab 2.0 differs from the above testbeds which also include the programmability on both IoT and cloud/edge devices in the following: (1) LinkLab 2.0 is the only testbed that supports the one-site development for IoT, edge and cloud. (2) For edge/cloud experiments, existing testbeds only provide bare-metal development, while LinkLab 2.0 supports both Docker- and serverless-based development. The above two differences accelerate the experiment setup process for the users of LinkLab 2.0.

IoT and sensor network testbeds. MoteLab [67] is a wireless sensor network testbed maintained by Harvard University. MoteLab includes 30 MicaZ motes and a web interface. Indriya2 [7] is a sensor network testbed with 41 TelosBs and 17 CC2650 sensortags that enables users to upload the executable, monitor the outputs of the devices and obtain data from the database. SmartSantander [57] is a city-scale testbed containing thousands of sensors with IEEE 802.15.4 or RFID connections that deploy across Santander city. Users could both work with the sensing data and program the sensors with executables to for their experiments. GioTTO [3, 4, 66] is a campus-scale sensor testbed deployed at scale on the UCSD and CMU campuses across several buildings. With the data accessing APIs and the machine learning layer of GioTTO, users could build intelligent inference applications without writing much code.

Compared with these testbeds, LinkLab 2.0 proposes a novel multi-tiered architecture for managing and controlling IoT devices to achieve high deployment extensibility. Furthermore, LinkLab 2.0 leverages the containerized management services for elastic scaling to enable the concurrent experimentation of multiple users.

9 Concluding Remarks

We introduce LinkLab 2.0, a fully programmable testbed that facilitates the integrated experiment with IoT devices, edge devices and the cloud infrastructures. LinkLab 2.0 achieves multi-tenancy and scalability by leveraging the container-based architecture and the auto-scaling technique of Kubernetes. Moreover, our multi-level monitoring system ensures the 24/7 availability for both the hardware infrastructure and software services. Since its public availability in December 2018, LinkLab 2.0 has supported over 17,000 submissions of experiments from 2,100+ users, and the accumulated usage time across all the devices exceeds 17,000 hours. From the four-year operational experience of LinkLab 2.0 for academia and education, we summarize five key observations and lessons learned concerning the device support, CI/CD process and open-source projects adoption, which we believe is favorable for other projects and testbeds requiring the integration of cloud, edge and IoT.

Acknowledgement

We would like to express our gratitude to the reviewers and our shepherd, Anirudh Badam, for their invaluable advice to improve this work. We also thank all members in the EmNets group for their contributions to this work. This work is supported by National Science Foundation of China under grant No. 62072396 and 62272407, Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under grant No. LR19F020001, and the Fundamental Research Funds for the Central Universities (No. 226-2022-00087). Yi Gao is the corresponding author.

References

[1] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, et al. FIT IoT-LAB: A large scale open experimental iot testbed. In *Proc. of IEEE WF-IoT*, 2015.

[2] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proc. of USENIX NSDI*, 2020.

[3] Yuvraj Agarwal and Anind K Dey. Toward building a safe, secure, and easy-to-use internet of things infrastructure. *IEEE Computer*, 49(4):88–91, 2016.

[4] Yuvraj Agarwal, Rajesh Gupta, Daisuke Komaki, and Thomas Weng. BuildingDepot: an extensible and distributed architecture for building data storage, access and sharing. In *Proc. of ACM BuildSys*, 2012.

[5] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *Proc. of IEEE INFOCOM*, 2020.

[6] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proc. of USENIX ATC)*, 2018.

[7] Paramasiven Appavoo, Ebram Kamal William, Mun Choon Chan, and Mobashir Mohammad. Indriya2: A heterogeneous wireless sensor network (wsn) testbed. In *Proc. of International Conference on Testbeds and Research Infrastructures*, 2018.

[8] Arm. Trustzone for cortex-a. https://www.arm.com/technologies/trustzone-for-cortex-a, 2022.

[9] Mohsen Azimi, Armin Dadras Eslamlou, and Gokhan Pekcan. Data-driven structural health monitoring and damage detection through deep learning: State-of-the-art review. *Sensors*, 20(10):2778, 2020.

[10] Ido Ben-Yair, Pavel Rogovoy, and Nezer Zaidenberg. AI & eBPF based performance anomaly detection system. In *Proc. of ACM ICSS*, 2019.

[11] Jiangfeng Cheng, He Zhang, Fei Tao, and Chia-Feng Juang. DT-II: Digital twin enhanced Industrial Internet reference framework towards smart manufacturing. *Robotics and Computer-Integrated Manufacturing*, 62:101881, 2020.

[12] Cilium. ebpf-based networking, observability, security. https://cilium.io, 2022.

[13] Andrew Cook, Göksel Mısırlı, and Zhong Fan. Anomaly detection for iot time-series data: A survey. *IEEE Internet of Things Journal*, 2019.

[14] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *Proc. of ACM WWW*, 2020.

[15] Jasenka Dizdarević and Admela Jukan. Experimental Benchmarking of HTTP/QUIC Protocol in IoT Cloud/Edge Continuum. In *Proc. of IEEE ICC*. IEEE, 2021.

[16] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of ACM ASPLOS*, 2022.

[17] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proc. of ACM ASPLOS*, 2020.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proc. of the USENIX ATC*, 2019.

[19] Eclipse Foundation. Theia: An open, flexible and extensible cloud and desktop ide platform. https://theia-ide.org/, 2022.

[20] Silvery Fu. Github repository of dspace. https://github.com/digi-project/dspace, 2022.

[21] Silvery Fu and Sylvia Ratnasamy. dSpace: Composable Abstractions for Smart Spaces. In *Proc. of ACM SOSP*, 2021.

[22] Gartner. Trends emerge in the gartner hype cycle for emerging technologies 2018, 2018.

[23] Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proc. of ACM MobiCom*, 2016.

[24] Gaoyang Guan, Wei Dong, Jiadong Zhang, Yi Gao, Tao Gu, and Jiajun Bu. Queec: Qoe-aware edge computing for complex iot event processing under dynamic workloads. In *Proc. of ACM TURC*, 2019.

[25] Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong. Tinylink 2.0: integrating device, cloud, and client development for iot applications. In *Proc. of ACM MobiCom*, 2020.

[26] Dolvara Gunatilaka and Chenyang Lu. REACT: an agile control plane for industrial wireless sensor-actuator networks. In *Proc. of IEEE/ACM IoTDI*, 2020.

[27] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proc. of ACM/IEEE IoTDI*, 2019.

[28] Mehrdad Hessar, Ali Najafi, Vikram Iyer, and Shyamnath Gollakota. TinySDR: Low-Power SDR Platform for Over-the-Air Programmable IoT Testbeds. In *Proc. of USENIX NSDI*, 2020.

[29] Pan Hu, Junha Im, Zain Asgar, and Sachin Katti. Starfish: resilient image compression for aiot cameras. In *Proc. of ACM SenSys*, 2020.

[30] Kang Huang, Wanchun Liu, Yonghui Li, Branka Vucetic, and Andrey V. Savkin. Optimal downlink-uplink scheduling of wireless networked control for industrial iot. *IEEE Internet Things Journal*, 7(3):1756–1772, 2020.

[31] Yutao Huang, Feng Wang, Fangxin Wang, and Jiangchuan Liu. Deepar: A hybrid device-edge-cloud execution framework for mobile deep learning applications. In *Proc. of IEEE INFOCOM Workshops*, 2019.

[32] Hongwen Hui, Chengcheng Zhou, Shenggang Xu, and Fuhong Lin. A novel secure data transmission scheme in industrial internet of things. *China Communications*, 17(1):73–88, 2020.

[33] Zsolt István, Gustavo Alonso, and Ankit Singla. Providing multi-tenant services with FPGAs: Case study on a key-value store. In *Proc. of FPL*. IEEE, 2018.

[34] Akshay Ramesh Jadhav, Sai Kiran MPR, and Rajalakshmi Pachamuthu. Development of a novel IoT-enabled power-monitoring architecture with real-time data visualization for use in domestic and industrial scenarios. *IEEE Transactions on Instrumentation and Measurement*, 70:1–14, 2020.

[35] Kavita Jaiswal, Srichandan Sobhanayak, Ashok Kumar Turuk, Sahoo L Bibhudatta, Bhabendu Kumar Mohanta, and Debasish Jena. An iot-cloud based smart healthcare monitoring system using container based virtual environment in edge device. In *Proc. IEEE ICETIETR*, 2018.

[36] Young Geun Kim, Young Seo Lee, and Sung Woo Chung. Signal strength-aware adaptive offloading with local image preprocessing for energy efficient mobile devices. *IEEE Transactions on Computers*, 69(1):99–111, 2019.

[37] Puneet Kumar and Behnam Dezfouli. Implementation and analysis of QUIC for MQTT. *Computer Networks*, 150:28–45, 2019.

[38] Seulki Lee and Shahriar Nirjon. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proc. of ACM MobiSys*, 2020.

[39] Xinyu Lei, Guan-Hua Tu, Chi-Yu Li, Tian Xie, and Mi Zhang. SecWIR: securing smart home IoT communications via wi-fi routers with embedded intelligence. In *Proc. of ACM MobiSys*, 2020.

[40] Borui Li and Wei Dong. EdgeProg: Edge-centric Programming for IoT Applications. In *Proc. of IEEE ICDCS*, 2020.

[41] Borui Li, Wei Dong, and Gao Yi. WiProg: A WebAssembly-based Approach to Integrated IoT Programming. In *Proc. of IEEE INFOCOM*, 2021.

[42] Nanjun Li, Faliang Chang, and Chunsheng Liu. Spatial-temporal cascade autoencoder for video anomaly detection in crowded scenes. *IEEE Transactions on Multimedia*, 2020.

[43] Yongbo Li, Yurong Chen, Tian Lan, and Guru Venkataramani. MobiQoR: Pushing the envelope of mobile edge computing via quality-of-result optimization. In *Proc. of IEEE ICDCS*, 2017.

[44] Fan Liang, Wei Yu, Xing Liu, David Griffith, and Nada Golmie. Toward edge-based deep learning in industrial internet of things. *IEEE Internet of Things Journal*, 7(5):4329–4341, 2020.

[45] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map. In *Proc. of the IEEE/CVF CVPR*, 2020.

[46] Chang Liu, Zhengong Cai, Bingshen Wang, Zhimin Tang, and Jiaxu Liu. A protocol-independent container network observability analysis system based on ebpf. In *Proc. of IEEE ICPADS*. IEEE, 2020.

[47] Peng Liu, Dale Willis, and Suman Banerjee. ParaDrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Proc. of ACM/IEEE SEC*, 2016.

[48] Tianyuan Liu, Avesta Hojjati, Adam Bates, and Klara Nahrstedt. Alidrone: Enabling trustworthy proof-of-alibi for commercial drone compliance. In *Proc. of IEEE ICDCS*, 2018.

[49] Ye Liu, Thiemo Voigt, Niklas Wirström, and Joel Höglund. Ecovibe: On-demand sensing for railway bridge structural health monitoring. *IEEE Internet Things Journal*, 6(1):1068–1078, 2019.

[50] Yuyi Mao, Jun Zhang, and Khaled B Letaief. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications*, 34(12):3590–3605, 2016.

[51] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *Proc. of IEEE ICDCS Workshops*. IEEE, 2017.

[52] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. DarkneTZ: towards model privacy at the edge using trusted execution environments. In *Proc. of ACM MobiSys*, 2020.

[53] NGINX. Using nginx as http load balancer. https://kubernetes.io/, 2020.

[54] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proc. of ACM ASPLOS*, 2020.

[55] Francisco Lucas F Pereira, Iago Castro Chaves, João Paulo P Gomes, and Javam C Machado. Using autoencoders for anomaly detection in hard disk drives. In *Proc. of IEEE IJCNN*, 2020.

[56] Dipankar Raychaudhuri, Ivan Seskar, Gil Zussman, Thanasis Korakis, Dan Kilper, Tingjun Chen, Jakub Kolodziejski, Michael Sherman, Zoran Kostic, Xiaoxiong Gu, et al. Challenge: Cosmos: A city-scale programmable testbed for experimentation with advanced wireless. In *Proc. of ACM MobiCom*, 2020.

[57] Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, et al. Smartsantander: Iot experimentation over a smart city testbed. *Computer Networks*, 61:217–238, 2014.

[58] Carlos Segarra, Ricard Delgado-Gonzalo, and Valerio Schiavoni. MQT-TZ: Hardening IoT Brokers Using ARM TrustZone:(Practical Experience Report). In *Proc. of SRDS*. IEEE, 2020.

[59] Victor Seoane, Carlos Garcia-Rubio, Florina Almenares, and Celeste Campo. Performance evaluation of coap and mqtt with security support for iot environments. *Computer Networks*, 197:108338, 2021.

[60] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. of USENIX ATC*, 2020.

[61] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proc. of USENIX ATC*, 2020.

[62] Christopher Stelly and Vassil Roussev. Scarf: A container-based approach to cloud-scale digital forensic processing. *Digital Investigation*, 22:S39–S47, 2017.

[63] Timur Tasci, Jan Melcher, and Alexander Verl. A container-based architecture for real-time control applications. In *Proc. IEEE ICE/ITMC*, 2018.

[64] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. Packet-level signatures for smart home devices. *Signature*, 10(13):54, 2020.

[65] Junliang Wang, Chuqiao Xu, Jie Zhang, Jingsong Bao, and Ray Zhong. A collaborative architecture of the industrial internet platform for manufacturing systems. *Robotics and Computer-Integrated Manufacturing*, 61:101854, 2020.

[66] Thomas Weng, Anthony Nwokafor, and Yuvraj Agarwal. Buildingdepot 2.0: An integrated management system for building analysis and control. In *Proc. of ACM BuildSys*, pages 1–8, 2013.

[67] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: A wireless sensor network testbed. In *Proc. of IEEE IPSN*, 2005.

[68] Song Wu, Die Hu, Shadi Ibrahim, Hai Jin, Jiang Xiao, Fei Chen, and Haikun Liu. When fpga-accelerator meets stream data processing in the edge. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1818–1829. IEEE, 2019.

[69] Changqing Xia, Xi Jin, Chi Xu, Yan Wang, and Peng Zeng. Real-time scheduling under heterogeneous routing for industrial internet of things. *Computers & Electrical Engineering*, 86:106740, 2020.

[70] Chenren Xu, Shuang Jiang, Guojie Luo, Guangyu Sun, Ning An, Gang Huang, and Xuanzhe Liu. The case for fpga-based edge computing. *IEEE Transactions on Mobile Computing*, 2020.

[71] Ran Xu, Chen-lin Zhang, Pengcheng Wang, Jayoung Lee, Subrata Mitra, Somali Chaterji, Yin Li, and Saurabh Bagchi. Approxdet: content and contention-aware approximate object detection for mobiles. In *Proc. of ACM SenSys*, 2020.

[72] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. Deep compressive offloading: speeding up neural network inference by trading edge computation for network latency. In *Proc. of ACM SenSys*, 2020.

[73] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. Awstream: Adaptive wide-area streaming analytics. In *Proc. of ACM SIGCOMM*, 2018.

[74] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: Nimble task scheduling for serverless analytics. In *Proc. of USENIX NSDI*, 2021.

[75] Wenzhao Zhang, Hongchang Fan, Yuxuan Zhang, Yi Gao, and Wei Dong. Enabling rapid edge system deployment with tinyedge. In *Proc. of ACM SIGCOMM Posters and Demos*, 2019.

[76] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. Towards efficient edge cloud augmentation for virtual reality mmogs. In *Proc. of ACM/IEEE SEC*, 2017.

[77] Xiao Zhu, Jiachen Sun, Xumiao Zhang, Y Ethan Guo, Feng Qian, and Z Morley Mao. Mpbond: efficient network-level collaboration among personal mobile devices. In *Proc. of ACM MobiSys*, 2020.

Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker

Yinfang Chen, Xudong Sun, Suman Nath†, Ze Yang, Tianyin Xu

University of Illinois at Urbana-Champaign †Microsoft Research

Abstract

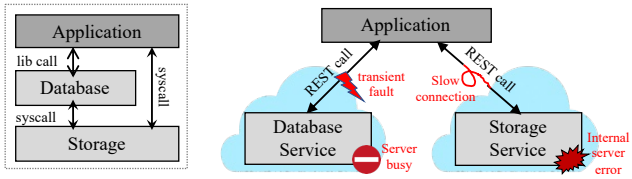
Modern applications have been emerging towards a cloud-based programming model where applications depend on cloud services for various functionalities. Such "cloud native" practice greatly simplifies application deployment and realizes cloud benefits (e.g., availability). Meanwhile, it imposes emerging reliability challenges for addressing fault models of the opaque cloud and less predictable Internet connections.

In this paper, we discuss these reliability challenges. We develop a taxonomy of bugs that render cloud-backed applications vulnerable to common transient faults. We show that (mis)handling transient error(s) of even one REST call interaction can adversely affect application correctness.

We take a first step to address the challenges by building a "push-button" reliability testing tool named Rainmaker, as a basic SDK utility for any cloud-backed application. Rainmaker helps developers anticipate the myriad of errors under the cloud-based fault model, without a need to write new policies, oracles, or test cases. Rainmaker directly works with existing test suites and is a plug-and-play tool for existing test environments. Rainmaker injects faults in the interactions between the application and cloud services. It does so at the REST layer, and thus is transparent to applications under test. More importantly, it encodes automatic fault injection policies to cover the various taxonomized bug patterns, and automatic oracles that embrace existing in-house software tests. To date, Rainmaker has detected 73 bugs (55 confirmed and 51 fixed) in 11 popular cloud-backed applications.

1 Introduction

Modern applications have been emerging towards a cloud-based programming model where applications depend on cloud services for various functionalities. Such "cloud native" practice greatly simplifies application development and deployment, and realizes cloud benefits (e.g., scalability, availability, and cost efficiency). Today, all major cloud providers offer various cloud services to support cloud-based programming, e.g., storage, database, and machine learning [5, 10, 14]. These cloud services have been increasingly adopted, e.g., the .NET SDK of Azure Storage services has tens of thousands of daily downloads [70]. We term the applications that rely on cloud services *cloud-backed applications*.



(a) Traditional applications (b) Cloud-backed applications

Figure 1: Fault domains of (a) traditional applications and (b) cloud-backed applications (the subjects of this paper).

Cloud-backed applications interact with one or more cloud services, usually through REST APIs over HTTP/HTTPS. To ease programming, cloud providers typically offer SDKs on top of the REST APIs to support applications written in different programming languages. For example, AWS provides SDKs in 12 languages, such as .NET, Java, Python, and C++.

Despite the attractive benefits, cloud-based programming imposes emerging reliability challenges introduced by the fault models of opaque cloud backends and less predictable connections between the application and cloud services. Figure 1 compares the fault model of cloud-backed applications with traditional applications backed by local services. Unlike traditional applications that have simple, shared fault domains as the system services with well-specified APIs (e.g., POSIX), the fault domains of cloud-backed applications are more heterogenous, unpredictable, and opaque. It is reported that cloud-backed applications commonly experience transient errors and network delays [22, 42, 81, 88].

In this paper, we unravel the reliability challenges faced by cloud-backed applications. We show that there is a lack of standards and consistencies of existing cloud services on what errors are communicated by cloud service APIs, and how SDKs handle the errors. As a consequence, it is challenging for application developers to anticipate and correctly handle myriad faults that could occur during the application's interaction with the cloud services, resulting in critical bugs. For example, many SDKs employ automatic retries to handle transient errors; however, retries on non-idempotent APIs, if not done correctly, could result in elusive behavior, such as silent semantic violations and unhandled exceptions (see §3).

Contributions. We take a first step to address the emerging reliability challenges by building a "push-button" reliability testing tool named Rainmaker, as a basic SDK utility for any cloud-backed application. Rainmaker helps developers

easily and *systematically* test their applications' correctness, in the face of various errors under the cloud-based fault model. Rainmaker does not need developers to write policies, oracles, or test cases. It directly works with existing test suites and is a plug-and-play tool for existing test environments. Rainmaker is generic to any type of cloud-backed applications.

Designing Rainmaker is challenging. Despite the rich literature on fault injection techniques and error-handling analysis (see §7), we find that no technique can support a push-button solution for cloud-backed applications. Many existing tools provide only the basic randomized fault injection policies and basic crash oracles [6, 37, 47] that can miss critical bugs, and application-specific techniques are not widely applicable (e.g., checking data consistency for databases [16, 52] and file systems [20, 69]). Techniques that address program faults (exceptions and errno) and component faults (e.g., node crashes) are too coarse-grained to capture the nuances of complex interactions between the application and cloud services.

Rainmaker puts its focus on *transient* errors faced by cloud-backed applications. Basically, Rainmaker injects transient faults (e.g., temporary service unavailability and request timeouts) by intercepting outbound REST API calls from the application to the cloud service at the HTTP layer. HTTP-layer interception makes it easy to capture fine-grained interactions (including those triggered by SDK-level retries) and is transparent to applications under test. Hence, Rainmaker requires no modification of application source code and can be directly applied to an existing test environment.

A key component of Rainmaker is its *automatic* fault injection policies that define 1) *what* faults to inject and 2) *where* (e.g., at which REST calls) to inject faults. The former determines the effectiveness and validity of the injected faults, while the latter also affects test efficiency. Rainmaker's fault injection policies are guided by a bug taxonomy we developed to describe how error handling could go wrong under the cloud-based fault model. The taxonomy is simple: it considers transient error(s) that can occur during *one* REST API call initiated by the application (and the corresponding retries by the SDK); yet, it captures common bug patterns and shows that error (mis)handling of even one REST call can have major impacts on application correctness. Rainmaker uses a small set of injections to cover all taxonomized bug patterns.

Rainmaker also enables efficient testing to achieve high testing coverage with a small number of test runs. Rainmaker employs automatic dynamic instrumentation to record the application's calling context of each REST API call and to inject call-site information in the HTTP header of outgoing requests; Rainmaker's HTTP-layer fault injection uses the information to selectively inject faults based on a desired code coverage metric. The calling context also enables Rainmaker to build diagnosis support to help developers debug application behavior under fault injection (when a bug is detected).

Finally, Rainmaker includes automatic oracles to flag a fault-injection test outcome as a likely bug, with low false positives. The oracles utilize exceptions and assertions of existing software tests. For exceptions, Rainmaker does not naïvely report any exception that fails a test as a bug, but checks whether the exception is consistent with the injected fault—an inconsistent exception indicates that the fault was handled intentionally, but inappropriately (at least insufficiently).

Key results. We have implemented Rainmaker for .NET applications. Rainmaker supports a number of cloud services: Azure Storage (including Blob Storage [7], Queue Storage [11], and Table Storage [12]), Azure CosmosDB [9], AWS Simple Storage (S3) [1], and AWS Simple Queue (SQS) [4]. Supporting a new cloud service only takes the configuration of the SDK API namespace and the request-ID tag. Rainmaker is fully transparent to the application under test. We evaluate Rainmaker with 11 popular .NET applications that use the supported cloud services. Rainmaker found 73 new bugs in total, among which 55 have been confirmed, and 51 have been fixed (after we reported them). Many of the bugs have severe consequences, such as unexpected application termination, data loss/inaccessibility, and resource leaks. Rainmaker's test oracles are mostly accurate, with a very low false-positive rate (1.96%), making its test results trustworthy.

Summary. The paper makes the following contributions:

- We unravel emerging reliability challenges of cloud-based programming, faced by cloud-backed applications, under the existing design of cloud service APIs and SDKs;
- We present a taxonomy to systematically understand error-handling bugs that render cloud-backed applications vulnerable to transient errors under the cloud-based fault model;
- We develop Rainmaker, the first push-button reliability testing technique for cloud-backed applications, which can effectively and efficiently detect bugs of myriad patterns;
- We have made Rainmaker publicly available at `https://github.com/xlab-uiuc/rainmaker`, with instructions to reproduce all discovered bugs.

2 Background and Motivation

We discuss the emerging reliability challenges faced by cloud-backed applications as the background and motivation of our work. Ideally, cloud-based programming should *not* be different from traditional application programming using native libraries. Unfortunately, as we will show in this section, this is rarely the case in practice—handling errors under the cloud-based fault model is challenging and error-prone.

2.1 Errors in Cloud-backed Applications

There are three key components related to how cloud service related errors are exposed to the applications.

Error responses from cloud services. A request from the application can fail due to a client-side error (e.g., local network timeout) or a service-side error (e.g., temporary service

SDK	Retry (HTTP Status Codes)	(API)	Notes
Azure Storage (Blob/Queue/Table)	408, 429, 500, 502, 503, 504	Any	Only 429 and 503 are retried before v12.3.0
Azure CosmosDB (HTTP mode)	403, 404, 408, 503	Read	Only enabled for multi-region (no retry for single-region)
AWS S3 / AWS SQS	500, 502, 503, 504	Any	Inconsistencies in different lang. SDKs (e.g., Java versus .NET)

Table 1: The retry policies of different cloud services. Besides the HTTP status codes, SDKs could also retry on error messages, e.g., the Azure Storage SDK also retries on messages including `InternalError`, `OperationTimedOut`, and `ServerBusy`.

unavailability). In the latter case, the service returns an error response indicating the error type. Most cloud services are RESTful, and their APIs reuse HTTP response status codes defined by the HTTP/1.1 standard. HTTP status codes 4XX and 5XX indicate "client errors" and "server errors" respectively. In addition, a cloud service can include service-specific error codes in response payload to indicate fine-grained error types. For example, Azure Blob Storage can return 44 different error codes (e.g., `BlobImmutableDueToPolicy` and `BlobAlreadyExists`) with the same HTTP status code 409 (`Conflict`) [8]. The practice is also used by other cloud services, e.g., CosmosDB [15], S3 [3], and SQS [2]. Applications use these codes to understand the nature of the errors and take error-handling actions accordingly.

Retry on transient errors. When a request fails due to a *transient* client- or service-side error (e.g., network timeout and server overload), an application may retry the request, hoping that it would eventually succeed. Cloud-backed applications mostly use the SDKs provided by the cloud service providers to interact with the cloud services. Besides offering easy-to-use, expressive APIs, SDKs also include error handling logic with the goal of providing a native programming experience. For example, when a REST API call to a cloud service fails due to a transient error, the SDK tries to mask the error from the application by retrying the request [67].

Propagating errors up to the application. If the retry efforts fail or if the error is of a *permanent* type, the SDK propagates the error up to the application in a way that is consistent with a native programming experience. For example, .NET and Java SDKs propagate errors to applications as exceptions.

2.2 Emerging Reliability Challenges

2.2.1 A lack of standards and consistencies

Our analysis of multiple cloud service APIs and SDKs from Azure and AWS reveals a lack of standards and behavior consistencies in all three components above, across cloud services and SDKs from the same/different cloud providers.

Unanticipated errors. We observe many undocumented and inconsistent error codes returned by cloud services. For example, as of September 2022, common HTTP error codes such as 408 (`RequestTimeout`) and 429 (`TooManyRequests`) that can be returned by Azure Table Storage [78] are absent from its official documentation [66]. We also observe that the same error can be represented by different error codes across services. For example, Azure Queue Storage represents the error `QueueNotFound` by the code 404, while AWS SQS uses the code 400 for the same error. We even reported and fixed multiple typos in error messages in Azure Storage SDKs.

Second, whether an error will be masked by the SDK (with retries) and whether an error will be propagated to an application vary widely across different services, different versions of the same service, and different language support of the same version. Table 1 shows that SDKs of different cloud services implement different retry policies. Azure Storage .NET SDKs before v12.3.0 only retry on error codes 429 and 503; the later versions add retry for 408, 500, 502, and 504 [28]. The .NET SDK for AWS retries on `LimitExceededException`, but the Java SDK does not. Finally, whether an error is propagated to the application varies even across SDKs from the same provider. The `DeleteMessage` API of Azure Queue propagates an exception to the application when a 404 is returned by the service. On the other hand, the `DeleteEntityAsync` API of Azure Table silently ignores 404 errors and returns success.

The inconsistencies make it hard for developers to anticipate whether a cloud service will return a specific error and whether the SDK will propagate it to the application.

Note that, unlike libraries and system services for traditional applications, a cloud service neither is a part of the application nor has standard APIs such as POSIX.

Retry and non-idempotent APIs. While retry is a common practice to mask transient errors, the retry may introduce subtle errors. In particular, a retry on a non-idempotent cloud service API can cause elusive effects, such as remote data corruption, which can remain latent or lead to additional errors (more details in §3). However, we observe that each service implements different retry logic, with no standard practice or discipline. As a result, the semantic of SDK APIs, under error conditions, is often opaque and inconsistent.

2.2.2 Rarity and large space of faults

Developers often miss error-handling bugs with small-scale, short-duration functional tests because cloud-based faults, which could expose such bugs, are rare. Fault-injection tools allow developers to simulate error scenarios during testing. However, although it is not hard to implement fault-injection mechanisms [35, 44], the key challenges lie in specifying *policies* about what faults to inject, where and when to inject them, and *oracles* about what post-fault conditions indicate a likely bug for developers to inspect. The space of possible fault policies and oracles is large, if not infinite.

2.3 Our Goal

Our goal is to address the emerging reliability challenges faced by cloud-backed applications by (1) systematically understanding the bug patterns, and (2) building practical tooling to systematically test whether a cloud-backed application can correctly handle the myriad errors that may happen during interactions with the cloud services it depends on. We emphasize a "push-button" technique that can be directly used by application developers in an existing testing environment, without the need of writing additional code or configurations.

3 Bug Taxonomy

To systematically understand the patterns of error handling bugs of cloud-backed applications and to guide the design of the Rainmaker tool, we develop a bug taxonomy to describe how error handling could go wrong under the cloud-based fault model. Figure 2 depicts the bug taxonomy as a tree. An important trait is that the taxonomy considers transient error(s) that occurred during *one* REST API call interaction initiated by the application. It does not reason about multiple independent REST API calls.

3.1 No Error Handling

In this pattern, the application simply does not handle certain transient errors. This can happen due to the inconsistencies mentioned in §2.2.1. An application may not anticipate a cloud service to return a specific error or may mistakenly expect the SDK to mask a known transient error. For example, an application using Azure Storage's .NET SDKs before v12.3.0, which do not retry on certain transient error codes, may mistakenly assume that *all* transient errors are masked by the SDK and hence not handle them. As a result, some transient error from the cloud service can lead to application crashes or other undesirable behavior.

3.2 Throwing Unrelated Exceptions

In this pattern, (mis)handling of an error results in a new unhandled error that is usually *unrelated* to the root-cause fault. A common example of this pattern involves request retries. When a request to a cloud service fails with a timeout, the SDK or the application cannot determine whether the timeout happened on the request path or on the response path. SDKs commonly treat timeout as a transient failure and retry. However, if the timeout happened on the response path (in which case, the original request was executed by the service successfully), the retry can fail with a new error (different from the original error) since the original request has invalidated the precondition of the retry. This new error, if not handled properly, can lead to undesirable effects.

Figure 3 shows an example of such bugs [32] detected by Rainmaker in Microsoft BotBuilder [17]. BotBuilder stores logs in the Azure Blob Storage service. Each log operation

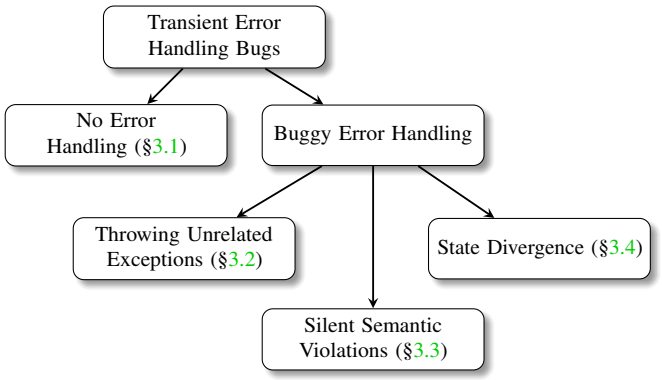


Figure 2: Taxonomy of error handling bugs in cloud-backed applications. The taxonomy addresses the handling logic of transient error(s) that occurred during the interaction of one REST API call.

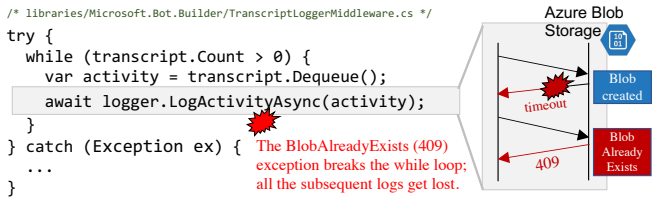


Figure 3: A bug of throwing unrelated exceptions in Microsoft BotBuilder detected by Rainmaker (confirmed and fixed). The Azure Storage SDK automatically retries on timeouts, which returns a 409 error because its precondition is invalidated by the first request.

calls an SDK API to create a new blob. If the API call successfully creates the blob, but the response times out, the Azure Storage SDK automatically retries the request (§2). However, since the blob has already been created by the first request, the retry operation fails with a 409 (`BlobAlreadyExists`) error. The SDK propagates this *permanent* error to the application. BotBuilder does not anticipate or handle the error. This breaks the execution of a loop that is supposed to upload a list of logs to the Blob Storage service, resulting in a loss of subsequent log data. Note that the exception seen by the application is unrelated to the root cause (a transient timeout).

In the next two categories, the buggy error handling does not immediately throw an exception. Rather, it causes unexpected (local or remote) state changes that may cause visible symptoms (e.g., exceptions) during subsequent execution.

3.3 Silent Semantic Violations

In this pattern, mishandling of a transient error causes semantic violations of the REST API specification, without observable symptoms. The REST call returns successfully, and hence the application executes in a happy path. However, the silent semantic violation may eventually result in data loss/corruption, or other incorrect application behavior. One common example of this pattern is manifested by a similar root cause as the one in §3.2: response timeout. Differently,

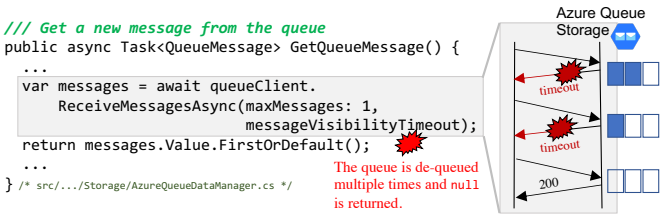


Figure 4: A silent semantic violation bug in Microsoft Orleans detected by Rainmaker. Azure Storage SDK automatically retries multiple times on timeouts, and mistakenly empties the queue.

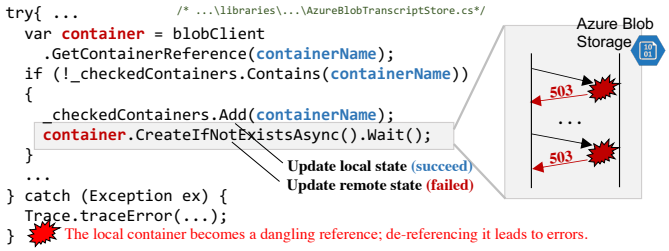


Figure 5: A local-state divergence bug in Microsoft BotBuilder detected by Rainmaker (confirmed and fixed). Azure Storage SDK automatically retries multiple times (for 503).

in this pattern, the retry operation succeeds, and the SDK successfully hides the transient error from the application.

Figure 4 shows a silent semantic violation [71] detected by Rainmaker in Microsoft Orleans [13]. Orleans uses Azure Queue Storage service to manage messages. It implements a method `GetQueueMessage()` to dequeue one message from the queue. As shown in Figure 4, the method calls the SDK API `ReceiveMessagesAsync` to dequeue a message from the remote service. The corresponding HTTP request is non-idempotent and should not be naïvely retried [76], because each retry changes the contents of the queue. However, the SDK API automatically retries the request on a transient fault. If a request successfully dequeues a message but its response times out, the SDK retries the request multiple times, each of which, if successful, can dequeue a message. If the last retry/response succeeds, the API successfully returns the message. The application does not handle this corner case, even though the API documentation mentions it. Such behavior violates the semantic of the `GetQueueMessage()` API which is documented to only "*get a message from the queue*" [62]. In fact, repeated retries can dequeue all the messages from the queue, in which case the SDK API returns `null`; Orleans does not expect such behavior and would dereference the `null` pointer and crash. Note that `ReceiveMessagesAsync` is not the only method that has such behavior. If we replace it with `SendMessageAsync`, the above example can enqueue more messages than expected, which may lead to silent resource leaks on the cloud service. Such silent semantic violations are hard for application developers to fix or even detect, as the retries are done by the SDK and are agnostic to the application.

3.4 State Divergence

In this pattern, a mishandled transient error leads to divergence of the local state (in the application) and the remote state (in the cloud service). There is no API semantic violation as in §3.3, but the state divergence could lead to undesired application behavior, e.g., exceptions and resource leaks.

State divergence can happen when an application *optimistically* updates a local state that is correlated with the success of a cloud API call made after the update. The bug manifests if a transient fault fails the request (so no change on the cloud side), but the application does not restore the optimistic update. The updated state makes the application behave as if the REST API call succeeded, while it actually failed.

Figure 5 shows a state-divergence bug [30] from Microsoft BotBuilder [17] detected by Rainmaker. BotBuilder uses Azure Blob Storage to store blob data which is organized into containers. To create a container, BotBuilder calls REST API `CreateBlobContainer`. When transient errors (e.g., 503 `ServerBusy` errors) occur on the request path of a `CreateBlobContainer` call, BotBuilder swallows the exception in the `catch` block. However, BotBuilder adds the container into its local state of created containers *before* calling `CreateBlobContainer`. As a result, the local state is corrupted with a dangling container pointer, which leads to crashes when BotBuilder dereferences the pointer (e.g., with a list operation). The bug has the same essence as file system bugs that violate update dependencies [39]. On the other hand, transient errors are likely more frequent than file system crashes.

State divergence can also happen when a request changes the remote state, but the application is unaware of the change. Such bugs can manifest when a transient fault breaks the *return path* of a REST call that has changed the remote state. If not handled correctly, the application would assume the call never succeeded, leading to inconsistencies of states.

4 Rainmaker

4.1 Overview

Rainmaker is a "push-button" reliability testing tool for applications that use RESTful cloud services, such as Azure Storage, Azure CosmosDB, and AWS S3. It checks whether the application under test can correctly tolerate or handle common transient errors under the cloud-based fault model (e.g., temporary service unavailability and request timeouts), and detects bugs like the ones described in §3. Its "push-button" nature comes from the automatic fault-injection policies (§4.2) and oracles (§4.3), which are generic and applicable to any application that uses the supported cloud services.

A developer can directly apply Rainmaker as a "plugin-and-play" tool to their existing test suites in their existing testing environments, without writing additional code or configurations. The plugin-and-play nature is achieved by its fault

injection mechanism—Rainmaker injects errors by intercepting outbound REST API calls made by the application to the cloud service at the HTTP layer. It includes a standalone HTTP proxy component to do the interception. For example, to inject a 5XX error on the request path of a REST call, the proxy blocks the request from reaching the service and responds with an HTTP response containing the 5XX error code. To inject a timeout on the response path, the proxy lets the request go to the service; however, on receiving the response, it introduces a delay to force a timeout at the application.

Compared with injecting faults directly into application code (e.g., in the forms of exceptions), intercepting at the HTTP layer brings a number of technical benefits: 1) it allows intercepting *fine-grained* REST calls made by the application, including those triggered by SDK retries. As we discussed in §3, injecting errors into retry requests and responses is crucial for exposing certain categories of bugs; 2) error handling in cloud-backed applications depends on not only exception types but also HTTP status codes; 3) it makes the fault injection mechanism transparent to the application under test and work irrespective of the application language and architecture; 4) HTTP requests/responses are highly interpretable, and errors can be uniformly injected by manipulating HTTP responses, with no need to understand external dependencies (e.g., complex exception objects with multiple fields).

Usage. Rainmaker takes as input an existing testing suite that uses RESTful cloud services such as Azure Storage services or their emulators [68], and a desired coverage metric (§4.2.2). Rainmaker first installs a local HTTP proxy that can intercept and manipulate HTTP traffic to and from cloud services (or their emulators). It then selects and executes a minimal set of tests required to achieve the target coverage. As tests are executed, Rainmaker injects faults into their REST API calls according to automatic fault-injection policies. After each test is executed, Rainmaker's oracles analyze the test outcomes and raise alerts as potential bugs are detected.

We envision Rainmaker to be a standard, widely-used testing utility as a part of cloud service SDKs.

4.2 Fault Injection Policy

A fault injection policy specifies *what* faults to inject and *where* (at which REST API calls) to inject them. Rainmaker's fault injection policies are designed with two main objectives.

First, the policies should be *effective*. This requires them to 1) inject only *valid* faults and 2) cover the myriad bug patterns of the taxonomy (§3). One common policy is *randomized* fault injection (e.g., selecting a random fault at a random REST call). However, randomized injection can hardly be effective. As shown in §3, to expose certain bug patterns needs multiple specific faults injected along the interaction of a REST API call—randomized injection is unlikely to hit the specifics. In fact, randomized injection cannot even guarantee valid faults. For example, returning a 503 (`ServiceUnavailable`) error after a write request is successfully executed is invalid, because this is inconsistent with the cloud service contract.[1]

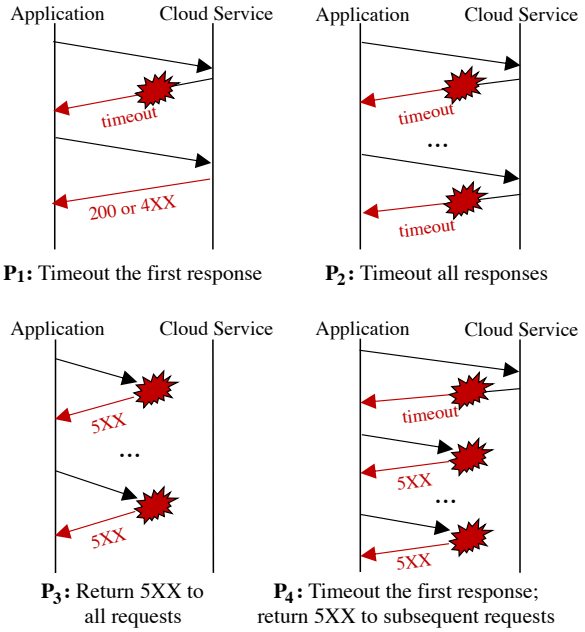


Figure 6: Fault injection policies of Rainmaker that cover all the bug patterns in our taxonomy (see Table 2). Arrows represent HTTP requests and responses for a *single* REST call (and retries). 5XX represents an error code for transient service-side failure. For a REST call with no retry, the four policies are reduced to two.

Second, the policies should enable *efficient* testing—achieving high testing coverage with a small number of test runs. Exhaustively injecting all possible faults at every REST call could be prohibitively expensive, because one test could issue thousands of REST API calls (see §5.3), and many different faults are possible for each call. This is further aggravated by the fact that each fault injection may require a separate test run because injecting the first fault might disrupt a test's subsequent execution.

We next discuss how Rainmaker achieves the two objectives in §4.2.1 and §4.2.2, respectively. Note that Rainmaker can be easily extended to support new policies.

4.2.1 What faults to inject (for a REST API call)?

Rainmaker injects transient faults that occur during the interaction of one REST API call, following the taxonomy in §3. However, the fault space is large even for a single REST call. This is because each of the large number of possible faults may occur on the request or the response path of the original request or subsequent retires issued by the SDK. Interestingly, we find that a small set of four policies (Figure 6) are sufficient to cover the taxonomized bug patterns, as shown in Table 2. For REST calls that do not retry, the four policies are

[1] In practice, a cloud service backend can have bugs to return such an inconsistent response [23]. However, we do not consider buggy cloud services.

Bug Pattern	Fault Injection Policies
No error handling	P_1, P_2, P_3, P_4
Throwing unrelated exception	P_1
Silent semantic violation	P_1, P_2
State divergence	P_1, P_2, P_3, P_4

Table 2: The mapping from the bug patterns and the error injection policies that can potentially expose each bug pattern.

reduced to two: 1) return a transient error code to the request, and 2) timeout the response. The four policies are:

- **P_1** (Timeout the first response). This policy forces a retry that can expose bugs related to invalidated preconditions. Since the timeout is at the response path after the request takes effect at the cloud service, the retry could trigger bugs of throwing unrelated exceptions, silent semantic violation, and/or state divergence, depending on the REST API semantics and the handling logic.
- **P_2** (Timeout all responses). This policy presents an entirely timed-out REST call to the application, while the REST call has taken effects (potentially multiple times) at the cloud service. It could trigger bugs of silent semantic violation and state divergence.
- **P_3** (Return transient error codes to all requests). Under this policy, the REST call does not reach the cloud service as all requests are returned with transient service-side errors. The policy can potentially expose bugs of state divergence, if the local state is optimistically changed before the REST call, but not restored after the call fails.
- **P_4** (Timeout the first response and return transient error codes to all subsequent requests). This policy presents a failed REST call (with a transient error code in response) to the application, which on the contrary has been successfully executed exactly once at the cloud service. It could trigger bugs of state divergence.

Rainmaker by default injects 503 (`ServiceUnavailable`), as the transient fault(s) for request failures. For the responses, Rainmaker injects a timeout fault. These two types of transient faults are common and safe to inject and thus are valid faults on the request and response path, respectively. In comparison, error codes like 500 (`InternalServerError`) have undefined semantics and service-side behavior. The error code can be further customized for specific cloud services and their SDKs based on their definitions of transient faults and retry policies.

Rainmaker identifies all the retried requests and their responses of each REST API call by checking the unique request ID in the HTTP header (e.g., `x-ms-client-request-id` for Azure Storage services). The request ID is provided by the SDK to identify the specific REST call request and is shared by all the subsequent retried requests and responses.

One design choice we make is to avoid encoding specifications of REST/SDK APIs in policies (e.g., idempotency of a REST API and retry behavior of an SDK API). Leveraging API information can help optimize test efficiency. However, it is known that specifications are expensive to maintain and are often incomplete and outdated in practice. Rainmaker minimizes its assumption on the REST/SDK APIs.

	Coverage Metric
C_1	Cover all tests; for each test, select the first REST call.
C_2	Cover all unique call sites of the application code; if a call site is exercised by multiple tests, select the cheapest test.
C_3	Cover all tests and all unique call sites in a pairwise manner.
C_4	Cover all unique call sites of every test; if multiple REST calls exercises the same call site, select the first call to inject.

Table 3: The coverage metrics supported by Rainmaker. We use C_4 as the default metric. Note that injecting faults in every REST call in every test is prohibitive (see §5.3).

4.2.2 Which REST API calls to inject faults?

A test suite may generate an excessive number of REST calls; injecting faults into all of them can be prohibitively expensive, even with the above optimized policies (it could take several machine-months for one application, §5.3). In fact, many REST calls can be redundant (e.g., invoked by the same application code location) for the purpose of covering new error-handling code. Rainmaker therefore selectively injects faults into a small number of REST calls, which achieves certain coverage metrics and optimizes testing resources.

Coverage metrics. Rainmaker supports four different coverage metrics (Table 3). While C_1 measures coverage in terms of the REST calls, C_2–C_4 involves call sites of cloud service APIs. We use the term *call site* to denote a location in the application code that invokes an SDK API that eventually makes one or more REST calls (typically, one SDK API invokes one REST API [26, 27]). Call sites reside in the application code, while REST calls are constructed by the SDK. Hence, C_2–C_4 are more intuitive to developers than C_1.

However, computing C_2–C_4 is challenging for Rainmaker and for any other tools that inject faults via a separate HTTP proxy [37, 47]. This is because the proxy process does not have visibility of call sites within the test/application process.

Making call sites available to the HTTP proxy. Rainmaker addresses this challenge with two techniques using automated instrumentation. First, Rainmaker enables a test to communicate with the HTTP proxy through headers of outgoing HTTP messages. Given test binaries, Rainmaker automatically instruments their outbound HTTP calls. An instrumented call can put its call site information in the header of an outbound HTTP request, so the HTTP proxy can retrieve the information. This can be done automatically since outbound HTTP calls are usually made with a small number of standard core APIs. For example, one needs to instrument only four HTTP client API families (e.g., `HttpClient.SendAsync`) provided by .NET core libraries to intercept outgoing HTTP calls from *all* Azure SDKs (see §4.5).

Second, an outbound HTTP call needs to automatically find its call site to put in the HTTP header. If the application were single-threaded, the instrumented HTTP call could identify the call site by taking the caller of the bottom-most SDK function in the call stack. But, modern applications are multi-threaded, and an HTTP call usually happens asynchronously in a child thread created as a result of executing the call site. In this case, a call stack taken at an outbound HTTP call does not capture the call site that resides in a different thread.

To solve this problem, Rainmaker utilizes inheritable thread-local storage (ITLS) supported by modern languages such as .NET [64] and Java [49]. Any data stored in the current thread's ITLS automatically propagates to all its child threads. Rainmaker automatically instruments all call sites (identified by SDK namespace) so that at runtime, they store their location information in the current thread's ITLS. When a call site eventually invokes an instrumented, outgoing HTTP call, in the same thread or in a child thread, the call retrieves the call site information from its ITLS and puts it in the HTTP header for the proxy process to examine.

Test planning. With the call site information available at the HTTP proxy, Rainmaker can inject faults according to the specified coverage metric in Table 3. Each coverage metric is a tradeoff between completeness and cost.

To plan the fault injection runs, Rainmaker first performs a *reference run* of the test suite with no fault injection. During the reference run, Rainmaker measures the time taken by each test and observes, by using its HTTP proxy, the REST calls made by different tests. It then selects the tests that issue REST calls as candidate tests for fault injection. Rainmaker then performs an offline analysis to generate test plans containing the minimum number of fault-injection target REST calls (and their tests) in order to achieve target coverages. It also outputs an approximate running time of each plan using the time of the tests in the reference run and, if any, the delays to be injected to create timeout errors. The time helps a developer choose the right coverage metric by understanding the tradeoff between completeness and cost.

Given the time taken by each test and the set of REST calls each test makes in the reference run, it is straightforward to implement the policies C_1, C_2, and C_4. For C_3, Rainmaker models it as a linear programming (LP) problem of generating a set of pairs that cover all N tests and M unique call sites (with each test covering a subset of call sites), while minimizing the total test running time (each test has different running time). It then uses an LP solver to generate a plan.

Note that test planning, including the reference run and LP solving, is done offline as a one-time effort. The results can potentially be reused across test runs in CI/CD environments.

4.3 Test Oracles

A test oracle checks whether the outcome of a fault injection test run indicates a bug. A trustworthy oracle catches different types of bugs with no false alarms so that developers can focus their investigation only on true bugs.

```
/// test code
public async Task
Should_be_able_to_send_if_container_was_not_found()
{ ...
  await plugin.BeforeMessageSend(message); ...
} /* ServiceBus.AttachmentPlugin.Tests/When_sending_message_using_connection_string.cs */

/// application code
public override async Task<Message>
BeforeMessageSend(Message message)
{ ...                                                    503
  try {
    await container.CreateIfNotExistsAsync();
  }                                          Azure.Storage.StorageException:
  catch (StorageException ex) {              Service unavailable (503)
    // intentionally swallow and continue
  }
  ...
  await blob.UploadFromByteArrayAsync(...);
  ...                                        Azure.Storage.StorageException:
} /* ServiceBus.AttachmentPlugin/AzureStorageAttachment.cs */  Container does not exist (404)
```

Figure 7: An exception captured by Rainmaker to detect a state-divergence bug in ServiceBus AttachmentPlugin. The exception that fails the test (404) is inconsistent with the injected fault (503).

With the goal of being generic and widely applicable to any cloud-backed application, Rainmaker does not use any application-specific oracle. Instead, it devises a set of application-agnostic oracles on top of the existing test oracles encoded in developers' test code. These oracles are effective in identifying various types of bugs, with low false positives.

4.3.1 Exception Oracle

This oracle flags a fault-injection test outcome as a potential bug if 1) the test fails with an exception, 2) the exception is created in application code rather than in test code, and 3) the exception is inconsistent with the injected fault. We now explain the rationale behind the three conditions.

When a test fails with an exception as a result of an injected fault, it may not always mean a bug. For example, in the applications we use for our evaluation, many tests directly interact with a cloud service (e.g., to setup the test environment) but without proper error handling. If Rainmaker injects faults into such REST calls, the test will fail with an exception. However, the failure does not indicate application bugs. Rainmaker applies the second condition to filter out test failures due to exceptions created in test code, based on the exception call stack. Note that Rainmaker avoids injecting faults into REST calls with call sites from the test code.

However, not all test failures due to exceptions created in application code are bugs. For example, a utility method of an application may intentionally propagate an exception to the upper layer and expect it to be handled there. When a test for this utility fails due to not handling the exception, the test failure is expected, and it does not indicate a bug.

Rainmaker applies the third condition to only report bugs when the final exception that causes the test failure is inconsistent with the injected fault (by searching the injected HTTP

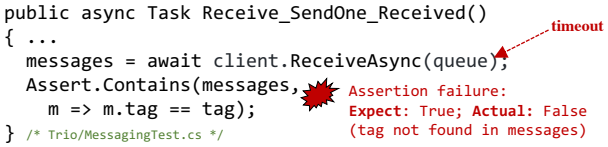


Figure 8: An assertion utilized by Rainmaker to detect a bug of silent semantic violation in Storage.NET backed by AWS SQS. When timeouts are injected, `ReceiveAsync` dequeues multiple times, causing an empty message list returned, which fails the assertion.

status code or error code in the exception stack). The intuition is that, if the test fails due to an error different from the injected fault, it indicates that the fault was once handled (it shows the developer's intention to handle it), but the handling is inappropriate (at least insufficient) and causes a different error that fails the test. This oracle can capture bugs of throwing unrelated exceptions as well as silent bugs of semantic violations and state divergence which do not cause immediate exceptions but result in exceptions eventually. Figure 7 shows such an example, where a 503 fault injected to `CreateIfNotExistsAsync` leads to a 404 exception from `UploadFromByteArrayAsync` and fails the test.

Note that the oracle is incomplete. If an application misses error handling (§3.1), exceptions exposed by the test could be a bug. However, at the unit test level, it is indeterminate.

Relaxing the oracle. With all three conditions, the oracle above is conservative. One can relax it to identify other types of likely bugs. If the developer is testing an application or running a system test (instead of a unit test), she can disable the third condition so that Rainmaker flags any failure (e.g., crash) due to exceptions from appliation code as a bug. This is because Rainmaker only injects transient faults that are expected to be handled gracefully.

4.3.2 Assertion Violation Oracle

This oracle flags a fault-injection test outcome as a potential bug if the test fails due to an assertion violation (*and* not an unhandled exception). Intuitively, transient faults injected by Rainmaker should not impair semantic correctness of application code; hence existing assertions should not be violated if the faults are properly handled. The assertions in test code could be brittle to fault injection [45], i.e., an assertion violation is not a bug, but caused by the fault injection changing application runtime behavior. In practice, we find such brittle assertions are small in numbers, as discussed in §5.2. The assertion oracle can capture bugs of silent semantic violations and state divergence. Figure 8 shows how Rainmaker leverages the existing assertion to capture a silent semantic violation in Storage.NET [19].

4.4 Diagnosis Support

Associating source-code information with faults. Diagnosing and localizing bugs in application code triggered by HTTP-level faults can be challenging without source-code context. This can be true even when the developer knows the fault-injected REST API or SDK API (via instrumentation), because they can be invoked by multiple program locations.

To help developers debug the test failures, Rainmaker associates the fault injection with source-code information in the form of the call site, together with the call stack of runtime exceptions or assertions (§4.3). We find the REST API call site and exception/assertion call stack are critical to debugging. They help developers to understand what and where fault(s) were injected and reason about the error propagation inside application code. One can further apply existing techniques to automatically reconstruct the failure execution (e.g., [85]).

Reproducing bugs. Rainmaker can reproduce a reported bug because all fault injections for a test are determined by the test planner (§4.2.2). If an injected fault exposes a bug, Rainmaker can rerun the planned fault injection to reproduce the triggered bug. If the test is nondeterministic, it may take several runs to reproduce the bug. In our experience, the error handling behavior for REST calls is typically local to the call and is rarely affected by nondeterminism of test execution.

4.5 Implementation

We have implemented Rainmaker for Windows. Its HTTP interception is implemented using MockServer [18], with fault-injection policies implemented as MockServer rules in Java. Rainmaker registers a system proxy for Windows Internet Services to forward all HTTP traffic to the MockServer proxy. This enables Rainmaker to inject faults to *any* application that issues REST APIs. The oracles are implemented in Python, which analyzes the raw test results and logs.

Rainmaker currently supports coverage metrics C_2–C_4 for .NET applications only, which needs dynamic instrumentation to record and propagate call site information (§4.2.2). The instrumentation is implemented using the .NET profiling API [65] that enables changing bytecode of a method before it is JITed. Rainmaker inserts call site information in HTTP headers by instrumenting four HTTP API families from the .NET core library that cover all outgoing HTTP messages. We believe the same mechanism can be implemented for Java.

To support a new cloud service in Rainmaker only takes two inputs in the form of configurations: 1) the SDK namespace (e.g., `Azure.Storage*` for Azure Storage SDK) and 2) the request-ID tag of the cloud service (e.g., `x-ms-client-request-id` for Azure Storage services and `amz-sdk-invocation-id` for AWS S3). The former instructs Rainmaker what call sites to record, and the latter identifies a request and its retries (they all have the same request ID).

5 Evaluation

Our evaluation addresses the following questions: 1) Can Rainmaker find new bugs in real-world cloud-backed applications? 2) Are Rainmaker's testing results trustworthy? 3)

Application	Cloud Service	# Stars	# LOC	Selected Tests
Alpakka	Queue	106	14K	9
AttachmentPlugin	Blob	67	1.3K	32
BotBuilder	Blob, Queue	758	18K	67
DistributedLock	Blob	838	17.1K	30
EF Core	CosmosDB	11.7K	842.4K	420
FHIR Server	CosmosDB	897	112.8K	202
Insights	Blob, Queue, Table	20	51.7K	147
IronPigeon	Blob	255	5.4K	7
Orleans	Blob, Queue, Table	8.8K	187K	155
Sleet	S3	276	18.7K	2
Storage.NET	Blob, Queue, S3, SQS	567	12.4K	36

Table 4: The cloud-backed applications used in our evaluation.

What is the tradeoff between running time and coverage?

- §5.1: Rainmaker finds 73 new bugs in all 11 evaluated cloud-backed applications, which represent a swathe of reliability issues. So far, 55 of them have been confirmed, and 51 have been fixed by the developers.
- §5.2: Rainmaker's test oracles have a low false-positive rate (1.96%) with regards to test failures.
- §5.3: Rainmaker significantly reduces running time compared to exhaustive fault injection with coverage guarantee.

Evaluation setup. We evaluated Rainmaker on 11 popular .NET applications that use six different cloud services from two cloud service providers, Azure and AWS (Table 4). These applications are mature and widely used; many are maintained by software companies, such as Orleans, BotBuilder, EF Core, FHIR Server from Microsoft, Insights from NuGet, and Alpakka from Petabridge. They use six cloud services: Blob Storage [7], Queue Storage [11], Table Storage [12], and CosmosDB [9] from Azure, and Simple Storage Service (S3) [1] and Simple Queue Service (SQS) [4] from AWS. We configure Rainmaker to support these services (§4.5).

We apply Rainmaker to existing test suites of the applications. Rainmaker automatically selects tests that interact with the cloud services from the test suite by monitoring HTTP traffic during the reference run (§4.2.2). The number of selected tests varies from 2 to 420 across the applications (Table 4), including both unit and system tests. We differentiate unit and system tests based on their naming conventions.

All the tests that interact with Azure cloud services are run with emulators: Azurite [68] for Blob, Queue, and Table and the CosmosDB emulator [63] for CosmosDB. The tests that interact with AWS are run with the real S3/SQS services; we did not find an official AWS emulator.

5.1 Finding New Bugs

Rainmaker finds a total of 73 *new* bugs in the evaluated applications (Table 5). Those bugs include all the bug patterns in the taxonomy (§3): 29 bugs of no error handling, 23 bugs of throwing unrelated exceptions, four bugs of silent semantic violations, and 17 bugs of state divergence. Rainmaker finds bugs in *every* application, showing the error-proneness of handling transient faults with cloud-based programming. We have reported 66 (out of 73) bugs. So far, 55 of them have been confirmed, and 51 of them have been fixed.

Application	No Error Handling	Throw New Exception	Semantic Violation	State Divergence	Total
Alpakka	0	0	1	1	2
AttachmentPlugin	0	0	0	2	2
BotBuilder	0	2	0	2	4
DistributedLock	0	2	0	0	2
EF Core	7	0	0	0	7
FHIR Server	11	0	0	0	11
Insights	0	10	0	0	10
IronPigeon	0	1	0	0	1
Orleans	0	5	2	11	18
Sleet	0	2	0	0	2
Storage.NET	11	1	1	1	14
Total	29	23	4	17	73

Table 5: New bugs detected by Rainmaker across the applications.

Many of the detected bugs have severe consequences, such as unexpected application termination, data loss/inaccessibility, and resource leaks (Table 6). All these consequences are triggered by transient faults against *one* REST API call.

Rainmaker detects bugs that are unlikely to be exposed by randomized fault injection. For example, a bug [36] in DistributedLock is only triggered when timeout happens to the response of a specific SDK API call site. The test used for detecting this bug issues 900+ requests in total; only four requests are from that specific call site. Rainmaker can consistently detect this bug as it systematically exercises the call sites of the REST API calls (§4.2.2).

Table 7(a) shows that all four fault injection policies (Figure 6) employed by Rainmaker are effective in finding bugs. The four policies address different fault scenarios and are complementary to each other. No policy detects all the bugs. Similarly, Table 7(b) shows that oracles are complementary to each other. For example, all the semantic violation bugs are captured by assertions as they do not cause exceptions.

The 73 bugs cause 2,654 test failures in total. To inspect them, we cluster test failures based on (a) the application call site that invokes the REST API where fault(s) are injected and (b) the exception stack trace in the application namespace, or assertion. For two test failures with both (a) and (b) being the same, they are considered to have the same root cause, i.e., injecting to the same API call site causes the same exception stack trace in application namespace or assertion violation.

No error handling. Rainmaker found 29 bugs of this type, where neither the SDK nor the application handles the injected faults. These bugs are all from applications that use Azure Storage SDK with versions before 12.3.0 and the CosmosDB SDK; the former does not retry timeouts, and the latter does not retry with our single-region setting. The applications using these SDKs are expected to handle transient errors but do not have error handling. These bugs are manifested in system tests when Rainmaker injects faults into the REST calls.

Consequence	Example	# Bugs
Externalizing unrelated exception	IronPigeon-133: Deletion operations perform on non-existent resources [46].	36
Operation failure	AttachmentPlugin-277: Containers cannot be created due to transient error [24].	35
Incorrect results or states	BotBuilder-5787: The timestamp metadata of blobs is not set correctly [31].	13
Application crash	FHIR Server-2732: FHIR Server can crash unexpectedly upon transient faults [38].	11
Data loss or inaccessibility	BotBuilder-6407: Activities that should be logged are lost [32].	4
Resource leak	Orleans-7790: Redundant messages were added incorrectly to the Queue service [72].	2

Table 6: Consequences of the bugs found by Rainmaker. One bug can lead to multiple consequences.

Throwing unrelated exceptions. Rainmaker found 23 bugs of this type. Rainmaker triggers these bugs by injecting timeout to the response path after a request takes effect at the cloud service (see Figure 3). While the bug in Figure 3 is captured by an assertion on the number of created blobs, many other bugs are captured by the exception oracle (the exception is mostly inconsistent with the injected fault). Such bugs can be avoided if the cloud service can collapse the retries: If the first attempt is successful, the cloud service should ignore the following retries of the same SDK API call. This requires the cloud service to identify the retries of each SDK API call, which can be specified by the SDK when it issues a retry.

Silent semantic violations. Rainmaker found four bugs of this type. All of them are caused by retrying non-idempotent REST APIs (e.g., `ReceiveMessagesAsync` in Figure 4). Rainmaker detects these bugs by injecting timeout to trigger non-idempotent retries and leveraging assertions to catch semantic violations (as in Figure 8). Different from traditional system services (e.g., file systems), cloud services seldom have standard API specifications like POSIX for file system APIs; documents are often outdated or incomplete. Without precisely understanding the semantic and side effect of each REST API, it is difficult for developers to avoid silent semantic violations.

State divergence. Rainmaker found 17 bugs of this type. Some applications maintain local data structures to reflect the state of the remote resources hosted by the cloud service. Rainmaker triggers state divergence by injecting 5XX error codes to the request path or timeout to the response path (e.g., Figure 5). Although the inconsistencies do not immediately lead to exceptions, Rainmaker can still catch them when the test throws exceptions (e.g., Figure 7) or fails assertions.

5.2 False Positives

While identifying bugs with its test oracles, Rainmaker introduces a very low false positive rate of 1.96% (52/2,654). It reports in total 2,654 test failures for the evaluated applications. Among the failures reported, only 52 of them were false alarms. The low false positive rate is attributed to Rainmaker's exception oracles (see §4.3.1). If Rainmaker directly reports exceptions thrown by unit tests, it would have reported 3.07 times more test failures. To validate that the oracles filter out little true alarms, we randomly sample a hundred exceptions that are filtered out by Rainmaker's exception oracle and find that none of them indicates a bug.

Application	Fault Injection Policy				Test Oracle		
	P_1	P_2	P_3	P_4	Exp (Unit)	Exp (Sys)	Assert.
Alpakka	2	1	1	0	0	0	2
AttachmentPlugin	0	1	2	1	2	0	0
BotBuilder	2	1	2	1	3	0	1
DistributedLock	2	0	0	0	2	0	0
EF Core	7	n/a	7	n/a	0	7	0
FHIR Server	11	n/a	11	n/a	0	11	0
Insights	10	0	0	0	10	0	0
IronPigeon	1	0	0	0	1	0	0
Orleans	17	11	11	11	16	0	2
Sleet	2	0	0	0	2	0	0
Storage.NET	13	12	12	12	2	11	1
Total	67	26	46	25	38	29	6

Table 7: The breakdown of the number of bugs captured by the fault injection policies (left) and oracles (right). For EF Core and FHIR Server, the four policies are reduced to two, because CosmosDB SDK does not retry in our setting (single region). For the exception oracle ("Exp"), we differentiate unit tests and system tests. Note that one bug can be exposed by multiple fault injection policies.

Among the 52 false alarms, 10 of them come from five tests of Insights. These tests exercise scenarios where a client issues invalid requests and expects the return of certain error codes (e.g., 404 `Not Found`). Since Rainmaker injects 5XX on the request path (P_3 and P_4 in Figure 6), the REST call fails by assertions on the original error codes. To avoid those false alarms needs to understand those REST calls, e.g., based on information from the reference run. Note that Rainmaker does not inject faults on an HTTP response that already has an error code.

The other 42 false alarms were caused by 14 tests from Alpakka and IronPigeon. Those tests use a small connection timeout that was exceeded due to the fault injection, resulting in either assertion failures or inconsistent exceptions. These tests are considered flaky tests in software testing literature [54]. Strictly speaking, flaky tests should not be counted as false alarms. However, detecting flaky tests is not a goal of Rainmaker, and the flakiness is indeed triggered by fault injection (it can also be triggered by slow connections).

5.3 Running Time with Coverage

Rainmaker takes 0.57–212.77 hours to test each application under coverage metric, C_4 (Table 3), as shown in Table 8. All the experiments were run on a Windows 10 Pro with AMD Ryzen 9 5900X 3.70 GHz CPU and 32 GB memory. Over 93.54% of the running time is spent on executing fault injection test runs. Rainmaker also spends 0.02–16.61 hours to 1) conduct the vanilla run and 2) generate the test plan. The test plan generation using a linear programming (LP) solver takes only 1.46–3.67 seconds across the applications and is negligible compared with the time for vanilla test runs. The numbers of constraints and variables range from 14 to 241 and

Application	Running Time (Machine Hours)			# Fault Injection
	Test Planning	Fault Injection	Total	Test Runs
Alpakka	0.02	0.97	0.99	196
AttachmentPlugin	0.04	2.83	2.87	416
BotBuilder	0.32	10.36	10.68	888
DistributedLock	0.13	4.97	5.10	572
EF Core	10.65	159.60	170.25	4786
FHIR Server	16.61	196.16	212.77	6428
Insights	0.36	10.97	11.33	3056
IronPigeon	0.06	0.51	0.57	120
Orleans	2.57	53.17	55.74	3804
Sleet	0.11	0.63	0.74	72
Storage.NET	2.30	40.27	42.57	1512

Table 8: Running time of Rainmaker in machine hours, for the most expensive coverage metric C_4. Test planning includes both the vanilla test run and test plan generation (the latter takes seconds).

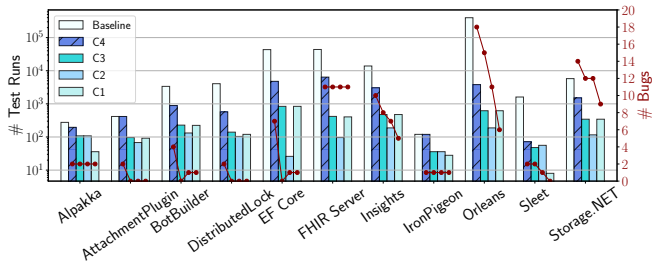


Figure 9: The number of fault injection test runs of each coverage metric, C_1–C_4 in Table 3 (bars) and the number of bugs detected by each coverage metric (dots). Baseline refers to exhaustively injecting faults to all the REST API calls of all the tests.

from 18 to 3214, respectively. Note that test plan generation is a one-time effort and is done offline; fault injection takes multiple test runs and is more time-consuming.

Rainmaker's test planning avoids exhaustively injecting faults in every REST call (the baseline). Figure 9 shows the number of fault injection test runs each coverage metric can reduce, compared to the baseline. Overall, Rainmaker with C_4 (default coverage) reduces on average 64.47% of test runs for each application compared to the baseline. In particular, C_4 reduces 394,172 (99.04%) test runs for Orleans alone because Orleans has stress tests that repeatedly exercise the same SDK API call site in large loops. C_4 reduces fault injection runs by "deduplicating" REST calls with the same call site. Without the reduction, Rainmaker would take 588 days to test Orleans.

In terms of bug finding effectiveness, C_4 covers all the tests and REST calls that are covered by C_1–C_3. Thus, C_4 detects all the bugs that are detected by C_1–C_3.

6 Discussion and Limitation

Rainmaker's effectiveness depends on the adequacy of the existing test suite, in particular, tests that interact with cloud services. Such tests may not be abundant (Table 4). For example, in Microsoft Orleans, only 155 out of 7,002 tests interact with cloud services. A more common testing practice is to mock REST APIs [53]. Our future work is to auto-rewrite mocked tests into tests that can be utilized by Rainmaker.

Rainmaker is not cheap. It may need to run a test multiple times, each injecting different fault(s) or to a different REST call. For big test suites, Rainmaker would need significant machine hours (Table 8). In fact, our evaluation shows that many bugs trigger multiple test failures (§5.1). A future work is to reduce the cost using test selection techniques [83].

Rainmaker currently only targets faults that occur in one interaction of REST API call initiated by the application and the subsequent retries (§3). We are investigating how to inject faults to multiple *correlated* API call interactions which has a larger fault space and a more complex fault model.

Rainmaker can be extended to test applications under potential cloud service bugs. In our evaluation, we find that cloud services have various correctness issues. For example, we observe that the same REST API of AWS S3 has different consistency guarantees at different regions [25], which can break application assumptions. Also, the error behavior is often opaque and hard to reason, e.g., the side effect of a non-idempotent API call when it returns 500 (`InternalServerError`).

Not every bug found by Rainmaker is easy to fix. For example, timeouts on the response path make it hard for SDK/application to know whether the failed request has taken effect at the cloud service. Server-side support such as versioning (e.g., based on HTTP ETag) and transaction-like API support could potentially help non-idempotent APIs. Moreover, SDKs should not blindly retry non-idempotent APIs, which however is not an uncommon practice, as shown in Table 1.

Our current prototype of Rainmaker focuses on .NET applications. We believe that the prototype can be generally extended to support applications in other languages as well. The high-level idea of injecting faults with an HTTP proxy is independent of the programming language of the target application. The only .NET-specific component in our prototype is the one that computes the coverage metrics C_2, C_3, and C_4 (in Table 3) by using .NET profiling API [65] for dynamic instrumentation and .NET inheritable thread-local storage (ITLS) [64] for propagating call-site information. Neither dynamic instrumentation nor ITLS is unique to .NET; they are already available or can be supported in other languages and runtimes such as Java.

7 Related Work

Our work focuses on push-button tooling to help developers address emerging reliability challenges of cloud-backed applications. The techniques that embody Rainmaker have lineages of error-handling analysis and fault injection.

- **Error-handling code analysis.** It is known that error handling is a main root cause of production failures of software systems [40, 41, 84]. Prior work developed static analysis for error-handling code to search missing logs and `TODO`s in error-handling code [84, 86], check error specifications [48], and understand error propagation [41, 77, 82].

- **Fault injection.** Prior work developed fault injection techniques for traditional software applications [29, 35, 59, 87] and for distributed backend systems (e.g., storage and data processing systems) that empower modern cloud services [16, 20, 21, 33, 34, 40, 43, 51, 55–57, 69, 73, 80]. They implicitly or explicitly target error-handling code. Moreover, many existing fault injection tools support injection at the HTTP layer [37, 47].

Unfortunately, we realized that none of the above techniques support a push-button tool that is generic, widely applicable to cloud-backed applications because they all have one or more of the following limitations. First, many fault injection tools only provide mechanisms without fully automatic policies beyond randomization or oracles beyond crashes [6, 21, 35, 40, 43, 44, 51, 58, 61, 73, 75]. Developers need to manually implement them, which is nontrivial. Second, many techniques use application or domain knowledge to devise policies and oracles [16, 20, 33, 34, 55, 56, 69, 80]. For example, fault injection for distributed databases checks consistency and isolation properties by injecting node crashes and network partitions [16, 52]. These techniques cannot be used as a common, basic utility for diverse types of applications. Third, fault injection at the program level [35, 50, 59, 60, 87] is fundamentally limited to cloud-backed applications: 1) program errors (exceptions and errno) are too coarse-grained to expose certain bug patterns (e.g., silent semantic violations) which need fine-grained injection at the HTTP layer; 2) it is nontrivial to construct exception objects—error handling of cloud-backed applications is based on not only exception types, but also HTTP status codes (Table 1); 3) few program fault injection considers cloud states, but assumes a single program. Rainmaker instead injects faults at the REST interface, effectively addressing the above three limitations.

The early form of cloud-backed applications is mobile apps that interact with cloud backends via REST APIs. Unlike today's cloud services, the cloud backend is specific to an app and is not widely used as a building block of generic application development. Most fault injection tools for mobile apps focus on GUI testing [74, 79]; few considers app-cloud interactions. Rainmaker applies to cloud-backed mobile apps.

8 Concluding Remarks

Rainmaker serves as a first step tooling to help developers test application reliability under the cloud-based fault model conveniently, when writing cloud-backed code. Despite being a simple tool, Rainmaker can effectively detect bugs in many existing cloud-backed applications, indicating the challenge and error-proneness of correctly handling transient errors. Our goal is to make Rainmaker a basic utility running against every cloud-backed application to detect critical bugs at development time. We hope to inspire more advanced, specialized tooling and raise discussions on cloud service support and SDK designs to eliminate reliability threats in the first place.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Raja Sambasivan, for their insightful comments. We thank Shan Lu, Darko Marinov, Lalith Suresh, and Jun Zeng for valuable feedback and discussions that helped improve this work. We thank Yifei Song for his help on the evaluation and Shuai Wang and Jinghao Jia for helping with the machine setup. We thank all the cloud-backed application developers who engaged with us and reviewed our reports and patches. This work was funded in part by NSF CNS-2130560, SHF-1816615, CNS-2145295, and Microsoft Azure credits.

References

[1] Amazon S3. https://aws.amazon.com/s3/, 2022.

[2] Amazon Simple Queue Service Common Errors. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/CommonErrors.html, 2022.

[3] Amazon Simple Storage Service Error Responses. https://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses.html, 2022.

[4] Amazon SQS. https://aws.amazon.com/sqs/, 2022.

[5] AWS Cloud Products. https://aws.amazon.com/products, 2022.

[6] AWS Fault Injection Simulator. https://aws.amazon.com/fis/, 2022.

[7] Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/, 2022.

[8] Azure Blob Storage error codes. https://learn.microsoft.com/en-us/rest/api/storageservices/blob-service-error-codes, 2022.

[9] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/, 2022.

[10] Azure products. https://azure.microsoft.com/en-us/products, 2022.

[11] Azure Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/, 2022.

[12] Azure Table Storage. https://azure.microsoft.com/en-us/services/storage/tables/, 2022.

[13] dotnet/orleans. https://github.com/dotnet/orleans, 2022.

[14] Google Cloud products. https://cloud.google.com/products, 2022.

[15] HTTP Status Codes for Azure Cosmos DB. https://docs.microsoft.com/en-us/rest/api/cosmos-db/http-status-codes-for-cosmosdb, 2022.

[16] Jepsen. https://jepsen.io/, 2022.

[17] microsoft/botbuilder-dotnet. https://github.com/microsoft/botbuilder-dotnet, 2022.

[18] MockServer. https://www.mock-server.com/, 2022.

[19] Storage.NET. https://github.com/aloneguid/storage, 2022.

[20] ALAGAPPAN, R., GANESAN, A., PATEL, Y., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[21] ALQURAAN, A., TAKRURI, H., ALFATAFTA, M., AND AL-KISWANY, S. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[22] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., LOO, B. T., AND OUTHRED, G. 007: Democratically Finding the Cause of Packet Drops. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (Apr. 2018).

[23] ATLIDAKIS, V., GODEFROID, P., AND POLISHCHUK, M. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'19)* (May 2019).

[24] ATTACHMENTPLUGIN-277. Transient error happening in container existence check will lead to container not found exception. https://github.com/SeanFeldman/ServiceBus.AttachmentPlugin/issues/277, 2022.

[25] AWS-SDK-NET-2084. The retry request of PutBucket will behave differently between regions. https://github.com/aws/aws-sdk-net/discussions/2084, 2022.

[26] AWS-SDK-NET-2102. 1-to-N Mappings between SDK API and REST APIs. https://github.com/aws/aws-sdk-net/discussions/2102, 2022.

[27] AZURE/AZURE-SDK-FOR-NET-31001. 1-to-N Mappings between SDK API and REST APIs. https://github.com/Azure/azure-sdk-for-net/issues/31001, 2022.

[28] AZURE/AZURE-SDK-FOR-NET-9670. Retry on 408, 500, 502, 504 status codes. https://github.com/Azure/azure-sdk-for-net/pull/9670, 2022.

[29] BANABIC, R., AND CANDEA, G. Fast Black-Box Testing of System Recovery Code. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys'12)* (Apr. 2012).

[30] BOTBUILDER-DOTNET-5778. _checkedContainers can become inconsistent with blob storage and further lead to unhandled exception. https://github.com/microsoft/botbuilder-dotnet/issues/5778, 2021.

[31] BOTBUILDER-DOTNET-5787. Metadata of blob can be missing due to transient error which leads to unhandled exception. https://github.com/microsoft/botbuilder-dotnet/issues/5787, 2022.

[32] BOTBUILDER-DOTNET-6407. Activities that should be logged are missing due to transient network errors. https://github.com/microsoft/botbuilder-dotnet/issues/6407, 2022.

[33] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)* (Apr. 2012).

[34] CHEN, H., DOU, W., WANG, D., AND QIN, F. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)* (Sept. 2020).

[35] CHRISTAKIS, M., EMMISBERGER, P., GODEFROID, P., AND MÜLLER, P. A General Framework for Dynamic Stub Injection. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)* (May 2017).

[36] DISTRIBUTEDLOCK-132. Transient error leads to unhandled 409 when releasing an Azure lease. https://github.com/madelson/DistributedLock/issues/132, 2022.

[37] ENVOY DOCS. Envoy Fault Injection. https://www.envoyproxy.io/docs/envoy/latest/api-v3/extensions/filters/http/fault/v3/fault.proto, 2022.

[38] FHIR SERVER-2732. Retry other HTTP error codes from Cosmos DB? https://github.com/microsoft/fhir-server/issues/2732, 2022.

[39] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS) 18*, 2 (May 2000), 127–153.

[40] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. Fate and Destini: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).

[41] GUNAWI, H. S., RUBIO-GONZÁLEZ, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIBLIT, B. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)* (Feb. 2008).

[42] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)* (Aug. 2015).

[43] HEORHIADI, V., RAJAGOPALAN, S., JAMJOOM, H., REITER, M. K., AND SEKAR, V. Gremlin: Systematic Resilience Testing of Microservices. In *Proceedings of the IEEE 36th International Conference on Distributed Computing Systems (ICDCS'16)* (June 2016).

[44] HUNT, G., AND BRUBACHER, D. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium* (July 1999).

[45] HUO, C., AND CLAUSE, J. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)* (Nov. 2014).

[46] IRONPIGEON-133. make blob deletion tolerant of transient errors. https://github.com/AArnott/IronPigeon/pull/133, 2022.

[47] ISTIO DOCS. Istio Fault Injection. https://istio.io/latest/docs/tasks/traffic-management/fault-injection/, 2022.

[48] JANA, S., KANG, Y., OHIO, S. R., AND RAY, B. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proceedings of the 25th USENIX Security Symposium* (Aug. 2016).

[49] JAVA API SPECIFICATION. Class InheritableThreadLocal<T>. https://docs.oracle.com/javase/7/docs/api/java/lang/InheritableThreadLocal.html, 2022.

[50] JIANG, Z.-M., BAI, J.-J., LU, K., AND HU, S.-M. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *Proceedings of the 29th USENIX Security Symposium* (Aug. 2020).

[51] JU, X., SOARES, L., SHIN, K. G., RYU, K. D., AND SILVA, D. D. On Fault Resilience of OpenStack. In *Proceedings of the 12th ACM Symposium on Cloud Computing (SOCC'13)* (Oct. 2013).

[52] KINGSBURY, K., AND ALVARO, P. Elle: Inferring Isolation Anomalies from Experimental Observations. In *Proceedings of the VLDB Endowment* (Nov. 2020).

[53] KRYMETS, P. Unit testing and mocking with Azure SDK .NET. https://devblogs.microsoft.com/azure-sdk/unit-testing-and-mocking/, 2020.

[54] LAM, W., MUŞLU, K., SAJNANI, H., AND THUMMALAPENTA, S. A Study on the Lifecycle of Flaky Tests. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)* (May 2020).

[55] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[56] LU, J., LIU, C., LI, L., FENG, X., TAN, F., YANG, J., AND YOU, L. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (Oct. 2019).

[57] MAJUMDAR, R., AND NIKSIC, F. Why is Random Testing Effective for Partition Tolerance Bugs? In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[58] MARINESCU, P. D., BANABIC, R., AND CANDEA, G. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)* (June 2010).

[59] MARINESCU, P. D., AND CANDEA, G. LFI: A Practical and General Library-Level Fault Injector. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)* (June 2009).

[60] MARINESCU, P. D., AND CANDEA, G. Efficient Testing of Recovery Code Using Fault Injection. *ACM Transactions on Computer Systems (TOCS) 29*, 4 (Dec. 2011), 1–38.

[61] MEIKLEJOHN, C. S., ESTRADA, A., SONG, Y., MILLER, H., AND PADHYE, R. Service-Level Fault Injection Testing. In *Proceedings of the 2013 ACM Symposium on Cloud Computing (SOCC'21)* (Nov. 2021).

[62] MICROSOFT DOCS. AzureQueueDataManager.GetQueueMessage Method. https://learn.microsoft.com/en-us/dotnet/api/orleans.azureutils.azurequeuedatamanager.getqueuemessage, 2022.

[63] MICROSOFT DOCS. Install and use the Azure Cosmos DB Emulator for local development and testing. https://learn.microsoft.com/en-us/azure/cosmos-db/local-emulator, 2022.

[64] MICROSOFT DOCS. .NET CallContext Class. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.remoting.messaging.callcontext, 2022.

[65] MICROSOFT DOCS. .NET profiling. https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview, 2022.

[66] MICROSOFT DOCS. Table Storage error codes. https://learn.microsoft.com/en-us/rest/api/storageservices/table-service-error-codes, 2022.

[67] MICROSOFT DOCS. Transient fault handling. https://learn.microsoft.com/en-us/azure/architecture/best-practices/transient-faults, 2022.

[68] MICROSOFT DOCS. Use the Azurite emulator for local Azure Storage development. https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azurite, 2022.

[69] MOHAN, J., MARTINEZ, A., PONNAPALLI, S., RAJU, P., AND CHIDAMBARAM, V. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[70] NUGET. WindowsAzure.Storage NuGet. https://www.nuget.org/packages/WindowsAzure.Storage, 2022.

[71] ORLEANS-7738. Popping up queue messages may cause data loss and unexpected NullReferenceException. https://github.com/dotnet/orleans/issues/7738, 2022.

[72] ORLEANS-7790. Data was added repeatedly to the queue unexpectedly without any warning. https://github.com/dotnet/orleans/issues/7790, 2022.

[73] PILLAI, T. S., CHIDAMBARAM, V., KISWANY, S. A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[74] RAVINDRANATH, L., NATH, S., PADHYE, J., AND BALAKRISHNAN, H. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys'14)* (June 2014).

[75] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)* (May 2006).

[76] RFC 9110 HTTP SEMANTICS. Idempotent Methods. https://www.rfc-editor.org/rfc/rfc9110.html#name-idempotent-methods, 2022.

[77] RUBIO-GONZÁLEZ, C., GUNAWI, H. S., LIBLIT, B., ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI'09)* (June 2009).

[78] STACKOVERFLOW-39661635. Retry on 408 Timeout from Azure Table Storage service. https://stackoverflow.com/questions//retry-on-408-timeout-from-azure-table-storage-service, 2016.

[79] SUN, J., SU, T., LI, J., DONG, Z., PU, G., XIE, T., AND SU, Z. Understanding and Finding System Setting-Related Defects in Android Apps. In *Proceedings of the 2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)* (July 2021).

[80] SUN, X., LUO, W., GU, J. T., GANESAN, A., ALAGAPPAN, R., GASCH, M., SURESH, L., AND XU, T. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[81] TAN, C., JIN, Z., GUO, C., ZHANG, T., WU, H., DENG, K., BI, D., AND XIANG, D. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)* (Feb. 2019).

[82] WEIMER, W., AND NECULA, G. C. Finding and Preventing Run-Time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)* (Oct. 2004).

[83] YOO, S., AND HARMAN, M. Regression Testing Minimisation, Selection and Prioritization: A Survey. *Software Testing, Verification, and Reliability 22*, 2 (Mar. 2012), 67–120.

[84] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[85] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proceedings of the 15th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-XV)* (Mar. 2010).

[86] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., TANG, X., ZHOU, Y., AND SAVAGE, S. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Oct. 2012).

[87] ZHANG, P., AND ELBAUM, S. Amplifying Tests to Validate Exception Handling Code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)* (June 2012).

[88] ZHANG, Q., YU, G., GUO, C., DANG, Y., SWANSON, N., YANG, X., YAO, R., , CHINTALAPATI, M., KRISHNAMURTHY, A., AND ANDERSON, T. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (Apr. 2018).

Test Coverage for Network Configurations

Xieyang Xu[1] Weixin Deng[1] Ryan Beckett[2] Ratul Mahajan[1] David Walker[3]

[1]*University of Washington* [2]*Microsoft* [3]*Princeton University*

Abstract

We develop NetCov, the first tool to reveal which network configuration lines are tested by a suite of network tests. It helps network engineers improve test suites and thus increase network reliability. A key challenge in developing a tool like NetCov is that many network tests test the data plane instead of testing the configurations (control plane) directly. We must be able to efficiently infer which configuration elements contribute to tested data plane elements, even when such contributions are non-local (on remote devices) or non-deterministic. NetCov uses an information flow graph based model that precisely captures various forms of contributions and a scalable method to infer contributions. Using NetCov, we show that an existing test suite for Internet2, a nation-wide backbone network in the USA, covers only 26% of the configuration lines. The feedback from NetCov makes it easy to define new tests that improve coverage. For Internet2, adding just three such tests covers an additional 17% of the lines.

1 Introduction

As critical infrastructure, networks must be highly reliable but, unfortunately, network outages are common. A primary culprit is networks' reliance on complex, low-level configuration that dictates how routers select best paths and forward traffic. Day-to-day updates to network configuration are error-prone, leading to outages that knock off important online services (e.g., banking), ground airplanes, and disable critical communication (e.g., emergency calls) [3, 33, 34, 39, 45].

To improve network reliability, automatic testing and verification of configurations is becoming commonplace. Today, network operators have at their disposal many tools with increasing sophistication that can scale to large networks and check various aspects of network behavior [5, 23, 40, 49, 51].

However, using such tools is not sufficient by itself; one must also use them *effectively*. Outages can occur despite automated testing when the test suite is poor and does not cover key aspects of network configuration. This was the case with the massive Facebook outage during which Facebook, WhatsApp, Instagram, and Oculus were unavailable for six hours [35]. Current tools have pushed the limits of *what can be tested* but left open the question of *what needs to be tested*.

Without tool support, it is difficult for engineers to know if they are effectively testing network configurations. In industrial networks with possibly millions of lines of configurations, engineers' understanding of network behavior and dependencies is necessarily incomplete. It is even harder to update an existing test suite after the network evolves because the engineers likely do not know what the old test suite is or is not testing for the updated network.

Recent work has proposed data plane coverage [47] to reveal testing gaps. It shows which data plane elements, such as forwarding rules, are exercised by a test suite. However, well-tested data plane does not imply well-tested configurations. Data plane elements are the output of network's configurations (which define its control plane) and the current operating environment (failures, external routing information). Testing a given data plane only tests configuration elements that are exercised in that particular environment. Other configuration elements are not tested. We demonstrate this empirically via a scenario where testing *all* data plane elements leaves over half of configuration lines untested.

We develop *configuration coverage* to provide comprehensive and precise feedback to network engineers on test suite quality. Our goal is to identify exactly which configuration lines are tested and which ones are not. We want to consider all configuration elements, not only those that contribute to the current data plane. Revealing exactly which lines are untested helps improve tests—add tests that target untested lines—which in turn can improve network reliability. This is similar to how code coverage tools help improve tests and software reliability [9, 11, 22].

A major challenge we face is that many network tests do not exercise configurations directly. Instead, they reason about the data plane elements produced by configurations. We need to infer the configuration elements that contribute to the tested data plane elements. This inference is complicated because contributions can be non-local and non-deterministic. In a distributed control plane, a piece of tested routing information may have been propagated and transformed multiple times along its path, and both local and non-local configurations may have contributed to its existence. For example, the path attributes of a BGP route is shaped by routing policies on each and every hop that it traverses. Further, not all contributions are deterministic. For instance, any one of possibly multiple sub-prefixes can lead to the route of an aggregate prefix. We must scalably account for local and non-local contributions and for non-deterministic contributions.

Our solution is to model the contribution between configuration elements and data plane elements as an *information*

flow graph. An IFG is a directed acyclic graph (DAG) where vertices denote network elements and edges denote contributions. In addition to direct contributions from configuration elements to data plane elements, we also model contributions between data plane elements (from predecessors to successors). For instance, a BGP route contributes to the BGP message that derived from it. Indirect contributions are thus modeled by multi-hop paths in the DAG. When contributions exhibit non-determinism, we use special *disjunctive* nodes to organize possible DAG paths that may contribute to a given data plane element.

We build a tool called NetCov based on this model. It annotates which configuration lines and logical elements are tested by a given test suite and produces aggregated coverage statistics. To efficiently map tested data plane element to the set of contributing configuration elements, it materializes the IFG lazily, instead of tracking contributions proactively, during data plane generation. This design avoids the cost to compute and store contributions for transient or untested data plane elements. NetCov is open-sourced on GitHub [30].

We evaluate NetCov on Internet2, a nation-wide backbone network in the USA, and on synthetic data center networks. We show that test suites proposed in prior work can have poor coverage. The three tests proposed by Bagpipe [44] covered only 26% of the configuration lines of Internet2. We also show how surfacing untested configuration elements suggests new tests that improve coverage. By adding just three such tests to the Internet2 test suite based on NetCov's feedback, we could improve coverage to 43%, and more similar tests can be added to further increase coverage. NetCov performs reasonably well. The time to compute coverage is 1.2 hours for the largest network that we study, which has over 2 million forwarding rules. This time is an order of magnitude less than the time to execute tests.

Stepping back, we note that networking is not alone in its reliance on configuration. Today, a lot of infrastructure and distributed applications are deployed by composing existing components using configuration (e.g., infrastructure deployment using Terraform, and application deployment using containers and service meshes). These configurations are central to correct behavior, which is why there is an intense focus on testing them properly [21, 38, 43]. As for networks, there are no tools to help engineers discover how well the configurations are tested. The techniques developed in our work, the IFG-based contribution tracking and its lazy traversal, can provide a starting point toward better testing of infrastructure and distributed application configuration as well.

2 Background on Network Testing

In networks with distributed control planes, each device runs one or more routing protocol (e.g., BGP, OSPF) instances. Each instance exchanges routing messages with its neighboring instances. Routing messages contain attributes of paths that the sender is using to various destinations. A routing instance may learn multiple paths to the same destination via different neighbors. It selects the best one (or multiple best ones if multipath routing is enabled) based on its policy and stores that path in its protocol RIB (routing information base). Multiple routing protocol instances on a device may have best paths to the same destination. The device selects the best one(s) based on the relative preference of the protocols and stores the selection in its main RIB. Information in the main RIB is used to forward packets.[1]

Network engineers can control many aspects of the computation above using device configuration. This includes the routing protocol instances that are running; the peering between instances; the destination prefixes that are announced by each routing protocol instance; how routing messages are transformed prior to sending (export policy) and upon reception (import policy); and the preference function for best path selection. Naturally, thus, how the network forwards packets is intimately dependent on device configurations.

Given the importance of configurations to correct network behavior, network engineers use automatic testing to find bugs and gain confidence in their correctness. Network tests come in two flavors. *Data plane tests* analyze the computed data plane state (*i.e.*, RIBs), e.g., checking that node A can reach B and that route to a particular destination is present at node C. *Control plane tests* directly analyze device configuration, e.g., checking that the import policy blocks routing messages for private address space (such as 10.0.0.0/8) and BGP peerings are correctly configured.

3 Configuration Coverage: Overview

Network engineers today create data and control plane tests based on past outages and their knowledge of which behaviors are important to test. There are no tools to provide feedback on how well they are testing configurations and which aspects of the configuration are untested. We aim to build such a tool. Given the complexity of real-world networks, it is difficult for humans to know if they have covered all important elements of configurations. As with software, high coverage is necessary but not sufficient for a good test suite. In addition to exercising all key behaviors, the tests must also properly assert that those behaviors match intent. This latter task is not our focus.

Our goal is to reveal which elements of the network configuration are covered by a suite of data and control plane tests. Before discussing our approach, we define what it means for a configuration element to be covered.

[1] In reality, for fast forwarding, routers have a forwarding information base (FIB), which maps each main RIB destination to its outgoing interface, by recursively resolving next hop information (which may be an IP address). The difference between main RIB and FIB is not material for our work, and we use the term main RIB for the table that has forwarding information.

3.1 Defining coverage

We deem a configuration element to be covered if it *i*) is tested directly by a control plane test; or *ii*) contributes to the production of a data plane state element (i.e., an entry in the protocol or main RIB) tested by a data plane test. For now, assume that contributions are deterministic. We discuss non-deterministic contributions in the next section.

Figure 1 illustrates configuration coverage as a result of a data plane test. It shows parts of the two routers' configuration. R1's configuration defines one interface (Lines 1-2) and one BGP peer (192.168.1.2, which is R2's address), and it specifies the import and export policy to use. The import policy (R2-to-R1 at Lines 6-11) denies routing messages for a particular prefix and sets the preference for another.

R2's configuration defines two interfaces, a BGP peer (R1) and routing policies. At Line 13, it states that the prefix 10.10.1.0/24 should be announced to BGP peers *iff* it is in the main RIB.[2] In our example, 10.10.1.0/24 will be in the main RIB as it corresponds to the eth1's prefix. (Address statements like Line 4 encode the IP address and prefix length. For eth1, given the address 10.10.1.1 and prefix length of 24, the prefix is 10.10.1.0/24.) Routers add interface prefixes to the "connected" protocol RIB, from where those prefixes can enter the main RIB. The resulting RIBs on the two routers are shown in the figure. Each entry includes the next hop and source routing protocol ("conn" = connected).

Suppose the entry for 10.10.1.0/24 at R1 was tested by a data plane test. The covered configuration elements are highlighted. On R1, the BGP peer configuration and import policy binding (Lines 3-4) are covered because the tested entry came via that peering and passed through that policy. Parts of the routing policy R2-to-R1 relevant to the tested state (Lines 6, 9-11) are also covered. The interface definition (Lines 1-2) is covered because it enables the BGP peering to be established. In contrast, the export policy R1-to-R2 and unexercised parts of R2-to-R1 (Lines 7-8) are not covered.

There are covered configuration elements at R2 as well. These include the interface definitions—eth0 enables the BGP edge and 10.10.1.0/24 was announced due to eth1—and BGP peering, the export policy, and the BGP network statement.

Alternative definitions of coverage. One may consider an alternative definition of coverage that disregards non-local configuration elements. But we posit that including non-local elements is more meaningful. These elements, such as the BGP network statement on R2's Line 13, are just as key to the existence of 10.10.1.0/24 at R1 as the local elements.

Another definition of coverage is based on mutation [4]: a configuration line is deemed covered if its mutation alters the test result. Compared to the definition of coverage we adopt, mutation-based coverage will report an additional class of configuration elements as covered—configuration elements

[2]Different router vendors have different semantics for BGP network statements. We are assuming Cisco semantics.

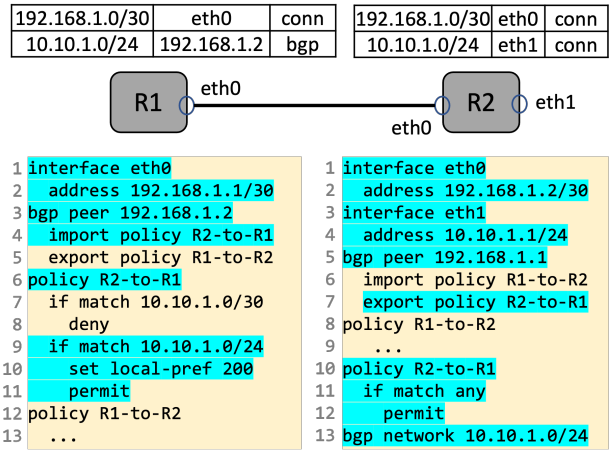


Figure 1: An example network with routing tables and configurations. The highlighted configuration lines are covered when the route to 10.10.1.0/24 is tested at R1.

that de-prioritize (or reject) the competitors of the tested data plane element. Mutation-based coverage tends to be significantly harder to compute [24], and its results can be hard to interpret. In developing the first tool in this space, we decided to focus on a simpler, more direct definition of coverage. We will explore more sophisticated definitions in the future.

3.2 Our approach

While it is straightforward to identify configuration elements covered by a control plane test, it is not so for data plane tests. Data plane tests analyze the "output" of the control plane, and we need a scalable way to compute which configuration elements contributed to tested data plane state. The relationship between these inputs and outputs is complex. How a particular RIB entry comes about relies on many configuration elements across multiple devices. The need to map tested outputs to input space sets computation of configuration coverage apart from data plane coverage and software coverage, for both of which the coverage domain is the same as test domain.

To motivate our approach to solving this problem, let us first sketch two strawman approaches. One potential approach is to express control plane computation declaratively, e.g., in Datalog. This enables identification of contributing inputs for a given output using a form of backward-reasoning [46, 52]. However, network control plane computations can be quite complex (e.g., non-monotonic behaviors [16,36]). While declarative encodings may work in special cases [27], it is generally hard to get high-fidelty, performant encodings. That is why most control plane analysis tools use an imperative approach [12, 31, 32, 49].[3]

[3]Batfish [12], a widely used control plane analysis tool, originally used Datalog to encode network control planes but switched to imperative simulations due to expressiveness and performance challenges.

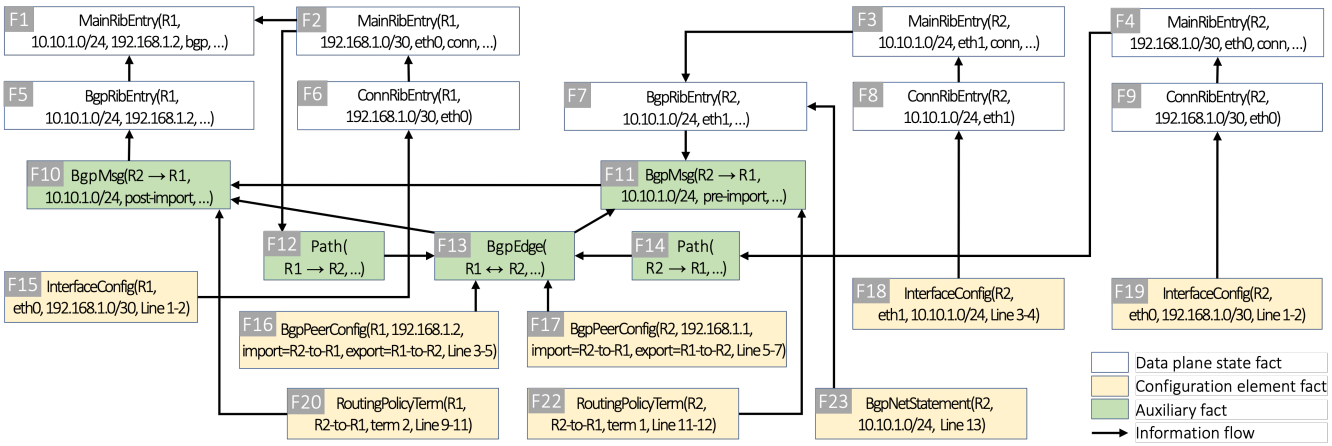


Figure 2: Subset of the IFG for the Figure 1 example. It tracks configuration elements that contribute to the tested RIB entry (F1).

Another potential approach is to use simulation-based forward reasoning, i.e., simulate the control plane (imperatively) and track which configuration elements feed into each part of the data plane state. However, this approach has scalability limitations. Network simulation is time and memory intensive [12, 32, 49], and it will become significantly worse if it needed to track all necessary information along each hop.

Our approach is based on two observations. First, for the purposes of computing coverage, we do not need a full computational model of the control plane. We need to only track which configuration elements contribute to tested data plane state (i.e., taint analysis [41]), not the exact input-output relationship; and we need to reason only about the stable state (i.e., the the of devices once they have settled on best paths), not the transient states. Data plane testing [19, 23, 25, 26, 48] assumes that the analyzed state is stable. Our second observation is that the stable state contains enough information for us to infer contributions of configuration elements after the fact, based on the semantics of the control plane. This inference is vastly cheaper than tracking contributions towards all data plane state entries, independent of whether they are tested.

To model contributions to the stable state, we use an *information flow graph* (IFG). Figure 2 shows a subset of the IFG for the example in Figure 1. Each node is a *fact* and arrows denote information flow from the tail to head. IFGs have three types of facts: *i*) data plane state, *ii*) configuration elements, and *iii*) auxiliary facts that capture intermediate dependencies between data plane state and configuration elements.

The main RIB entry 10.10.1.0/24 at R1 (F1) is derived from the corresponding BGP RIB entry (F5), which in turn is derived from the BGP message from R2 (F10). This message exists because of the BGP edge between R1 and R2 (F13), the source message sent by R2 (F11), and the relevant configuration element within import policy (F20). R2 sent the BGP message because of the same BGP edge (F13), its export policy elements (F22), and the BGP RIB entry (F7). This BGP RIB entry exists because of the configuration element (F23) and the RIB entry (F3), which exists because of the connected route (F8). The BGP edge (F13) exists because of the configuration elements that define the peering (F16, F17) and paths between R2 and R1 that enable the BGP session to be established. The paths depend on the RIB entries (F2 and F4, respectively), the contributions to which can be similarly traced. In this manner, the IFG captures all configuration elements that led to the tested RIB entry (F1).

We do not track IFG dependencies proactively but infer them on-demand based on control plane semantics, using a mix of backward-forward reasoning. Backward inference infers the parent (tail) of the edge from its child (head). The information in child nodes is not enough to fully recover the parent nodes, but is often enough to select them from the known stable state. For instance, we can compute the BGP RIB entry F5 from the main RIB entry F1—the main RIB entry indicates that its source routing protocol is BGP, and we thus look up the BGP RIB for 10.10.1.0/24.

Lookup-based inference does not always work. For instance, given a BGP message which has passed through an import policy, we cannot compute backwards which terms of the import policy were exercised (F10 ← F20). Another parent of F10, the pre-import BGP message (F11) cannot be looked up either because it is not part of the input and needs to be computed on-the-fly. To address these limitations, we combine backward and forward inference. When a parent can not be directly looked up, we first look up the prerequisites of the parent. For instance, we can look up F7 based on F10. Next, we use targeted simulations to compute non-existing facts and to select relevant facts exercised in a control plane process or data plane process. For instance, given the BGP route at R2 (F7), we simulate its processing through the export policy, which allows us to derive the pre-import BGP message (F11) and find the policy term exercised during the export process (F22). Once F11 is computed, we conduct another targeted

simulation to discover the policy term exercised in the import process (F20). Unlike a full control plane simulation, these targeted simulations are fast. They have limited scope (e.g., best path selection is not simulated) and are done only for messages of interest, not all messages.

By combining backward and forward inference, atop the stable state IFG, we can scalably discover all covered configuration elements. We describe this approach in detail next.

4 Design of NetCov

NetCov takes as input configuration files, data plane state (protocol RIBs, main RIB and active routing edges) of the network. The data plane state may be pulled from live network or produced by control plane analysis tools [12, 32, 49]. In addition, NetCov takes as input what is tested: data plane entries that are tested by data plane tests, and configuration elements that are tested by control plane tests. This information is produced by network testing tools [12, 47].

Based on these inputs, NetCov computes which configuration elements are covered. The core of this computation efficiently mapping a data plane fact to configuration elements that contribute to it. We describe this computation next.

4.1 Information flow model

IFGs are directed acyclic graphs whose nodes denote network *facts* and edges denote information flow between facts. Table 1 shows the types of network facts modeled by NetCov and the information flow between different types. Data plane state has three subtypes: main RIB entries, protocol RIB entries, and access control list (ACL) entries.

Auxiliary facts have three subtypes: routing edges, routing messages, and paths of routing messages. These facts are not strictly necessary, but they help create a compact IFG and speed up graph walking. For instance, the routing messages of many protocol RIB entries depend on the same path which in turn may depend on many main RIB entries. Adding an explicit fact for the path avoids the need to add all pairs of edges between routing messages and main RIB entries.

In our model, the auxiliary facts for routing messages represent messages between routing protocol instances across devices as well as within a device, i.e., redistribution [10]. This uniform treatment is a modeling convenience. In reality, explicit messages are not exchanged during redistribution (though redistribution is subject to routing policies akin to messages between cross-device routing instances).

The last column of Table 1 shows how information flows among different types of facts. A main RIB entry stems from a protocol RIB entry and optionally another main RIB entry (when its next hop is an IP address whose corresponding output interface needs further resolution). A protocol RIB fact stems from a routing message (for protocols such as BGP), a configuration element (for connected interfaces and static routes), a main RIB entry accompanied with a configuration element (such as when a BGP network statement populates a main RIB entry into BGP RIB) or a set of RIB entries accompanied with a configuration element (for aggregate routes). ACLs facts stem from configuration facts and have no other dependencies. Routing messages stem from a RIB fact or another message (e.g., post-import-policy message depends on pre-import-policy message), and they also depend on routing edges and routing policy configurations. Inter-device routing edges stem from paths that enable sessions to be established and configuration facts that define peerings; Intra-device routing edges stem from configuration facts that define redistribution. Finally, path facts depends on main RIB facts and ACL facts that impact routing traffic along the way.

Network fact		**Information flow**
Configuration element (c)		None
Data plane state	Main RIB entry (f)	$f_i \leftarrow r_j$ $f_i \leftarrow r_j, f_k$
	Protocol RIB entry (r)	$r_i \leftarrow m_j$ $r_i \leftarrow c_j$ $r_i \leftarrow f_j, c_k$ $r_i \leftarrow \{r_{j_1}, ...\}, c_k$
	ACL entry (a)	$a_i \leftarrow \{c_{i_1}, ...\}$
Auxiliary	Routing message (m)	$m_i \leftarrow r_j, e_k, \{c_{l_1}, ...\}$ $m_i \leftarrow m_j, e_k, \{c_{l_1}, ...\}$
	Routing edge (e)	$e_i \leftarrow \{c_{j_1}, ...\}$ $e_i \leftarrow \{c_{j_1}, ...\}, \{p_{k_1}, ...\}$
	Path (p)	$p_i \leftarrow \{f_{j_1}, ...\}, \{a_{k_1}, ...\}$

Table 1: Information flow model: Types of facts and all possible information flows for each type. $\{t, ...\}$ denotes a set of facts.

For correct computation of coverage, the IFG model must be sound and realizable. Soundness means that it includes all relevant dependencies (per control plane semantics) and no more. Realizable means parents (tails) along all information flow edges can always be inferred, via lookup or simulation or a mix. Our model is sound to our knowledge; and that we are able to use it to compute coverage, using the framework described next, points to its realizability.

4.2 Inferring the IFG on demand

Based on the information flow model, NetCov uses a backward-forward inference framework to lazily materialize the IFG from any set of facts whose coverage need to be tracked. The framework is abstracted using a set of *inference rules* and an iterative construction algorithm. Each inference rule is function that takes a materialized IFG node as input and materializes a set of its ancestor nodes as well as the

Algorithm 1: Rule to infer BGP RIB entry from main RIB entry.

```
1  def infer_from_main_rib_entry(f,
   ↪ stable_state):
2    if not (f is MainRibEntry and f.protocol ==
     ↪ 'bgp'):
3      return []
4    bgp_entry = stable_state.bgp_rib.lookup(
5      host=f.host,
6      prefix=f.prefix,
7      nexthop=f.nexthop,
8      status='BEST'
9    )
10   return [(bgp_entry, f)]
```

Algorithm 2: Rule to infer ancestors of a post-import BGP message.

```
1  def infer_from_bgp_message(m, stable_state):
2    if not (m is BgpMsg and m.is_post_import):
3      return []
4    bgp_edge = stable_state.bgp_edges.lookup(
5      recv_host=m.host
6      send_ip=m.nexthop
7    )
8    origin_entry = stable_state.bgp_rib.lookup(
9      host=bgp_edge.send_host,
10     prefix=r.prefix,
11     status='BEST'
12   )
13   pre_import_msg, export_clauses =
     ↪ policy_simulation(
14     input=origin_entry,
15     policy=bgp_edge.export_policy
16   )
17   _, import_clauses = policy_simulation(
18     input=pre_import_msg,
19     policy=bgp_edge.import_policy
20   )
21   return [(pre_import_msg, m), (bgp_edge, m)]
     ↪ +
22     [(cl, m) for cl in import_clauses] +
23     [(origin_entry, pre_import_msg), (bgp_edge,
       ↪ pre_import_msg)] +
24     [(cl, pre_import_msg) for cl in
       ↪ export_clauses]
```

edges the allows the ancestors to reach the input node. These nodes and edges will be merged into the materialized IFG by the construction algorithm. The implementation of these functions uses one or both of the *lookup-based inference* and *simulation-based inference*. Let us elaborate.

Lookup-based inference. The computation of data plane state is lossy. While a main RIB entry may be derived from a BGP RIB entry, we cannot infer the complete BGP RIB entry from the main RIB entry because BGP specific attributes (e.g., AS-path) are not preserved in the main RIB.

To handle this information loss, our inference takes two steps. It first infers attributes that can be known from heuristics (we know such heuristics from control plane semantic, *e.g.*, the BGP RIB entry should have the same prefix as the main RIB entry derived from it). Next, we look up all entries in the stable state that match the inferred attributes. For instance, Algorithm 1 shows the simplified function to infer the BGP RIB entry that led to a main RIB entry. Based on control plane semantics, if a main RIB entry indicates its source protocol to be BGP, it must have stemmed from a BGP RIB entry on the same router with the same `prefix` and `nexthop` attributes (Lines 5-7). Besides, the BGP RIB entry should have been selected as the best route (Line 8). Such information is enough to uniquely identify the parent within the known stable state. The return value (Line 10) is a list of tuples denoting the IFG edges materialized by this rule.

Simulation-based inference. Lookup-based inference falls short in two scenarios. First, when a parent fact is absent from the known stable state (e.g., routing messages), and second, when the heuristics fail to infer enough information so as to uniquely identify the parents (e.g., we cannot know which policy clauses are used in the production of a BGP route by looking at the resulted route). We use local simulations to complement lookup-based inference. But simulations can only be performed in the forward direction, *i.e.*, to compute a fact using simulations, we first need to know its parent. We use a generalized version of lookup-based inference to discover grandparent facts of a known fact, and then use simulations with the grandparents to infer their children (i.e., parents of the original fact).

Algorithm 2 shows the simplified inference rule that infers the ancestors of a post-import BGP message. Line 13 demonstrates the use of simulation-based forward inference to compute a missing parent fact on the fly. The two prerequisites to simulate the BGP message–the grandparent BGP RIB entry (`origin_entry`) and the BGP edge–are discovered via lookup-based backward inference, on Line 8 and Line 4 respectively. The simulation returns the derived BGP message after applying the routing policy, as well as the policy clauses exercised during the process. The second forward-simulation (Line 17) is to discover the policy clauses that are hit during the import process. The return value includes the inferred IFG edges that connect to the input node `m` as well as ones that connect to parent `pre_import_msg`. The former corresponds to information flow $m_i \leftarrow m_j, e_k, \{c_{l_1}, ...\}$ in Table 1 and the latter corresponds to $m_i \leftarrow r_j, e_k, \{c_{l_1}, ...\}$.

IFG construction. Next, we detail IFG materialization using inference rules. Assume for now that the information flow

Algorithm 3: IFG lazy materialization

Input: Initial nodes $\{v_i\}$; Inference rules $\{\phi_i : v \mapsto \{(u_i, v_i)\}\}$;
Output: Materialized IFG (V, E)
Data: Stable state data plane state (main RIB and protocol RIBs); Routing edges; Configuration elements;

```
1  Procedure BuildIFG({v_i}, {φ_i})
2      V, E ← {v_i}, ∅
3      V_prev ← {v_i}      // dirty nodes of previous iteration
4      while |V_prev| > 0 do
5          V_curr ← ∅      // dirty nodes of current iteration
6          foreach c ∈ V_prev do
7              foreach φ ∈ {φ_i} do
8                  E' ← φ(c)
9                  foreach (u_i, v_i) ∈ E' do
10                     if u_i ∉ V then
11                         V ← V ∪ {u_i}, V_curr ← V_curr ∪ {u_i}
12                     if v_i ∉ V then
13                         V ← V ∪ {v_i}, V_curr ← V_curr ∪ {v_i}
14                     if (u_i, v_i) ∉ E then E ← E ∪ {(u_i, v_i)}
15         V_prev ← V_curr
16     return (V, E)
```

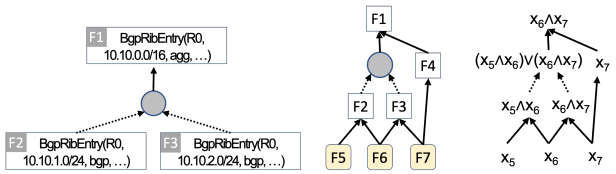
is deterministic; the next section discusses how we handle non-determinism.

As shown in Algorithm 3, the IFG initially contains only the nodes representing the tested data plane state facts from the input and does not have any edges (Line 2). It is then iteratively expanded by applying inference rules on existing nodes. In each iteration, all inference rules are applied to the dirty nodes derived from the previous iteration (Line 8). The new nodes and edges inferred during such process are collected and merged (with deduplication) into the IFG (Line 9-14). The computation repeats until no new facts can be derived in an iteration.

4.3 Handling uncertainty

There are situations where it is not certain which stable state facts contributes to a given fact. One such scenario is BGP aggregation, where a prefix (e.g., 10.10.0.0/16) is added to the RIB iff at least one more of its more specific prefixes (e.g., 10.10.1.0/24) is present. When multiple more specifics are present, we do not know which one triggered the aggregate. Another such scenario is when multiple paths are available for a routing edge to be established, which can happen when the network uses multipath routing. Here, we do not know which path is actually used by routing messages.

It is important to model and report such uncertainty because the notion of contribution is different. Unlike deterministic contribution, when the contribution is non-deterministic, one or more parent facts can disappear without impacting the outcome represented by the child. Our experiments have scenarios where 78% of the configuration lines have non-deterministic contribution, and the tested fact would not be



(a) (b) (c)

Figure 3: Modeling uncertainty. (a) BGP aggregate (F1) has two potential contributors. (b) F5 is weakly covered but F6 and F7 are strongly covered. (c) The predicates of IFG nodes.

impacted if any of them did not exist. Not separating such uncertain contribution would lead to misplaced confidence in how well configurations are tested.

We model contribution uncertainty using *disjunctive* nodes in the IFG. This node points to the parent fact (e.g., the aggregated RIB fact) and the multiple contributors to the parent point to this node. See Figure 3(a) for an example where a BGP aggregate could be triggered by either of the two more specific prefixes. When our inference rules encounter uncertainty during IFG materialization, they produce a disjunctive node and attach all contributors to it as children.

We introduce the notion of *weak* coverage to capture the configuration elements whose contribution to the tested facts is not critical. We define a contribution as non-critical if the tested fact will not be affected by deleting the configuration element from the IFG. In Figure 3(b), F5 is weakly covered when F1 is tested because F1 can be derived without any contribution from F5, via F2 and F6. On the other hand, F6 is strongly covered because, without it, neither F2 nor F3 can be derived and thus the disjunctive node cannot be derived. F7 is also strongly covered because it contributes to F4, which is essential to F1.

NetCov labels each covered configuration element as strong or weak after the materialization of the IFG. The label is determined as follows. We first assign a Boolean variable to each configuration element in the IFG. Next, we build a Boolean predicate of each IFG node on top of these variables. The predicate of a fact depends on the predicate of its ancestors in the IFG: A normal node depends on the conjunction of its immediate parents, and a disjunctive node depends on the disjunction of parents. Therefore the predicate of any IFG node is ultimately composed of the variables associated with configuration elements that lead to it, denoted as $\Gamma(v) = F(x_1, \ldots, x_n)$. Figure 3(c) shows the predicates of IFG nodes in Figure 3(b). We represent these Boolean predicates using Binary Decision Diagrams (BDDs) [8] and build BDD predicates by traversing the IFG. By definition, a configuration fact (denoted as x_i) is strongly covered if and only if there exists a tested data plane state fact (denoted as v), v is reachable from x_i in the IFG, and x_i is a necessary condition of $\Gamma(v)$. Therefore, once the predicates are built, we test graph reachability and logical necessity between each pair of configuration facts and

tested data plane facts. Necessity $\neg x_i \Rightarrow \neg\Gamma(v)$ is equivalent to unsatisfiability of $\neg x_i \wedge \Gamma(v)$. While (un)satisfiability is NP-Complete in general cases, we note that it is efficient in our case—it can be reduced to computing the cofactor $\Gamma(v)|_{x_i=0}$ and testing whether the cofactor is constant false, both of which are efficient using BDD operations.

We further reduce the size of BDD predicates by precluding configuration facts that can reach tested facts via a path with no disjunctive node, such as node F7 in Figure 3(b). These configuration facts must be strongly covered so their necessity do not need to be tested. Besides, their validity variables can be replaced with constant true when building BDD predicates, which will not affect the strong/weak classification of other configuration elements. We empirically find this heuristic to be effective in reducing the number of variables used for weak coverage computation.

4.4 Future Extensions

Our current model tracks the contribution of configuration elements to concrete data plane state entries. While this view aligns well with tools that perform data plane testing [50], data plane verification [25, 29], and control plane testing [12, 49], it is not applicable to control plane verification tools [1, 7] that reason about data plane symbolically (i.e., simultaneously reason about multiple data planes under different environments). Control plane verification tools turn configuration into an internal model that is used for validation. NetCov can be extended to these tools by tracking how configuration elements contribute to the model, akin to how compilers link program source information to its intermediate representations.

The current implementation of NetCov supports BGP, a path vector protocol, and static routes. Other protocols, including link state protocols (e.g., OSPF) and label switching protocols (e.g., MPLS) can be supported with appropriate extensions. Such extensions require defining protocol-specific configuration elements and data plane state facts (such as label information base entry for MPLS) as well as all new information flows.

5 Implementation

We implemented NetCov with 4,000 lines of Python code. A total of 18 lambdas (Python functions) encode the IFG inference rules. NetCov uses Batfish [6] to extract configuration elements from configuration files and to run targeted simulations, and it uses CUDD [37] for BDD operations.

NetCov supports several major router vendors supported by Batfish, including Arista, Cisco, and Juniper. It builds a vendor-neutral representation of configuration elements using vendor-specific information provided by Batfish. Table 2 lists the configuration elements that NetCov currently analyzes.

NetCov may not consider all components of a device's configuration. One category of such components is device management configuration (e.g., login settings), which does not impact data or control plane functionality. The second category is control plane components that are not currently modeled by NetCov. This includes IPv6 (which is not modeled by Batfish currently) and routing protocols other than BGP (e.g., OSPF). The presence of unconsidered components does not imply that NetCov cannot be used for that network. As we show in the next section, NetCov provides helpful coverage information for parts that are considered.

Type	Purpose
Interface	Interface and its settings (e.g., addresses)
BGP peer	BGP peer settings (e.g., IP address, AS number)
BGP peer group	BGP peer settings inherited by one or more peers
Route policy clause	One clause in an export or import route policy
Prefix list	List of prefixes, used in route policy clauses
Community list	List of BGP communities for route policy clauses
AS-path list	List of AS-path expressions for route policy clauses

Table 2: Configuration elements analyzed by NetCov.

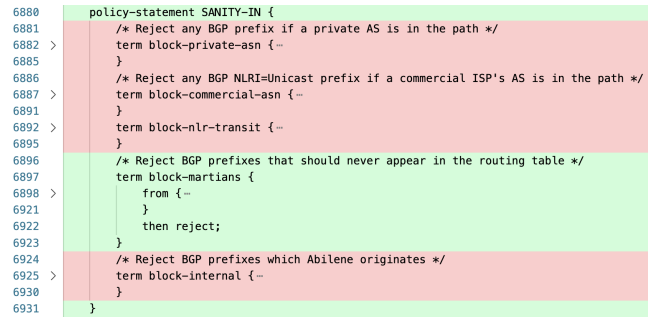
After constructing the IFG, which yields information on which configuration elements are covered, NetCov computes which lines are covered. NetCov leverages the Batfish parser to map configuration elements to line numbers. Each element typically spans multiple configuration lines, and when an element is covered, it deems all of those lines as covered.

Based on element and line coverage, NetCov produces three main outputs. The first is a coverage report at the granularity of individual lines (or elements). We produce this report in the `lcov` format, which is supported by common code coverage tools and enables users to visualize coverage results as annotations on configuration files. See Figure 4(a) for an example. The second is coverage aggregated at the file level, generated with the help of GNU LCOV [17]. See Figure 4(b) for an example. The third output is coverage aggregated by the type of configuration element, which shows what fraction of elements of each type are covered.

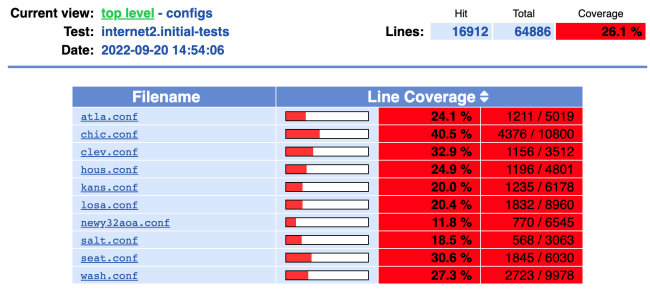
These outputs help users uncover testing gaps and improve their test suites in different ways. The aggregate results help identify systematic gaps such as "router A is poorly covered" or "routing policy clauses are poorly covered." The line-level results help them zoom in to specific gaps and develop tests that target them. The case study in the next section demonstrates this test suite improvement process.

6 Case Studies

We present case studies of using NetCov on two disparate networks, one a wide-area backbone and another a datacenter. In each case, using realistic test suites, we show that NetCov provides insight into what is and is not covered and how these insights help improve the test suites.



(a) Line-level coverage. Green background denotes covered lines, and red denotes uncovered lines. Some lines are collapsed for simplicity.



(b) File-level aggregate coverage. The overall coverage is at top right, and the coverage for individual files (devices) is in the table.

Figure 4: Example NetCov outputs.

6.1 Case Study I: The Internet2 backbone

Internet2 is a nation-wide network that connects over 60,000 US educational, research and government institutions. The routing design of Internet2 is typical of backbone networks. It has 10 BGP routers spread across the country. The routers are organized as a single autonomous system (AS), and they establish iBGP full mesh on top of internal reachability provided by the IS-IS protocol. The Internet2 routers connect to 279 external BGP peers, and heavily use route import and export policies. The import policy for an external peer has multiple policy statements, some specific to the peer and some shared within the same peer group. Peer-specific policies tend to specify a list of allowed prefixes from this peer, and others are used for sanity checking, preference setting, etc. Export policies are similarly structured.

Internet2's configurations that we study have 96,672 lines (in Juniper's JunOS format) across all routers. Of these, NetCov's coverage computation considers 64,886 lines. The bulk of the unconsidered lines correspond to device management, IPv6, and IS-IS protocol.

We do not have the data plane state of Internet2, which is needed to run data plane tests. We approximate it using Route Views [42], a repository of BGP routes from over two hundreds ASes worldwide. This data helps approximate BGP messages that external peers of Internet2 send to it. Consider a peer with AS number X. If we find a prefix P in RouteViews with AS-path $[A,X,Y]$, we assume that the peer sends P to Internet2 with AS-path $[X,Y]$. The existence of AS-path $[A,X,Y]$ means that AS A must have a route to P with AS-path $[X,Y]$, which it announces to its neighbors. If we find multiple AS-paths for a prefix, we pick the one with fewest AS hops.

We use these BGP messages that each peer sends to Internet2 as inputs to simulate Internet2's control plane using Batfish. The data plane state produced by this simulation is a coarse approximation of the real version, but it suffices to meet our goals of running data plane tests and characterizing configuration coverage.

6.1.1 Test suite coverage

To study how NetCov analyzes coverage for realistic test suites, we use the test suite proposed in Bagpipe [44]. It has three tests to validate Internet2's BGP configuration.

- *BlockToExternal*: ensure that BGP routes with BTE community are not announced to any external (eBGP) peer.
- *NoMartian*: ensure that incoming BGP messages from external peers for prefixes in the private address space ("Martian") are rejected.
- *RoutePreference*: ensure that if multiple routes to the same prefix are accepted from multiple external neighbors, the selected route belongs to the most preferred neighbor. The neighbor's preference depends on commercial relationship [13]. *Customers* are most preferred, followed by *p*eers, and then *providers*[4].

We implemented these tests using Batfish. *BlockToExternal* and *NoMartian* are control plane tests. *BlockToExternal* evaluates all BGP export policies on a set of BGP routes carrying the BTE community and asserts that the result be rejection. We generate the test cases by sampling BGP routes from the data plane state and attaching the BTE community to them. *NoMartian* evaluates all BGP import policies on a set of BGP routes destined for Martian addresses and asserts that the results be rejection. *RoutePreference* is a data plane test. It focuses on destination prefixes available via multiple neighbors and asserts that their local preferences reflect commercial relationship. We use CAIDA data [28] to infer commercial relationship between Internet2 and its BGP neighbors.

After running this test suite on Internet2, we find that it covers only 26.1% of configuration lines across all devices. Only a tiny fraction of configuration lines (0.5%) are weakly covered, so we do not separate weak/strong coverage for this case study; we will do that in the next one.

[4] As a not-for-profit network, Internet2 treats its member institutions as customers and other not-for-profit networks (such as ESNet) as peers. Internet2 does not have providers in its routing preference model.

To help understand what is and is not covered in more detail, NetCov enables network engineers to look at the data from multiple perspectives. Figure 4(b) shows per-device coverage. We see notable variation across devices, from 11.8% to 40.5%. As we show below, the test suite has systematic gaps, and the cross-device variation stems from different devices having different fractions of covered configuration elements.

Figure 5 shows the coverage broken down by the type of configuration elements. For simplicity, we create four buckets of element types, as shown in the legend. The bottom bar shows the fraction of reachable configuration lines in each bucket. The "Test Suite" bar shows the covered fraction of those lines, and the top three bars show the coverage of individual tests. The total coverage of individual tests is 0.6%, 0.9% and 24.7% respectively. *BlockToExternal* and *NoMartian* cover only one type of configuration element (routing policies), and even within this type, they cover a small fraction. *RoutePreference* covered all four buckets but its overall coverage is still limited.

Finally, NetCov reports that 27.9% of configuration lines are "dead code" that will never be exercised. They include defined BGP peer groups with no members and defined routing policies that are never used for any peer.[5]

With 69% of BGP configurations, 85% of interfaces, 88% of routing policies, and 57% of route attribute match lists being completely untested, this test suite is clearly undertesting the network. This leaves the network vulnerable to bugs in untested configurations elements. Prior to NetCov, it was not possible for network engineers to get any insight into the quality of their test suite. It was also not possible for them to get help toward systematically improving tests. We demonstrate this test suite improvement process next.

6.1.2 Coverage-guided test development

NetCov's feedback enables a test suite development process that enables users to systematically improve coverage, which helps test more critical aspects of the network and prevent outages. This process is iterative. In each iteration the user first identifies specific testing gaps and then creates new tests to target those gaps. We demonstrate the process using three iterations that focus on different types of gaps.

Iteration 1. We saw that routing policy coverage of *NoMartian* test is low (Figure 5) despite that it checks the import policies for all external peers. To investigate, we look at the structure of Internet2 import policies and find that routers have a policy named `SANITY-IN` which is shared by the majority of external neighbors. Figure 4(a) shows this policy with annotated coverage. Each router has an independent copy of this policy, but the copies and the coverage results are identical across routers. Of the five clauses in the policy, the clause `block-martians` starting at line 6,896 is the only clause that is covered. This coverage result confirms that the *NoMartian* test did its job, and more importantly, it revealed a systematic testing gap–the other four classes of forbidden routes are not being tested.

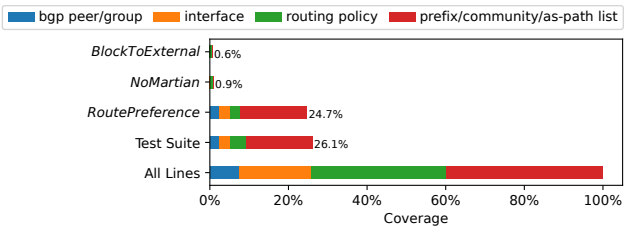


Figure 5: Coverage of the initial test suite broken down to each individual test and configuration type.

Once we know the gap, the solution suggests itself. We added a new test, *SanityIn*, to enforce that the other four classes of received BGP messages should be rejected. After adding this test, we used NetCov to confirm that this testing gap had been addressed. Routing policy coverage was improved by 0.6% and all five terms of `SANITY-IN` were covered by the new test suite. The quantitative improvement is low because `SANITY-IN` is just one of many policies in the network. With feedback from NetCov, network engineers can identify testing gaps in other routing policies and add more tests in a similar way.[6]

Iteration 2. BGP peer configuration coverage of *RoutePreference* test in Figure 5 is surprisingly low, given that all external BGP peers are supposed to be checked. Upon further investigation we find that the uncovered peers have permitted prefix-lists that do not overlap with other peers' lists, which left these peers untested.

We added a new test, *PeerSpecificRoute*, to check that BGP announcements received from external peers should be accepted if their prefixes is in a peer-specific prefix list. This test improved BGP peer coverage from 32% to 46%. The rest of untested BGP peers are either not allowed to send BGP routes to Internet2 or is intended for other internal use, such as monitoring and management. This test also improved prefix-list coverage from 45% to 63%. The remaining of untested prefix-lists are mostly (30% out of 37%) ones that are defined by never referenced.

Iteration 3. The low coverage of interface configuration in Figure 5 reveals another testing gap. *RoutePreference* is the only test in the initial test suite that checks interface configurations, and it only considers one category of interfaces–ones that are used to establish the tested BGP edges. Many other

[5] Per best practices, these lines should be deleted. Or, at a minimum, they should be tested lest someone start using an unused, erroneous policy. When it comes to testing, such lines can never be exercised by data plane tests, though control plane tests may be written for them.

[6] Automatic test generation based on coverage feedback will further help engineers. We will investigate this in the future.

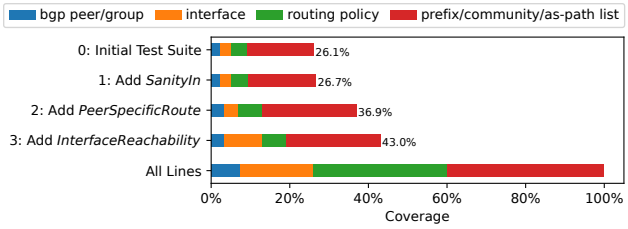


Figure 6: Coverage improvement with test suite iterations.

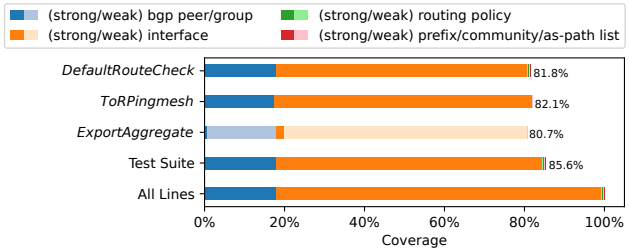


Figure 7: Coverage of synthetic datacenter network for different tests and types of configuration elements.

interfaces remain untested, including but not limited to ones that associate with untested BGP edges and other routing protocols, and the ones that are unused.

We added a new PingMesh-style [18] test, *InterfaceReachablility*, to check that the IPv4 addresses assigned to interfaces should be reachable from each router in the network. This test increased interface coverage from 15% to 53%. The rest of untested interfaces do not have IPv4 addresses assigned.

Figure 6 summarizes the coverage improvement for the three iterations of test improvement in our study. After only three iterations, the overall coverage was improved from 26% to 43%. This final coverage number is far from perfect, but our goal was not to develop the ideal test suite for Internet2; we wanted to demonstrate how coverage information helps develop new tests. Networks are complex, and we should not expect to get the job done with 6 tests. Many more tests are likely needed. With NetCov, network engineers now have a tool to develop new tests that meaningfully improve coverage.

6.2 Case study II: Datacenter networks

We study the coverage for data center networks which have a different topology and routing design. We create synthetic fat-tree [2] networks with routers across three tiers. The leaf routers at the bottom tier connect to hosts. Aggregation routers at the middle tier connect to leaf routers in a pod and to spine routers at the top tier. The spine routers connect to the wide area network (WAN). The WAN is not part of the tested network. Each leaf router is assigned a /24 prefix which is advertised inside the data center through eBGP. Spine routers receive a default route (prefix 0.0.0.0/0) from WAN via eBGP and propagate it to lower tiers. At each spine router, the entire address space of the network is summarized into a /8 prefix and is announced to WAN. Multipath routing (ECMP) is enabled with maximum number of paths set to 4. Routing policies are only configured at spine routers to white-list the default route received from WAN peers. We synthesize the configurations of these networks in Cisco IOS format.

We study a test suite of three tests inspired in prior works on data center network validation [18, 23].

- *DefaultRouteCheck*: ensure that each router has the default route.
- *ToRPingmesh*: ensure that each leaf router's assigned subnet is reachable from all other leaf routers.
- *ExportAggregate*: ensure that each spine router exports the aggregate route to WAN.

Figure 7 shows the coverage result when the network has a total of 80 routers. Given the uniformity of the network and the test suite, coverage results are similar for other network sizes. The total coverage of individual tests is 81.5%, 82.1% and 80.7% respectively, and the three tests together cover 85.3% of configuration lines. We find that these tests cover largely the same configuration elements—interfaces and BGP peerings between the data center routers—despite checking for seemingly different network behaviors. This result indicates that test development without coverage feedback can be ineffective in terms of covering the testing gaps.

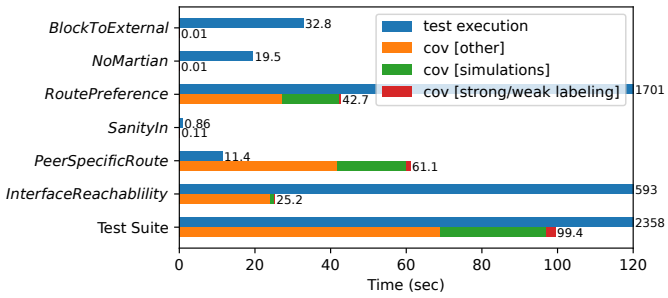
The coverage of *ExportAggregate* shows a large proportion of weak coverage. This is because a spine router has routes to all leaf routers, so that all leaf subnets contribute to the tested aggregate route, albeit weakly. Separating out weak coverage here avoids false negatives of testing gaps—the aggregate routes would be there even if some of the BGP peering or interfaces are misconfigured, therefore testing the aggregate routes provides a weaker endorsement for the covered BGP peerings and interfaces to be bug-free.

By looking at uncovered configuration lines reported by NetCov, we learn that most correspond to host-facing interfaces on leaf routers. Adding tests that target those interfaces improves this test suite and eliminate testing gaps. We omit results of this iteration.

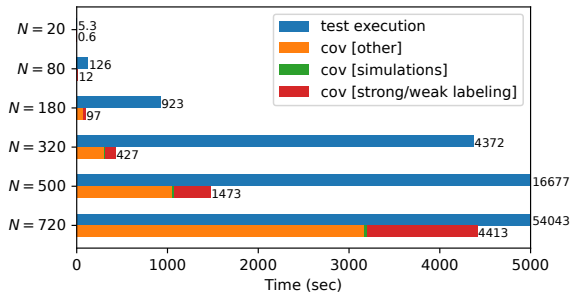
7 Performance Evaluation

We benchmark the performance of NetCov on both types of networks we studied above. Our test machine has two Intel Xeon CPUs (16 core each, 3.1 Ghz), 384 GiB of DRAM, and runs Ubuntu 18.04.

Figure 8(a) shows the time to compute coverage for each test in §6.1 and for the full test suite. It breaks out the time spent on simulations and strong/weak labeling, and, for reference, also shows the test execution time. We see that coverage



(a) Internet2.



(b) Fat-tree networks.

Figure 8: Time to compute coverage.

computation is reasonably fast. The full test suite takes only 99.4 seconds. In comparison, the test execution takes 2,358 seconds. The total coverage computation time is less than the sum for individual tests because facts tested by multiple tests are tracked only once. The graph also shows that simulations and strong/weak labeling are a minority component, which means that most of the time is spent on walking the IFG and doing lookups in stable state for backward inference.

Figure 8(b) shows test execution and coverage computation time for the test suite in §6.2, as a function of the data center network size. Coverage computation takes 4,413 *sec* on the largest network, which has 2,040,624 RIB entries. This time is less than 9% of the time to execute the test suite. While substantial, we deem it acceptable in practice. Configuration coverage analysis can be run in the background, as code coverage is often run. NetCov does not slow down test execution, which is on the critical path to finding configuration errors and updating the network.

However, time to compute coverage increases rapidly with network size. This is because the number of RIB entries grows quadratically and so does the number of vertices in the IFG. We find that the average time to materialize an IFG node does not change substantially because all computation is local to the node. The scaling trends suggest that to scale NetCov to much larger networks, we need a concurrent implementation of IFG materialization. Our current implementation is single-threaded (as Python interpreter is single-threaded).

8 Comparison to Data Plane Coverage

We demonstrate the unique value of control plane coverage by comparing it to data plane coverage. Following Yardstick [47], we quantify data plane coverage as the proportion of main RIB (forwarding) rules exercised. Figure 9 shows the comparison for different cases. Figure 9(a) shows the comparison for Internet2 for all tests in §6.1 and a hypothetical data plane test that inspects all main RIB rules. Figure 9(b) shows the comparison for fat-tree tests in §6.2.

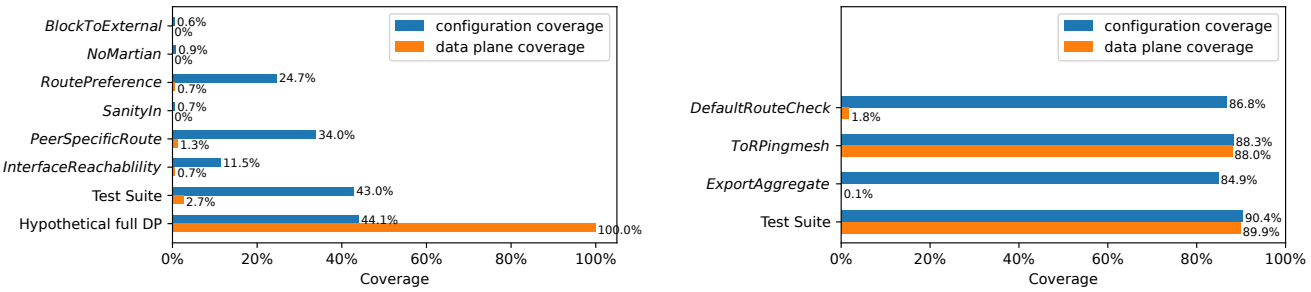
Besides the obvious advantage that only control plane coverage can support control plane tests—the graphs show 0% data plane coverage for these tests—there are two main advantages to using control plane coverage to guide network test development. First, it reveals testing gaps that can not be revealed by data plane coverage. Tests with high data plane coverage do not necessarily have high control plane coverage, as we can see in the last row of Figure 9(a). Covering 100% of the data plane state covered only 41% of the configuration. If the engineers were to improve the test quality under the guidance of only data plane coverage, they would not know that 59% of the configurations remain untested. The reason of this disagreement is that some configuration lines are only exercised under specific environments (failures, routing messages). For instance, list-filtered route policies apply on BGP messages within a specific range, and will only be exercised when such messages appear in the environment.

Second, testing more data plane state can sometimes be redundant in covering configurations, when the tests hit the same configuration elements. For example, the *DefaultRouteCheck* test in Figure 9(b) has only 1.8% data plane coverage because it only tests default routes, which is a small fraction of all main RIB routes. However, because correct propagation of default routes incorporates many BGP peerings and interfaces in the network, this test has extensive configuration coverage (87%). The *ToRPingmesh* test covers much more data plane state (88%), but adding it atop *DefaultRouteCheck* has little value because this state is derived from almost the same set of configurations lines. We do not necessarily imply that engineers should drop one of these tests, as there may be other reasons to keep both. Our observations are about their value toward configuration coverage.

9 Related Work

Our work builds on top of four lines of research.

Code coverage. We borrow from the software domain the idea of using code coverage to reveal testing gaps, quantify test suite quality, and help engineers improve their test suites [4, 15, 20]. Our coverage analysis techniques, however, are specialized to the operation of network configurations.



(a) Internet2.

(b) Fat-tree with k=10.

Figure 9: Comparing control plane and data plane coverage.

Data plane coverage. Yardstick introduced data plane coverage metrics [47] that quantify the proportion of data plane elements such as forwarding rules and paths that are exercised by network tests. Configuration coverage goes further and maps tested data plane components to configuration elements that contribute to them. It provides more direct feedback because network engineers author configurations, not data plane state, and it supports testing of configuration elements that are not exercised by the current data plane state.

Network testing and verification. A range of tools can analyse properties of network data and control planes [7, 12, 14, 18, 19, 23, 25, 26, 48, 49]. NetCov borrows ideas from verification tools to concisely model the network, *e.g.*, focusing on stable state and routing protocol instances [7, 14]. However, NetCov target a different problem—reveal what is tested vs enabling testing of new properties–and uses different techniques.

Network provenance. Provenance systems can track causal dependencies of events in distributed systems. Provenance systems like ExSPAN [52] materialize provenance graphs by tracing system execution in forward direction. Negative provenance systems can reason about missing events [46] and materialize provenance graphs lazily using backward inference. NetCov too uses a graph-based model. However, it is unique in terms of accommodating network configuration into a provenance model, and this model, tailored to the stable state assumption, is more succinct. Further, it combines backward and forward inference to overcome the limitations of using only one type of inference.

Software configuration testing. As for networks, configuration testing is an important problem for software systems as well. Sun et al. developed a system that can link software tests to exercised configuration parameters [38]. They exploit dependence on configuration settings being explicit, observable via read/write operations that use standard get/set APIs. NetCov targets a setting where the dependencies are implicit and non-local. Routers read the entire configuration file, and their forwarding behavior depends on that file and information received from neighbors who in turn act based on their configuration files and their neighbors. That led us to develop a different approach to tracking configuration dependencies. We will investigate in the future if our approach can be extended to software systems where dependence between tested runtime behavior and configuration is not explicit.

10 Summary

NetCov reveals which configuration lines are tested by a suite of network tests. It uses an information flow model based on control plane semantics to track which configuration lines contribute to tested data plane state. It accounts for non-local and non-deterministic contributions, and for performance, it discovers the graph lazily. Our experiments showed that NetCov successfully reveals coverage gaps for real-world networks and test suites, and these tests can have surprisingly low coverage, e.g., 26% of configuration lines for Internet2. They also showed how its feedback helps improve coverage.

Acknowledgments

We thank the NSDI'23 reviewers and our shepherd, Aditya Akella, for feedback on the earlier version of this paper. This work was supported in part by NSF grant CNS-2007073 and Cisco Systems.

Ethical considerations

This work does not raise any ethical issues.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *Proceedings of NSDI 20*, pages 201–219. USENIX Association, 2020.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network

architecture. In *Proceedings of SIGCOMM '08*, page 63–74. ACM, 2008.

[3] Mae Anderson. Time Warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, 2014.

[4] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[5] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. Reachability analysis for AWS-based networks. In *International Conference on Computer Aided Verification*, pages 231–241. Springer, 2019.

[6] Batfish: Network configuration analysis tool. https://github.com/batfish/batfish.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of SIGCOMM '17*, pages 155–168. ACM, 2017.

[8] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.

[9] Larry Brader, Howie Hilliker, and Alan Wills. *Testing for Continuous Delivery with Visual Studio 2012*. Microsoft, 2013.

[10] Cisco Systems, Inc. Configure protocol redistribution for routers. https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/8606-redist.html.

[11] Codecov. Codecov: The leading code coverage solution. https://about.codecov.io/, 2021.

[12] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of NSDI 15*, pages 469–483. USENIX Association, 2015.

[13] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on networking*, 9(6):681–692, 2001.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of SIGCOMM '16*, pages 300–313. ACM, 2016.

[15] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 302–313, 2013.

[16] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions On Networking*, 10(2):232–243, 2002.

[17] GNU Guix. lcov–code coverage tool that enhances gnu gcov. https://guix.gnu.org/en/packages/lcov-1.15/.

[18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of SIGCOMM '15*, page 139–152. ACM, 2015.

[19] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *Proceedings of NSDI 17*, pages 735–749. USENIX Association, 2017.

[20] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.

[21] Istio. Diagnose your configuration with istioctl analyze. https://istio.io/latest/docs/ops/diagnostic-tools/istioctl-analyze/.

[22] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963. ACM, 2019.

[23] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *Proceedings of SIGCOMM '19*, pages 200–213. ACM, 2019.

[24] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[25] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of NSDI 12*, pages 113–126. USENIX Association, 2012.

[26] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of NSDI 13*, pages 15–27. USENIX Association, 2013.

[27] Nuno P Lopes and Andrey Rybalchenko. Fast BGP simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–408. Springer, 2019.

[28] Matthew Luckie, Bradley Huffaker, Amogh Dhamdhere, Vasileios Giotsas, and KC Claffy. AS relationships, customer cones, and validation. In *Proceedings of IMC '13*, pages 243–256, 2013.

[29] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *Proceedings of SIGCOMM '11*, pages 290–301. ACM, 2011.

[30] Netcov: Network configuration coverage tool. https://github.com/UWNetworksLab/netcov.

[31] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *Proceedings of NSDI 20*, pages 953–967. USENIX Association, 2020.

[32] Bruno Quoitin and Steve Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE network*, 19(6):12–19, 2005.

[33] Steve Ragan. BGP errors are to blame for Monday's Twitter outage, not DDoS attacks. https://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html, 2016.

[34] Deon Roberts. It's been a week and customers are still mad at BB&T. https://www.charlotteobserver.com/news/business/banking/article202616124.html, 2018.

[35] Deon Roberts. Facebook says its outage was caused by a cascade of errors. https://www.nytimes.com/2021/10/05/technology/facebook-outage-cause.html, 2021.

[36] Joao Luis Sobrinho. Network routing with path vector protocols: Theory and applications. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60, 2003.

[37] Fabio Somenzi. CUDD: CU decision diagram package release 2.5.0. *University of Colorado at Boulder*, 2012.

[38] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *Proceedings of OSDI'20*. USENIX Association, 2020.

[39] Yevgeniy Sverdlik. United says it outage resolved, dozen flights canceled monday. https://www.datacenterknowledge.com/archives/2017/01/23/united-says-it-outage-resolved-dozen-flights-canceled-monday, 2017.

[40] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of SIGCOMM '19*, page 214–226. ACM, 2019.

[41] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[42] Route Views. University of Oregon Route Views project. http://www.routeviews.org/routeviews/, 1997.

[43] Rosemary Wang. Testing HashiCorp Terraform. https://www.hashicorp.com/blog/testing-hashicorp-terraform.

[44] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *Proceedings of OOPSLA 2016*, pages 765–780. ACM, 2016.

[45] Zach Whittaker. T-mobile hit by phone calling, text message outage. https://techcrunch.com/2020/06/15/t-mobile-calling-outage/, 2020.

[46] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. *ACM SIGCOMM Computer Communication Review*, 44(4):383–394, 2014.

[47] Xieyang Xu, Ryan Beckett, Karthick Jayaraman, Ratul Mahajan, and David Walker. Test coverage metrics for the network. In *Proceedings of SIGCOMM '21*, page 775–787. ACM, 2021.

[48] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Trans. Netw.*, 24(2):887–900, April 2016.

[49] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *Proceedings of SIGCOMM '20*, page 599–614. ACM, 2020.

[50] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252, 2012.

[51] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of NSDI 14*, pages 87–99. USENIX Association, 2014.

[52] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of SIGMOD '10*, pages 615–626. ACM, 2010.

Norma: Towards Practical Network Load Testing

Yanqing Chen[†,‡], Bingchuan Tian[‡], Chen Tian[†], Li Dai[‡], Yu Zhou[‡], Mengjing Ma[‡], Ming Tang[‡], Hao Zheng[†], Zhewen Yang[†], Guihai Chen[†], Dennis Cai[‡], and Ennan Zhai[‡]

[†]*State Key Laboratory for Novel Software Technology, Nanjing University* [‡]*Alibaba Group*

Abstract

Network load tester is important to daily network operation. Motivated by our experience with a major cloud provider, a practical load tester should satisfy two important requirements: **(R1)** stateful protocol customization, and **(R2)** real network traffic emulation (including high-throughput traffic generation and precise rate control). Despite the success of recent load testers, we found they fail to meet both above requirements. This paper presents Norma, a practical network load tester built upon programmable switch ASICs. To achieve the above requirements, Norma addresses three challenges: (1) modeling stateful protocols on the pipelined architecture of the ASIC, (2) generating replying packets with customized payload for stateful protocols, and (3) controlling mimicked traffic in a precise way. Specifically, first, Norma introduces a stateful protocol abstraction that allows us to program the logic of the state machine (*e.g.*, control flow and memory access) on the programmable switch ASIC. Second, Norma proposes a novel multi-queue structure to generate replying packets and customize the payload of packets. Third and finally, Norma coordinates meters and registers to construct a multi-stage rate control mechanism capable of offering precise rate and burst control. Norma has been used to test the performance of our production network devices for over two years and detected tens of performance issues. Norma can generate up to 3 Tbps TCP traffic and 1 Tbps HTTP traffic.

1 Introduction

Understanding whether the network meets expected performance is essential to today's cloud providers, especially for performance-sensitive services such as live streaming and edge cloud games [30, 35, 57]. For example, in edge cloud games, the players complain about their unsmooth feelings if the network latency reaches 50 ms, and cannot play the games when the latency exceeds 100 ms [54].

Network load tester is one of the most important testing tools that checks the performance of network devices by proactively generating various testing packets including different protocols, rates, and traffic patterns [7, 13]. A network load tester could be used by the operator to test the performance of devices, debugging the root causes of packet loss. In a typical network load testing scenario, the tester generates user-defined traffic and sends it to the Device Under Test (DUT). After receiving the testing packets, the DUT processes the traffic and forwards it back to the tester for further processing, such as dropping or replying to the incoming packets. Based on the analysis of the outgoing and incoming traffic, the tester can evaluate the performance of the DUT in multiple aspects, including throughput, latency, and packet loss.

As a major cloud provider, we also deploy load testers in production networks to test pre-online network devices and functions. Load testers have become indispensable for daily network operation tasks, such as performance monitoring, failure troubleshooting, and stress testing. Building a *practical* load tester that works for large-scale cloud networks should satisfy the following important requirements from our network operators.

- **(R1) Stateful protocol customization.** Besides switches and routers which work in a stateless way, cloud networks also have complex and stateful network functions, such as stateful packet filters and L4/L7 load balancers. The protocols used by these network functions might be stateful (*e.g.*, HTTP) or self-defined by the cloud provider. To provide the all-around testing capability, a practical load tester should be able to generate packets with not only stateless protocols (*e.g.*, UDP) but also stateful (*e.g.*, TCP) and customized (*e.g.*, private tunnel protocols) protocols.
- **(R2) Real traffic emulation.** As the scale and single-port bandwidth of cloud networks grow fast, a practical load tester should be able to mimic real, cloud-grade traffic in a cost-effective way: (1) it can generate Tbps-level traffic, and (2) it can create and send precise rate packets with customized payload and burst patterns.

A number of previous efforts have focused on network load testing [7, 9, 13, 18, 24, 31, 32, 47, 53, 55, 59]. While these state-of-the-art systems can work well in principle, in reality in our situation, they fail to simultaneously satisfy the above two requirements (see Table 1). Specifically, software load testers [9, 18, 24, 31, 32, 47] and FPGA-based load testers [60] are unable to generate Tbps-level traffic or control rate precisely (*i.e.*, fail **R2**). On the other hand, hardware load testers (*e.g.*, Keysight [7] and Spirent [13]) can only generate and emulate fixed types of protocols (*i.e.*, unable to support the customized protocols **R1**). Recently, researchers have developed load testers based on programmable switch ASICs [53, 55, 59], which are capable of sending Tbps-level traffic and are customizable. While these pioneer systems have shown the potential to partially solve the above problems, they cannot customize stateful protocols or provide precise rate control, *i.e.*, partially failing **R1** or **R2**.

We, therefore, decided to build a practical load tester to

satisfy our operators' requirements for their daily usage.

Our approach: Norma. This paper presents Norma, a high-performance network load tester, based on the RMT-based[1] programmable switch ASIC. The key idea of Norma is to execute load testing based on template packets derived from tested protocols, which enables Norma to achieve **R1** and **R2** simultaneously. First, template packets continuously loop in the pipeline and can be conditionally replicated to generate testing packets of both stateless and stateful protocols. Operators can flexibly customize headers and payload of template packets; thus, Norma can be used to test various cloud network functions. Second, through controlling rates and patterns of replicating template packets, Norma can generate load testing traffic that faithfully mimics realistic traffic. Besides programmable switch ASIC resource limitations [33, 38, 39, 41, 50–52, 56], building Norma nevertheless requires us to address the following challenges.

Challenge 1: First, RMT-based programmable switch ASICs are unfriendly to modeling stateful protocols (*e.g.*, TCP and HTTP), since the ASIC architecture is implemented as a pipeline that processes packets in a sequential way and only supports accessing states once per packet. However, most stateful protocols need to read and write a state multiple times. This, therefore, makes it difficult to model or customize stateful protocol behaviors on current programmable switch ASICs. On the other hand, testing the performance of stateful protocols (*e.g.*, stateful load balancer, DDoS defense, and ACL) is crucial, which accounts for the majority of our load testing requirements and tasks. To the best of our knowledge, none of the prior work solved this problem. Existing load testers based on programmable switch ASICs like HyperTester [59] can only support stateless protocol customization. To address this challenge, we introduce a new data structure, named *stateful protocol abstraction*, to enable programming the logic of the state machine (including control flow and memory access) on programmable switch ASICs. We construct a state machine framework via the stateful protocol abstraction. In the framework, packets are looped inside the ASIC, and each round corresponds to a step in the state machine of the emulated stateful protocol. We can use this framework to emulate arbitrarily complex protocols (including both stateful and stateless), as long as the hardware resources of the ASIC are sufficient (§4.1 and §4.2).

Challenge 2: Load testers need to reply according to state machines when receiving packets of stateful protocol from DUT. Replying packets involves packet generation with customized payload as well as header modification, which presents the second challenge. The programmable switch ASIC uses PHV[2] resources to add, delete, and modify packet headers and payload. Due to limited PHV resources, the capability of load testers to generate and modify packets with a large payload and statefully complex headers is inherently constrained. To address this challenge, we propose an efficient multi-queue structure based on registers[3] inside the programmable switch ASIC. In this structure, the stateful packet will enqueue to trigger the corresponding type of template packet to dequeue. In this way, Norma supports generating replying packets with customized payloads for most stateful protocols (§4.3 and §4.4).

Challenge 3: The final challenge is that programmable switch ASICs is hard to offer precise control of the packet rate and burst, thus resulting in unrealistic traffic emulation. The above two control capabilities are important requirements of our daily operation and testing; however, we have not seen any of state of the art systems that can achieve the above goals. The rate control relies on the specific hardware named meter; however, in practice, the speed limit of the hardware meter is coarse-grained, *i.e.*, not all target rates can be precisely achieved, which results in an error in the control of packet rate. In addition, the programmable switch ASICs do not support the generation of traffic bursts with given patterns. To address this challenge, we proposed a multi-stage rate control mechanism based on the coordination of meters and registers in the programmable switch ASIC. The meters provide coarse-grained rate control, which will be further tuned by the follow-up registers in a fine-grained manner. In this way, the special requirements of the tester for rate and burst control can be satisfied with great precision (§5).

Norma has been used to test the performance of pre-online devices that would be deployed in our production network for over two years. For example, we used Norma to test the forwarding capability and ARP learning rate of L2/L3 switches and tested stateful gateways by generating L4/L7 flows. Evaluation results show Norma can generate up to 3 Tbps TCP traffic and 1 Tbps HTTP traffic while maximizing the use of pipeline bandwidth. Experiments also show that Norma can achieve precise rate control and burst capability. The relevant rate error does not exceed 0.01% in the worst cases.

Contributions. We make the following contributions:

- It is new for us to implement the stateful responder into the programmable switch ASIC to support stateful protocols. The pipeline-folded switch ASIC and the queue implemented in P4 are the keys to make it possible.
- We propose high-precision packet rate and burst control method. This provides us the ability to reproduce traffic at accurate rates and desired burst patterns.
- We use Norma to test our pre-online devices. Norma is useful for our network developers and operators to find performance issues and system bottlenecks.

[1] RMT (Reconfigurable Match Tables) is a reconfigurable pipeline-based architecture for programmable switch ASICs. Each pipeline consists of a parser, multiple match-action stages, and a deparser [22, 40].

[2] PHV (Packet Header Vector) stores and transits parsed headers or metadata between neighboring stages. More details can be found in [10, 22].

[3] Registers are memory blocks attached to each stage, whose data can be shared by multiple packets across different ports inside a pipeline [10, 22].

Ethics. This work does not raise any ethical issues.

2 Background & Motivation

This section details our operators' requirements and discusses related work.

2.1 Requirements for Production

Based on the experience of our operators, we summarize the requirements of our network load tester in Table 1.

(1) Supporting protocol customization. In cloud networks, traffic can be carried via non-standard private protocols. These protocols are usually defined and experimentally developed by the cloud providers, *e.g.*, QUIC [42] and Multipath QUIC [28, 58], which provide great extensibility of network functions and can be quickly iterated according to the needs of upper-layer applications. These non-standard protocols and traffic are not supported by commercial hardware network testers.

In particular, the majority of protocols we need to test are stateful. The DUT keeps the state of L4/L7 sessions for stateful protocols. For example, an L4 gateway may perform a TCP relay or SYN proxy, and an L7 load-balancer balances the load of the HTTP traffic according to the HTTP header of the first packet. In these cases, the load tester should be able to emulate the establish, transmission, and release processes of a session, and reply to the incoming packets according to the specification of the protocol.

(2) Emulating realistic traffic. During the development and operation of network devices, our operators need to evaluate the device or optimize configurations by emulating realistic traffic. The volume of mixed traffic flows can be as large as O(1 Tbps), or O(1 Gpps) for small packets. It is essential to emulate the traffic similar to the realistic load. We have observed the case that a DUT works well in the experimental development with simple traffic, but suffers from continuous packet drop after it goes online. It is not acceptable for cloud providers.

To test the DUT with emulated traffic under heavy loads, the tester should support sending traffic at the line rate of DUTs. Besides, the tester needs to emulate various traffic patterns and mixed traffic flows for network operators to determine the optimal configuration like hash function, CPU allocation, and queuing policy of devices. The traffic is expected to be cheap in terms of hardware cost, power consumption, and rack size. Plus, in cloud network testing, the value of the field in the packet header is required to be editable. For example, one may expect the source IP address to be randomly chosen from the prefix 10.0.0.0/16.

In addition, since a network load tester is usually used for network checking, debugging, and troubleshooting, it is required to control the sending rate based on the determined configuration. In other words, the tester should be able to generate the random burst traffic and emulate the failure scenarios precisely. All of the testing data are collected by measuring the incoming and outgoing traffic in multiple dimensions, such as throughput and packet drops. A network load tester is required to support fine-grained bidirectional measurement of the large volume of traffic.

2.2 Related Work

Table 1 shows the comparison between Norma and the state-of-the-art load testers in terms of our production requirements.

Software network testers. Software solutions [9, 18, 24, 31, 32, 47] are highly flexible. The early software network testers [6, 11, 17, 23] are based on the standard Linux IO API which limits the performance and accuracy. There are many works [9, 18, 31, 47] that utilize the IO frameworks such as DPDK [2], Netmap [47], and PF_RING ZC [8] that are working on accelerating packet processing on various CPU architectures. However, the computing bottleneck makes it difficult to apply to 100 Gbps network test scenarios. The state of the art such as MoonGen [31] needs over 14 2.4 GHz cores to generate 64-byte packets at 100 Gbps, corresponding to only one port capability of Norma. In addition, software solutions are not stable when testing complex packet processing due to the indefinite packet processing time [59]. Therefore, they are not scalable and cost-effective in industrial scenarios that require Tbps-level load testing.

Commodity hardware testers. Vendors like Keysight [7] and Spirent [13] provide network infrastructure performance tests using their test suites with hardware-based modules. These commercial hardware testers [7, 13] are able to emulate standard traffic demands by providing rich testing functions. There are also application and security tests that cover the L4 protocols. Benefiting from the specially designed software and hardware, it achieves high throughput and accuracy on packet generation and measurements. Commercial hardware tester uses dedicated ASICs from the vendors to accelerate network traffic generation, which can provide O(1 Tbps) traffic for stateless protocols.

However, the commercial hardware tester is a black box, which makes it hard to adapt to the agile development of self-defined protocols. The vendors are aiming to provide standard tests thus the customizability of user-defined packet structures and protocols is lacking. Besides, they are expensive to deploy in a large-scale system (*e.g.*, $100,000 for a 100 Gbps dual-port packet generation module [59]) which requires a large number of testers.

Programmable hardware testers. To achieve a balance of programmability and performance, some network testers [19, 27, 48] using programmable hardware such as NetFPGA [60] are proposed. These works achieve accurate rate controlling and precise measurement results. However, the NetFPGA-based testers are still expensive to achieve Tbps-level test traffic (*e.g.*, a NetFPGA board costs $5,341 with two 100 GbE interfaces [1], and a programmable switch ASIC with 32 100 GbE interfaces only costs $2160 [5]). And it is non-trivial to develop new functions on FPGA boards.

Table 1: The required properties of a tester listed by our network operators, and the comparison between Norma and prior work.

Requirements	Meaning	Norma	CHT	ST	HT	NetFPGA
Stateful Protocol Support						
Generation	Whether the traffic of stateful protocols (*e.g.*, HTTP) can be generated?	✓	✓	✓	✗	✓
Customization	Whether the stateful protocol can be fully customized by users?	✓	✗	✓	✓	✓
Real Traffic Emulation						
Cheap High-Speed Traffic	Whether O(1 Tbps) traffic can be generated in a cheap way?	✓	✗	✗	✓	✗
Precise Rate Control	Whether the rate of generated traffic is precise?	✓	✓	✗	✗	✓
Precise Burst Control	Whether the traffic can be sent out with customized burst pattern?	✓	✓	✗	✗	✓
Precise Measurement	Whether the traffic features can be precisely measured?	✓	✓	✗	✓	✓

CHT=Commercial Hardware Testers ST=Software Testers HT=HyperTester

Programmable switch ASICs like Intel Tofino [16] provide customizable packet processing logic via programmer-friendly P4 language [21]. HyperTester [53, 55, 59] leverages the recirculate primitive in P4 language and packet replication engine to generate packets. It can generate stateless traffic at the rate of 1.6 Tbps. HyperTester confirms the feasibility to implement a stateless hardware tester via Tofino's programmable switch ASICs and proves the traffic quality via microbenchmarks. However, HyperTester cannot emulate the data plane behavior of the stateful protocol. We cannot use HyperTester to test a stateful L4/L7 gateway, because HyperTester cannot generate and maintain the session as what the TCP/HTTP specification describes. In addition, HyperTester cannot emulate realistic traffic in a high-fidelity way. We analyze the reason and conduct experiments in §9.2. Inspired by HyperTester, IMap [43] uses programmable switch ASICs for network scanning. It is not a network load tester in a general sense.

Stateful packet processing. Many works are providing stateful packet processing to offload networking functions into hardware. FlowBlaze [46], FAST [44], and OpenState [20] define state machine abstraction to describe network functions, while Domino [49], dRMT [26], SDP [34], and Ibanez *et al.* [37] propose customized RMT-based architecture using FPGA to achieve the processing ability of the stateful packet. These works are orthogonal to Norma. Norma focuses on emulating high-throughput stateful traffic to test the performance of the DUT, instead of implementing every detail of stateful protocols. This gives us the chance to implement the state machine on programmable switch ASICs like Tofino. None of the prior work focuses on this.

3 Norma Overview

Norma is a practical cloud network tester for load testing. We use the programmable switch ASIC to leverage its large capability of packet processing and programmability. In this part, we explain the reason for using the pipeline-folded [14, 15, 45] programmable switch ASIC first (§3.1). Then, we introduce the high-level architecture of Norma (§3.2). This architecture illustrates how our testing functions are arranged in the ASIC and work as a practical network tester.

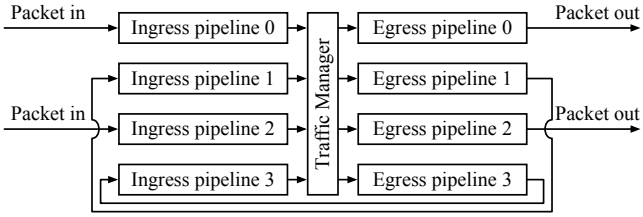


Figure 1: The packet path of pipeline-folded programmable switch ASICs. Half of the pipelines are in loopback mode.

3.1 The Pipeline-Folded Switch ASIC

As shown in Figure 1, the pipeline-folded programmable switch ASIC we use (*i.e.*, BFN-T10-032Q [5]) has 64×100G ports with maximum port bandwidth of 3.2 Tbps. It has four physical pipelines in total, but only two of them are connected to front panel ports (*e.g.*, pipeline 0/2, namely *external pipeline*). The remaining two pipelines are connected to the internal loopback ports, whose egress direction is wired to the ingress direction inside the ASIC (*e.g.*, pipeline 1/3, namely *internal pipeline*).

Norma chooses the pipeline-folded programmable switch ASIC for three reasons. (1) The internal pipelines and external pipelines can be programmed with different P4 programs. By dividing functions such as basic switching, packet editor, stateful responder, *etc.* into the above two groups, Norma can support all of them simultaneously with the limited hardware resources inside the ASIC. (2) Besides the recirculation capability provided by P4 primitives, internal pipelines provide 3.2 Tbps extra loopback bandwidths. Therefore, high-throughput traffic generation can be achieved without affecting the front-panel port throughput. (3) The folded pipelines double the stages we can use. That means we can implement more complex processing logic than the unfolded one, which is the fundamentals of stateful packet processing. The RMT-based programmable switch ASIC guarantees that these additional stages do not affect the processing rate of the traffic and only introduce negligible latency.

3.2 Norma's Architecture

Norma takes advantage of the pipeline-folded programmable switch ASIC and arranges all required functions in the architecture shown in Figure 2. The input and output represent the front-panel ports of the switch. The basic switching logic (omitted in figures) and measurement functions are imple-

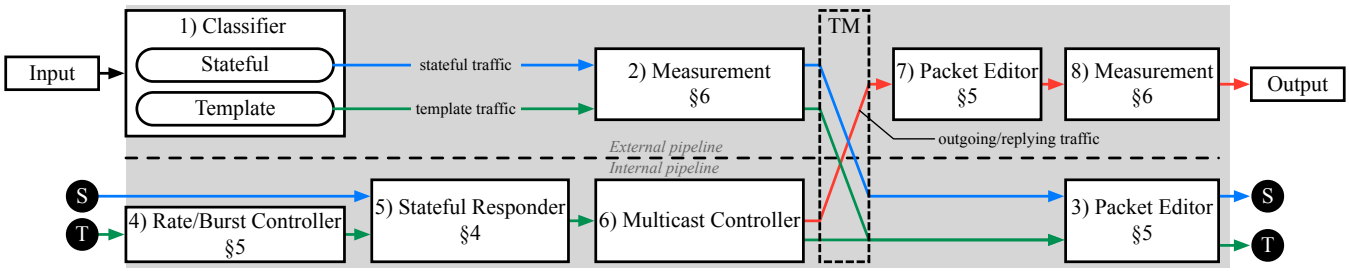


Figure 2: The function models and workflow of Norma. Different colors of arrow lines represent different traffic types. Nodes S and T are loopback ports.

mented in external pipelines, while other functions are implemented in internal pipelines. The **S** and **T** nodes represent two loopback ports in internal pipelines, allowing packets to travel from the egress back to the ingress. The traffic manager (TM) is responsible for packet replication and forwarding.

Traffic. Norma needs to classify incoming traffic to decide which function to enable according to the user's test task. We detail three kinds of traffic shown in Figure 2 as follows.

- **Stateful traffic** includes incoming packets of stateful protocols such as TCP and HTTP, represented as blue lines. Once received, these packets will be preprocessed by the packet editor (*e.g.*, updating the TCP sequence number) and then handled by the stateful responder, which triggers replying traffic.
- **Template traffic** includes template packets sent from the control plane, represented as green lines. Control plane programs construct template packets according to the user's test task and send them to the ASIC via PCIe. Then Norma keeps these packets looping all the time in loopback ports of internal pipelines. If a template packet is marked by the multicast controller, it will be replicated by the packet replication engine (PRE) in the ASIC's TM module. Then the replicated packet will be forwarded to the DUT as outgoing traffic via external egress pipelines. In this way, Norma can generate line rate traffic on the data plane, though there is no memory for the ASIC to store packets. The looping template packets determine what kind of traffic can be generated.
- **Outgoing/Replying traffic** is represented as red lines. The two traffic types can be regarded as the same because they go through the same paths. The replying traffic is triggered by the stateful responder, while the outgoing traffic is not.

Modules and workflow. We now show the function modules of Norma in turn by the following workflow.

1) **Classifier**. The traffic is classified by the classifier first. Norma mainly focuses on stateful traffic and template traffic. Other traffic is also classified, but omitted in Figure 2.
2) **Measurement**. All traffic enters the measurement module and is measured according to the user's measurement rules. Measurement functions are described in §6.
3) **Packet Editor**. Stateful traffic and template traffic are forwarded to internal egress pipelines. The packet editor preprocesses the stateful traffic or modifies template packet fields. The packet modification function is introduced in §5.
4) **Rate/Burst Controller**. After looping back to internal ingress pipelines, the rate/burst controller marks template packets to control the rate and burst pattern of the generated traffic. This part is detailed in §5.
5) **Stateful Responder**. Norma uses the extended finite-state machine (EFSM) [25] to abstract the stateful protocol. The stateful responder triggers the EFSM according to the input stateful traffic. The replying traffic is generated with the help of the template traffic. This part is detailed in §4.
6) **Multicast Controller**. The multicast controller marks the template traffic and forwards it to the internal egress pipeline it comes from to complete the high-speed looping of the template traffic. In addition, the marked template packets are replicated to the target output port through TM.
7) **Packet Editor**. Outgoing traffic needs to go through the packet editor on the external egress pipeline one more time. The supported actions of these two packet editors are different for sophisticated traffic generation capabilities.
8) **Measurement**. Finally, all outgoing traffic enters the measurement module in the egress direction.

4 Emulating Stateful Protocol

The essence of emulating a protocol is to run the processing program on the programmable switch ASIC. Although, it is not easy to port programs that originally run on CPUs to the ASIC. The protocol implemented on Norma for testing the DUT can be reduced to a human-descriptive sequence of packet interactions, as long as the DUT does not perceive the differences. Therefore, our high-level idea is to convert such a program into a state machine and write it into the match-action table.

In this section, we first introduce the EFSM [25] abstraction of stateful responder, which helps us establish a general stateful protocol programming pattern for Norma (§4.1). Then we take the HTTP protocol as an example to show the implementation details of the stateful responder in three steps: executing

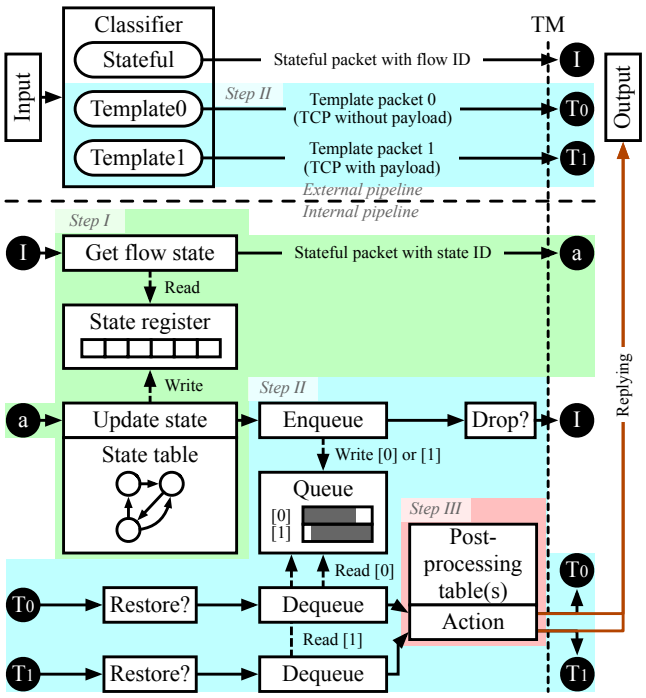


Figure 3: A detailed view of the components of the stateful responder. The classifier is not included in the stateful responder but used to describe where the inputs come from.

the state machine (§4.2), generating replying traffic (§4.3), and postprocessing of the replying traffic (§4.4). The brief packet path and table placement are shown in Figure 3.

4.1 The Stateful Protocol Abstraction

To handle stateful protocols, we need to program the processing logic of the protocol in the programmable switch ASIC. The main process is to generate replying packets according to the flow state and stateful packets, and the abstraction of this process can be represented by the EFSM.

EFSM abstraction of Norma. An EFSM in Norma is defined as a 7-tuple $M = (I, O, S, D, F, U, T)$. S is a set of flow states. I is the current flow state. O is the next flow state. D is a set of variables, such as packet fields, metadata, and registers. F is a set of enabling functions that trigger transitions based on the variables ($f_i : D \to \{0,1\}$). U is a set of update functions that update variables ($u_i : D \to D$). T is a transition relation ($T : S \times F \times I \to S \times U \times O$). Table 2 shows the state table of the EFSM that handles HTTP GET requests. The conditions are the enabling functions in set F, and the actions are update functions in set U.

Hardware bases and limitations. The EFSM abstraction provides an interface for users to customize stateful protocols. Specifically, users need to construct the following three parts in Norma to realize the EFSM. The first part is variables D and flow states S. The parser and deparser provide the ability to locate packet fields and metadata. Because programmable switch ASICs natively support registers, the flow states are saved in registers and updated by register actions [14].

Table 2: A part of the simplified HTTP state table.

Curr S	Condition	Next S	Action
0	RST	0	`ig_md.skip_mc = 1;`
0	SYN	2	`hdr.tcp.flags = R;`
0	Unknown	1	`hdr.tcp.flags = R;`
1	SYN	2	`hdr.tcp.flags = S|A;`
1	Unknown	1	`ig_md.skip_mc = 1;`
2	RST	0	`ig_md.skip_mc = 1;`
2	HTTP GET	3	`hdr.payload = 0x48···`
2	Unknown	1	`hdr.tcp.flags = R;`
		...	

The second part is conditions F. The conditions can be implemented as keys in the match-action table. State transitions and actions are triggered by matching the variables like the current flow state and the input stateful packet. However, it should be noted that the matching ability of the key is limited. The relationship between keys in a table can only be a logical AND relationship, so complex matching rules need to be expressed using multiple entries.

The final part is actions U. The actions modify and send the template packets to reply to the stateful packet. It should be noted that the implementation of actions is limited by the resource constraints of the programmable switch ASIC. Excessive register action and table execution may exceed the number of stages in the switch. And it is hard to implement actions that require iteration, such as sorting headers in a packet. A possible solution is splitting the action that requires iteration into multiple steps. The stateful packet loops in the internal pipeline through node I in the internal pipeline in Figure 3 and executes these steps. But this solution reduces the throughput of the flow processing.

4.2 Executing State Machine

Implementing the EFSM on programmable switch ASICs is challenging due to limitations on registers, and table actions. Next, we will introduce the details of the state machine implementation, as shown in Figure 3 Step I.

State register. Norma stores the state of each flow in a register array. Initially, a flow ID is assigned by the classifier to each flow and used as an index to access registers. Then, the stateful packet is forwarded to the internal pipeline with its flow ID for indexing its state register. The flow state registers store current state ID in Table 2 and other flow information such as TCP sequence number, which can be read by the template packet directly for generating the replying packet.

A side benefit of the flow ID is that it releases the fields of 5-tuples and MAC addresses in the stateful packet. When we generate the replying packet, these fields can be retrieved via a post-processing table with flow IDs. Therefore, we can write the flow ID and other bridged metadata[4] into these fields to avoid additional bandwidth consumption when the packet

[4]Bridged metadata is a temporary header carrying the data calculated in current and previous pipelines to the next pipeline [14].

carries the metadata looping from egress pipelines to ingress pipelines.

State updating. As described in §4.1, the state table stores the conditions for state transition. For the stateful packet, it first reads its state ID from the state registers. Then, it gets the next state ID and action ID from the state table. And finally, the stateful packet writes a new state ID back into the register. However, this process cannot be done simply because of the register access restriction that a register cannot be accessed more than once in one pass (a packet goes through the ingress pipeline and egress pipeline). Reading and updating are two accesses that cannot be merged into one due to complex dependencies. To break this limitation, our idea is to let the stateful packet do another pass. In the first pass, the stateful packet reads the state register to get the state ID. In the second pass, the stateful packet gets the new state ID through the state table and then writes back to the state register. Note that this design can make state register updating non-atomic, further discussed in Appendix A.

State machine bypass. Some protocols can be implemented with state machine bypass. Taking the SYN flood test as an example, the tester receives a SYN packet and replies with an RST packet. The SYN packet can be preprocessed by the packet editor to obtain the action ID. Then, the packet skips Step I in Figure 3 and directly passes the relevant packet information to generate replying packets. For the stateless traffic generation like UDP and ARP reply, the entire stateful responder can be skipped.

4.3 Generating Replying Packets

Norma generates replying packets by replicating the template packet looping in the internal pipeline, as shown in Figure 3 Step II. Take the HTTP GET test task as an example. Norma needs to generate a PUSH packet when receiving an ACK packet. The PUSH packet has a 1460-byte payload, and the ACK packet does not. If the programmable switch ASIC can add or delete the payload, we can generate these two kinds of packets by modifying the input stateful packet. However, PHV resources limit the total length of packet headers that can be parsed. The 1460-byte payload cannot be completely stored in the PHV. So we have to prepare ACK template traffic and PUSH template traffic with 1460-byte payload separately to generate corresponding traffic. Since the stateful packets and template packets do not share packet fields and metadata, Norma needs to transfer information from the stateful packets to the template packets, such as the sequence number and action ID. Then in post-processing tables, Norma can modify the template packets according to the transferred information to generate required replying packets.

Next, we will detail the approach to transferring information across different packets.

A strawman solution. A straightforward way to transfer information across packets is using a queue. The incoming stateful packet pushes necessary information into the queue, and then the template packet pops the information to its corresponding fields or metadata. The register array in programmable switch ASICs can be used to implement the queue. Besides the register array for the elements, the queue needs three more registers to maintain the data structure, one for the head pointer, one for the tail pointer, and one for the queue length. Compared with the common queue data structure, the one in programmable switch ASICs has two critical limitations.

First, the template packet cannot decide whether to dequeue according to the head element. To dequeue according to the head element, the tester must first check whether the queue length is zero, then read the queue element via the head pointer, and finally decide whether to decrease the queue length. In this way, the queue length register is accessed twice in one packet path, which is not allowed by programmable switch ASICs. Since template packets will inevitably take out information from the queue, the queue cannot be shared by multiple types of template packets.

Second, there is no way to prevent the queue from overflowing. Since the packet paths of the stateful packet and the template packet are parallel, to ensure parallel safety, the instructions for dequeuing and enqueuing must be executed in the following order. For dequeuing, reading elements must occur after decreasing the queue length; and for enqueuing, writing elements must occur before increasing the queue length. To ensure that the queue does not overflow, the queue length is compared with its capacity to get the enqueuing permission first. Then the stateful packet writes its information to the queue. And finally, the queue length increases. In this way, the tester accesses the queue length register twice. So there is no method to check the queue length before enqueuing.

In Norma, the consumer of the queue is the template packets, and the producer of the queue is the stateful packets. Template packets poll from the queue to generate replying packets. For example, the PUSH template packets poll the queue for ACK packets. Therefore we must ensure that the polling rate is higher than the arrival rate of the stateful packets to avoid queue overflowing.

Multi-queue for multiple template traffic types. Because of the limitations of the queue, only the stateful packet can choose what kind of template packets to reply to. It is straightforward to allocate a queue for each type of template packets. As shown in Figure 3 Step II, the stateful packet enqueues using queue ID which is obtained from the state table and then triggers the corresponding type of template packet to dequeue. Compared to the strawman solution, there are two changes to the multi-queue data structure. The first change is that three register arrays are used as head pointers, tail pointers, and queue lengths. Each queue ID identifies an element in these register arrays. The second change is that the index of the queue elements needs to be re-planned. A typical method is using some lower bits of the pointer to represent the queue ID, and the remaining bits to represent the element offset in the queue. For example, one register array with a capacity of

256 is used as 16 queues. The lower four bits are the queue ID and the higher four bits are the offset.

Modifying payload. For some test cases where the complete or partial payload of the template packet is capable to be parsed in the PHV, we can treat this payload as a normal packet header. First, the parser needs to parse the payload as a header according to the *total length* in the IP header. Before entering the post-processing table, the previously added payload header must be set to invalid to avoid adding payloads repeatedly. Second, the payload header needs to participate in the calculation of the TCP checksum, which should be implemented in both the parser and the deparser. In this way, Norma can add, delete, and modify the payload without queues.

4.4 Post-Processing

As shown in Figure 3 Step III, post-processing tables are used to modify the template packet and finally generate the replying packet. First, the stateful packet enqueues the flow ID and action ID. Then, the template packet obtains the action ID from the queue and executes state actions according to this action ID in post-processing tables. State actions must implement the following functions: (1) restoring the MAC addresses, IP addresses, and TCP ports according to the flow ID; and (2) setting the correct egress port to the replying packet. Other instructions such as updating the TCP sequence number depend on the user's testing requirements.

5 Emulating Realistic Traffic

Now we introduce how Norma emulates realistic traffic. We have two requirements for real traffic generation. First, the types of outgoing traffic generated by Norma cover our test scenarios (§5.1). The packet editor can modify the fields of the outgoing packet fields according to the user's test tasks such as port scanning and host probing. Second, the rate of outgoing traffic controlled by Norma covers our test scenarios (§5.2). Norma leverages the packet header compression technique to overcome the bandwidth bottleneck caused by bridged metadata conveyance, thus achieves the line-rate traffic generation (Appendix B). On this basis, the rate/burst controller is able to control the outgoing traffic rate accurately and the burst pattern can be customized by the controller to emulate the network traffic in corner cases.

5.1 Two-Stage Packet Editor

If the resources of the external pipeline are sufficient, the packet editor only needs to be implemented on the external egress pipeline. However, most resources of the external pipeline has been occupied by switching functions. There are no more stages available to support register operations like generating random numbers and execution of the packet editor table actions like packet field assignment.

We propose a two-stage editing mechanism to overcome this limitation. Instead of editing the outgoing packet on the external egress pipeline only, Norma splits the packet editor into two parts. Because the internal egress pipeline is not used by the switching function and measurement functions. We implement a major packet editor that edits the template packets on the internal pipeline first to complete most of the work. However, modifying the template packet alone is not enough. The outgoing traffic on multiple front-panel ports is identical if these modified packets are replicated to these ports through the PRE. So there is a minor one that edits the outgoing traffic on the external pipeline to differentiate them on each port. We leave implementation details in Appendix C.

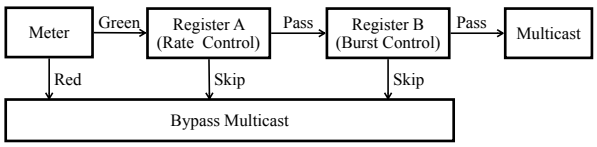


Figure 4: The multi-stage rate control mechanism. Meters and registers determine whether to multicast the template packets or not to control the traffic rate and burst pattern.

5.2 Precise Rate & Burst Control

The traffic rate is an important feature in realistic traffic emulation. Norma is required to send the traffic exactly at the configured rate with diverse burst patterns. The meter [36] (usually implemented by a token bucket) is a common rate control component provided by the programmable switch ASIC. In Norma, the meter first colors the template packets that loop in the internal pipeline at line rate to red or green. The multicast controller then marks the drop flag to the red packets and writes the multicast metadata to the green packets to generate outgoing traffic at the target rate. But the following two limitations make the meter not quite practical. 1) The meter colors the packets of the stable traffic with an equal time interval. It is impossible to generate burst traffic where packets are expected to be sent in batches. 2) The target rate has a finite precision, *i.e.*, not all target rates can be precisely configured. The actual rate of the meter can be different from what we set, and the error grows even larger if we choose a shallower bucket depth to avoid unexpected bursts.

Norma proposes a multi-stage rate control mechanism in the rate/burst controller to obtain the ability of accurate rate control and bursts. As shown in Figure 4, the mechanism is based on a meter, appended with multiple register units (*e.g.*, 2 units in the figure). Each unit is implemented in the same way, which skips the following m packets after passing n successive packets. These units are connected in a cascaded way, and the parameters m and n can be configured individually. Next, we show how the design solves both two problems.

First, for the case of accurate rate control, the user wants to generate 10 Gbps traffic. However, the two closest rates supported by the meter are 9.9 Gbps and 10.1 Gbps. In this case, we can configure the meter to the larger rate. The parameters m and n of register A in Figure 4 are set to 1 and 100, respectively, which means skipping 1 packet after passing 100 packets. Therefore, the final rate becomes $\frac{10.1\ \text{Gbps} \times 100}{1+100} = 10$ Gbps.

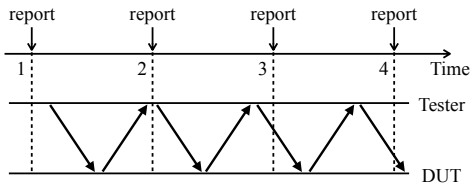


Figure 5: Cross-interval packets make counters out of sync. In time 2-3, the tester sends two packets and reports one is lost, but it receives the lost one in time 3-4.

Second, for the case of burst pattern control, the user wants to send burst traffic whose peak rate and average rate are 10 Gbps and 4 Gbps, respectively, which means there are 1,000 packets in a burst batch. To meet this goal, we only need to configure the m and n of register B to 1,000 and 1,500, respectively. These registers can be used flexibly. We can use two or more registers together to get a more precise rate or to construct complex burst patterns.

6 More Practical Considerations

To use Norma in practice, we also need to address some engineering challenges. For a load tester, measurement is one of the most important engineering challenges, since it is used to estimate the performance of tested networks. We have two requirements for the measurement. First, the measurement must be accurate, which means that the measurement must be done on the data plane as much as possible. Second, the measurement should not affect the functionality of the DUT and the pattern of the outgoing traffic. This means that the outgoing packet should not carry additional headers to store information such as timestamps.

This section first presents a measurement technique based on a flapped version bit, which can obtain high-precision traffic metrics in real time (§6.1). Then, we detail the blind measurement technique used in the delay measurement that avoids adding extra information to the outgoing packets (§6.2).

6.1 High-Precision Real-Time Measurement

A straightforward way is to let the programmable switch ASIC periodically report the counter value to control plane programs. Then, the metrics (*e.g.*, throughput) can be calculated as the quotient of the counter value difference and the reporting period. However, this method is inaccurate for complex metrics relying on bidirectional measurements. Consider the measurement of packet drop rate in Figure 5, which counts in both ingress and egress directions and reports the difference. Note that there is a *cross-interval packet* that is sent before the third report (at time 3) and received after it. Thus the ASIC knows that there are two packets sent in total and only one packet received during the reporting interval 2-3, and reports "one packet is lost" to the control plane at time 3. But in fact, there is no packet loss.

Norma synchronizes the counters in two directions by embedding a version bit in packet headers (*e.g.*, one bit in the IPv4 *identification* field). The version bit flaps every time a report happens. For example, in Figure 5, the version bit values in three reporting intervals can be (1,0,1) or (0,1,0), respectively. In the meanwhile, each original counter will be replaced by a counter group composed of two counters, corresponding to two versions. When receiving a packet, the ASIC reads the version it belongs to from the packet header, and updates the counter indexed by the version. Choosing the reporting period to a value larger than the maximum forwarding time of the DUT, the cross-interval packets will disappear.

While Norma achieves high-precision real-time measurement, it also adds a delay in reporting period to the data report. In addition, the SRAM used by counters is doubled.

6.2 Blind Measurement of Forwarding Delay

Typically, the forwarding delay can be measured by embedding a timestamp at the end of the packet. When Norma receives it, the timestamp will be parsed out and then compared with the current timestamp. However, this method cannot be applied to the programmable switch ASIC, whose pipelines are unaware of the packet payload. An alternative way is to embed to timestamp between the headers where the ASIC can parse, but it does not work for stateful protocols. For example, the HTTP header cannot be parsed by the ASIC due to the variable header length, and such embedding inserts the timestamp between the TCP header and the HTTP header. When the DUT (*e.g.*, an L7 gateway) receives the packet, it incorrectly treats the timestamp as an HTTP header. Another way is to embed the timestamp in packet headers, but there is not enough room for a 32-bit nanosecond timestamp, which is necessary for measurement precision.

Norma proposes the *blind measurement* technique, which does not embed or add a timestamp into the packet but sends blindly. Our approach is based on the synchronization framework (§6.1). Within each reporting interval, the ASIC records the timestamp of the first outgoing packet in a register and regards the value as base timestamp B_o. For following outgoing packets in the interval, the ASIC calculates relevant time as the difference between its timestamp and the base timestamp and adds it to a time register. Here we use the relevant time to avoid arithmetic overflow and denote T_o and P_o as the sum of relevant time and the outgoing packet number, respectively. The ASIC processes the incoming packets in the same way, and we denote corresponding values as B_i, T_i, and P_i, respectively. When the interval passes, the ASIC reports the all above values to the control plane, and then the control plane calculates the average delay d as: $d = (B_i + T_i/P_i) - (B_o + T_o/P_o)$. It is noticeable that packet loss can affect the precision of blind measurement. We leave the analysis in Appendix D.

We use the example in Figure 6 to illustrate how it works. Assume that the base timestamps of outgoing and incoming packets are 1000 and 1005, respectively. Three outgoing packets are sent at the time 1010, 1020, and 1030, so the time

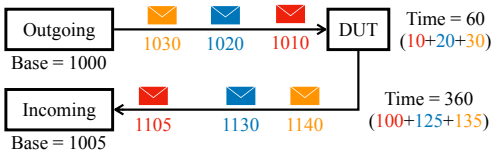


Figure 6: An example of blind measurement. The time offsets are accumulated by Norma and not carried with packets.

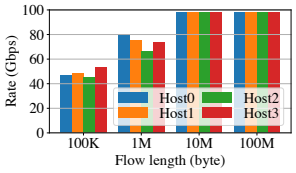


Figure 7: CDN load balance throughput.

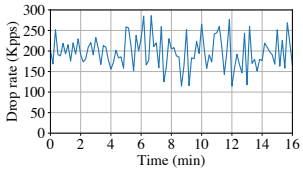


Figure 8: Packet drop on the traffic manager.

register of outgoing packets is added by 10, 20, and 30, respectively, which sum up to 60. Similarly, the time counter of incoming packets is 360. Therefore, the control plane calculates the average delay as $(1005+360/3)-(1000+60/3)=105$.

7 Implementation

Different from prior work, Norma depends on the pipeline-folded programmable switch ASIC, and therefore, is built from scratch. We write about 1,200 and 1,000 lines of P4 code to implement HTTP and TCP traffic responding functions, respectively. Except that, we write about 8,400 lines of P4 code to implement the rest of the data plane, as well as basic switch functions such as routing and ACL.

The control plane of Norma is implemented with about 2,500 lines of Python code and runs in a SONiC-like operating system [12]. It uses internal gRPC to communicate with the ASIC and provides HTTP APIs to users for job management. With these APIs, users can create a traffic-sending job with a desired packet header stack, traffic rate, burst pattern, measurement, stateful responder, *etc.*, and then submit it to Norma. After that, the job manager automatically allocates loopback ports (Appendix E) to these jobs and starts sending packets, until the job is terminated by users.

8 Case Study

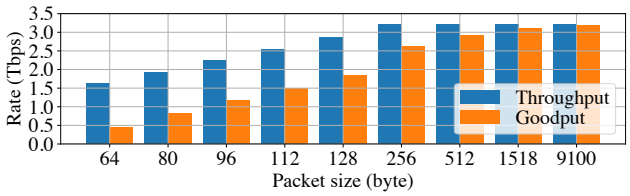
Norma has been used to test our pre-online devices for over two years. We present three real usage experiences.

CDN load balancer stress test. In a CDN system, the load balancer (LB) is responsible for distributing user requests to backend servers and then returning the servers' responses to the user. The LB, therefore, needs to afford a large amount of traffic. Its performance is critical to the CDN system. In one of our production pre-deployment, the load balancer employs four 100 Gbps links to connect to the ISP network and uses the other four 100 Gbps links to connect to the backend servers. To test the LB, we used Norma to emulate the traffic from both the ISP network and the backend servers. The traffic of the emulated HTTP clients was sent to the ISP ports of the LB, and the traffic of the emulated HTTP server was sent to the backend server ports of the LB. Norma initiated and maintained 4,000 HTTP connections. If one connection ends normally, Norma re-initiates the connection; if one connection ends abnormally, Norma shuts down the connection. Therefore we changed the connection establishing frequency by tuning the flow length to test whether there existed any performance issue. As shown in Figure 7, we observed that when the flow length was greater than 10 MB, the throughput of each backend server was close to 100 Gbps; however, when the flow length was less than 1 MB, the throughput was lower than 80 Gbps due to the limitation of the HTTP connection establishment capability of the LB. We therefore successfully measured the performance specifications of the LB under different types of loads.

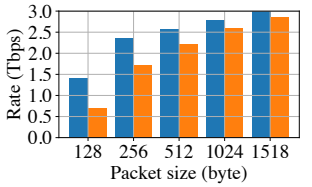
Traffic manager burst traffic test. In another LB setup, our switch should connect to the ISP network with two 100 Gbps links and 32 backend servers with 25 Gbps links. Normally, the throughput of the user requests from the ISP to the backend servers should be 50 Gbps. These requests trigger about 90 Gbps replying traffic, and 200 Mbps synchronization traffic from each backend server to other servers. But long-term operation in practice showed that there was packet loss on the LB. In troubleshooting, we find that requests sometimes generate bursts of 7 Gbps lasting 9-10 ms on one host, and drive the burst of synchronization traffic. Even with the bursts, this level of traffic should not cause a significant packet loss on the DUT. However, the QAC (Queue Admission Control) drop counters of backend servers increased irregularly.

The root cause is the improperly configured traffic manager. The burst traffic can rapidly fill the queue up and then cause packet loss. To tune the traffic manager configuration, we set the burst mode to sending 3,000 packets at 2.5 times the average throughput intermittently on the request traffic and the synchronization traffic, which can reproduce the bursts and packet loss. Figure 8 shows our result. There were about 197,000 packets dropped by the traffic manager every second. And therefore, Norma assisted our operators to optimize the traffic manager configuration.

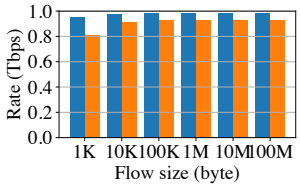
ARP learning rate test. We have deployed many programmable switches in our network. We need to ensure the correctness and speed of L2/L3 forwarding functions. To this end, we used Norma to connect these DUTs (*i.e.*, tested programmable switches) with two links. On one link, Norma generated ARP-reply traffic at line rate. It announced the MAC address of a segment of free IPs as the tester itself's. On another link, Norma generated UDP traffic at line rate, where the destination IPs were the free IP addresses announced by the ARP traffic. The DUTs must be able to learn ARP entries correctly first. Second, the DUTs need to forward UDP traffic according to the learned ARP entries. Finally, Norma judged whether the DUTs had learned all ARP entries by measuring the throughput of the forwarded UDP traffic, and then



(a) UDP traffic.



(b) TCP traffic.



(c) HTTP traffic.

Figure 9: Maximum throughput of Norma.

calculated the ARP learning rate.

In our testing, Norma showed that when the number of tested IP addresses was 2^{13}, the learning rate was about 462 ARP entries per second. However, when the number of IP addresses reached 2^{14}, the DUTs failed to learn all ARP entries. After our troubleshooting, we found that the control plane used a hash value of the entry to locate the ARP entry. If a hash conflict of ARP entries occurred, the control plane returned an exception. In addition, the control plane only supported up to 2^{14} entries, which made our test trigger hash collisions. These bugs were hard to find in unit tests because the number of ARP entries in unit tests is limited. No assertion can be triggered without hash collision.

9 Evaluation

All of our experiments were conducted in two programmable switches with Tofino programmable switch ASICs, where one was the tester, and the other was the DUT. Two switches were connected via 32 100 Gbps optical fibers, which provided 3.2 Tbps bidirectional bandwidth in total.

The traffic quality of Tofino ASICs and software testers has been well-learned in HyperTester [55] (see Appendix F). So we omit these experiments in the evaluation. Norma generates traffic only via dedicated hardware, rather than software. Therefore, the traffic generated is quite stable and very easy to reproduce. We got exactly the same results from multiple runs in our experiments.

9.1 Traffic Throughput

In this part, we evaluated the maximum throughput of three typical traffic, including UDP, TCP, and HTTP. Packets or flows with different sizes were required to generate to test the performance limit of Norma. And loopback ports were allocated by the algorithms in Appendix E. Unless otherwise specified, the Ethernet header and frame check sequence are taken into account when we describe packet or flow sizes, while the inter-packet gap and preambles are not.

UDP traffic. We first evaluated Norma's throughput of UDP traffic with different packet sizes, including small packets ranging from 64 to 512 bytes, MTU-sized packets (1518 bytes), and jumbo packets (9100 bytes). The allocation of loopback ports was straightforward for UDP, whose outgoing packets were directly multicast from template packets, and did not need other packets to trigger. Therefore, only one loopback port was occupied by template packets.

Figure 9a shows our results. Norma can generate traffic at the rate of at least 1.6 Tbps and reaches 3.2 Tbps for packets longer than 256 bytes. Two bottlenecks limit the performance of Norma. For small packets, the throughput was bounded by the operating frequency of the ASIC because there were more headers for the pipelines to process. And for large packets, however, the throughput was bounded by the 100 Gbps port rate. We used goodput to represent the transmission rate of the payload. As the packet size increased, the proportion of the packet header decreased, so the goodput increased. These results indicated that Norma's performance reached the limit of ASIC's capability. In the meanwhile, HyperTester can generate UDP traffic at the rate of 1.6 Tbps [55] and can be simply extended to 3.2 Tbps for large packets, similar to Norma.

TCP traffic. Next, we evaluated Norma's throughput of TCP traffic with state machine bypass. The TCP packet received was forwarded to the loopback pipeline where packet information was enqueued directly. Then the template packets looped in the pipeline read the information from the queue and generated a replying packet based on TCP flags. For example, when receiving a pure ACK packet, Norma would send back a PUSH packet with the payload of a specified size. Since the size of ACK packets was much smaller than that of PUSH packets, one loopback port was enough to process the ACK traffic received from multiple front-panel ports. For example, one loopback port processing ACK packets can support three front-panel ports that sent out 256-byte PUSH packets. That means, every four loopback ports can support up to 300 Gbps TCP traffic.

Figure 9b shows our results. According to the loopback port allocation algorithm in Appendix E, the expected throughput of PUSH packets sized by 128, 256, 512, 1024, and 1518 bytes was 1.6, 2.4, 2.6, 2.8, and 3.0 Tbps, respectively. However, the throughput of 128- and 256-byte packets was slightly lower than expected due to pipeline throughput limitations. For the rest of the PUSH packet sizes, each front-panel port generated near-line-rate TCP traffic.

HTTP traffic. Finally, we evaluated Norma's throughput of HTTP traffic by emulating HTTP sessions with different flow sizes, ranging from 1 KB to 100 MB. There were two types of packets in one HTTP session. One was the packets with HTTP content, and the other was the TCP control packets. Norma generated the packets with HTTP content by duplicating the template packets with the 1024-byte payload. For other packets, Norma used the template packet with no pay-

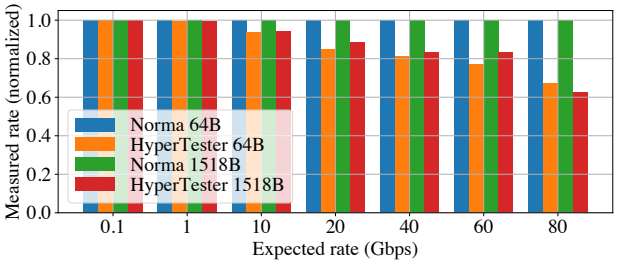
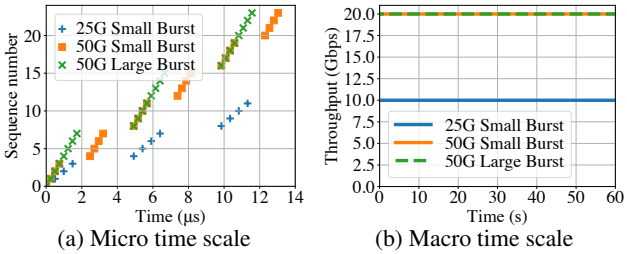


Figure 10: Comparison of rate control accuracy.



(a) Micro time scale (b) Macro time scale

Figure 11: The traffic generated by Norma's burst control.

load. To achieve the maximum throughput, in each 16-port internal pipeline, five ports were used for reading state registers, five ports were used for writing state registers, five ports were used for generating PUSH packets, and one port was used for generating TCP control packets. Therefore, we expected that Norma should generate 1 Tbps HTTP traffic.

Figure 9c shows our results. Norma generated HTTP traffic with over 950 Gbps throughput, which was slightly lower than expected. This was because of the gap between the looping frequency of large template packets (*e.g.*, 11 Mpps) and the arrival frequency of small packets (*e.g.*, 150 Mpps), which made the enqueue time and dequeue time unaligned. For example, a small HTTP control packet may wait in the queue for triggering a large data packet. This phenomenon was obvious when small packets dominate in short HTTP sessions and led to lower throughput.

Stability. We evaluated the stability of Norma by sending UDP, TCP, and HTTP traffic continuously over 24 hours, at the rate of 3.2, 3.0 and, 1.0 Tbps, respectively. We recorded the throughput of Norma periodically and found that it kept stable during the long-term run.

9.2 Traffic Control

In this part, we evaluated the traffic control capabilities in Norma in terms of rate control accuracy and traffic bursts.

Rate control. We evaluated rate control on two types of UDP traffic generated by Norma. One was composed of 64-byte packets and the other was MTU-sized packets. We measured the actual throughput of generated traffic and compared it with the expected rate. For clarity, the actual throughput was normalized by the expected rate. As shown in Figure 10, Norma achieved nearly 100% accuracy in all cases. However, the actual throughput of HyperTester became inaccurate as the expected rate grew larger. For example, the rate error reached 38% when the expected rate was 80 Gbps.

HyperTester controlled the rate of generated traffic by comparing the timestamp gap with a dropping threshold, which was calculated based on the expected rate. For any two successive packets, if the packet gap was less than the threshold, the second packet would be dropped. However, considering the scenario when the expected rate was more than half of the full rate, the dropping threshold would always be larger than the transmission time of a single packet. It meant there was always one packet being dropped for any two successive packets, and the actual rate cannot exceed 50% of the full rate. When the expected rate reached 80 Gbps, the actual rate of HyperTester was 80 Gbps $\times$ 62% $\approx$ 50 Gbps, which was the same as what we measured above.

We further evaluated the accuracy of our multi-stage rate control. We used the meter-based rate limiter provided by the programmable switch ASIC as a baseline. The actual rate is measured by a counter that counts how many packets pass through the egress pipeline in a range of time. The error is the ratio of the difference between the actual rate and the target rate to the target rate. For Norma, the rate control accuracy can be further guaranteed by our multi-stage rate control design. As shown in Table 3, the error of the meter-based rate limiter ranged between 0.1% and 1%, because the rate to limit supported by the hardware meter was not continuous. After applying the multi-stage rate control, the accuracy was promoted by at least 10$\times$, and the rate error was less than 0.01% in the worst cases.

Table 3: The rate error of our multi-stage rate control and pure meter when generating packets of different sizes.

Rate (Gbps)	64 Bytes		1518 Bytes	
	Multi-Stage	Pure Meter	Multi-Stage	Pure Meter
0.1	6×10^{-6}	3×10^{-3}	2×10^{-5}	1×10^{-3}
1	9×10^{-7}	2×10^{-3}	4×10^{-6}	5×10^{-4}
10	8×10^{-8}	4×10^{-3}	1×10^{-6}	3×10^{-3}
20	5×10^{-8}	4×10^{-3}	4×10^{-8}	3×10^{-3}
40	3×10^{-8}	4×10^{-3}	2×10^{-7}	3×10^{-3}
60	5×10^{-5}	3×10^{-3}	2×10^{-6}	8×10^{-4}
80	2×10^{-4}	4×10^{-3}	1×10^{-7}	3×10^{-3}

Burst control. To test the burst control, we made the Norma to generate three kinds of traffic with different burst patterns. For traffic A, B, and C, we set the expected average rate to 10 Gbps, 20 Gbps, and 20 Gbps, and the burst scale (*i.e.*, the number of packets in a burst batch) to 4, 4, and 8, respectively. In addition, the burst rate (*i.e.*, peak rate) of all the traffic was required to be 2.5$\times$ the average rate. The performance of burst control was evaluated on two scales. The micro time scale showed the packet-level burst pattern, while the macro time scale showed the traffic-level throughput. For the micro time scale, we gave each packet an increasing sequence number and recorded their transmission time. The result was shown in Figure 11a. In the two burst patterns with an average rate of

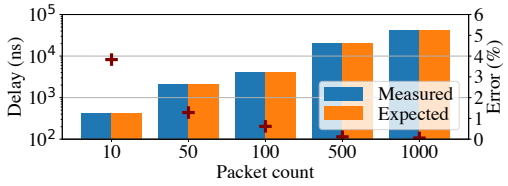


Figure 12: The packet delay when DUT meets bursts.

50 Gbps, there were 8 packets sent continuously in each batch under the large burst setting, and there were 4 packets under the small burst setting at the same rate with half the batch size. For the burst pattern with an average rate of 25 Gbps, there were 4 packets sent continuously at half rate in the same batch size as the 50 Gbps large burst pattern. At the macro time scale shown in Figure 11b, they all stayed at the target average rate.

9.3 Measurement

In this part, we combined traffic generation and burst control to evaluate the measurement function of Norma. First, Norma was connected to the switch under test and generated 100 Gbps line-rate UDP background traffic. The switch then forwarded background traffic back to one specified port of the tester. Second, Norma generated burst traffic at 100 Gbps with different burst scales. And burst traffic was also forwarded by the switch back to the same port of the tester. The packet size of burst traffic and background traffic was both 1024 bytes. Finally, Norma measured the average delay of packets in the burst traffic and compared it with the expected queuing time to judge the error of Norma's measurement.

Note that in this case, both burst traffic and background traffic were queued at the same egress port of the switch. We denoted the burst scale as n packets, and then the theoretical average packet queuing time was $\frac{(1024+20)\text{ Bytes}}{100\text{ Gbps}} \times \frac{n}{2}$[5]. In addition, the delay measured by Norma included the link propagation time, which is about 1454 ns.

Results are shown in Figure 12, where the link delay has been removed. For all of the burst scales, the queuing delay measured by Norma was very close to the theoretical value. When the burst scale was greater than 10 packets, the error of the average delay was less than 4%. The larger the burst size, the more accurate the measurement of average delay was.

10 Limitation & Discussion

Can Norma emulate full functions of stateful protocols? It depends on the complexity of the protocol. Besides the hardware limitations we detail in §4.1, the hardware resources also constrain the implementation of the stateful protocol. A stateful protocol in Norma consists of its state transitions, the replying traffic types, and the state actions. The capacity of the state table determines how many state transitions Norma can support. The loopback ports determine how many template packet types Norma can support and the maximum throughput of replying traffic Norma can generate. And most importantly, the state action may be too complex to implement into the programmable switch ASIC, because the switch resource allocation algorithms and related optimizations in compilers are unknown to developers. Without trying to implement the protocol and compile it, it is hard to know whether a stateful protocol can be fully emulated. Therefore, for complex protocols, we need to simplify them under the premise of being able to complete the test task.

Can Norma support testing customized protocols? Users can customize the packet structure and the processing logic of the protocol in most cases. For example, if we want to measure the forwarding performance of the GPRS tunneling protocol [3], we can modify the parser and deparser to support it. If we want to measure the RDMA write-only throughput of a host, things become complex. The process of exchanging information, congestion control algorithm and packet loss recovery are difficult to express with the EFSM. Even if possible, it is difficult to implement within limited instructions. Our approach is to retain only the process of transferring content in RDMA and remove other logic such as congestion control. However, Norma cannot support protocols with encryption due to the limitation of programmable switch ASICs, unless the encrypted data can be regarded as a fixed payload, or special acceleration cards such as IPU [4] are available.

Can Norma localize the root cause of performance issues? Norma cannot localize the root cause of performance issues because the DUT is typically a black or gray box that cannot be simply modeled. For example, a performance problem can come from misconfigurations, ASIC capabilities, bottleneck of switch CPUs, and even signal strength when wireless links involve. It might be possible to extend Norma to root cause localization if sufficient information is provided.

11 Conclusion

We present Norma, the first practical network load tester used in production. Norma employs the programmable switch ASIC to support stateful protocol generation and customization and realistic traffic emulation such as high precise rate control. Norma has been used in our operation for over two years and successfully detected many performance issues.

Acknowledgments

We thank our shepherd, Muhammad Shahbaz, and NSDI reviewers for their insightful comments. We also thank Xiaoliang Wang for his valuable feedback on earlier drafts of this paper. This work is supported by Alibaba Group through Alibaba Research Intern Program. Yanqing Chen, Chen Tian, and Guihai Chen are also supported in part by the National Key R&D Program of China (2022YFB2702803), the National Natural Science Foundation of China under Grant Numbers 62072228, and the Fundamental Research Funds for the Central Universities. Chen Tian and Ennan Zhai are co-corresponding authors.

[5] The inter-packet gap and preambles (20 bytes) should be considered.

References

[1] Alveo U200 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#buy-from-xilinx.

[2] DPDK. https://www.dpdk.org.

[3] GPRS tunnelling protocol. https://www.3gpp.org/DynaReport/29274.htm.

[4] Intel IPU. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[5] Intel Tofino 3.2 Tbps, 4 pipelines. https://www.intel.com/content/www/us/en/products/sku/218642/intel-tofino-3-2-tbps-4-pipelines/specifications.html.

[6] iPerf. https://iperf.fr.

[7] Keysight. https://www.keysight.com/us/en/products/network-test/network-test-hardware.html.

[8] PF_RING ZC. https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[9] Pktgen. https://github.com/pktgen/Pktgen-DPDK.

[10] Programmable data plane at terabit speeds. https://conferences.sigcomm.org/sigcomm/2018/files/slides/p4/P4Barefoot.pdf.

[11] Scapy. https://scapy.net/.

[12] SONiC. https://sonic-net.github.io/SONiC.

[13] Spirent. https://www.spirent.com/products/testcenter-ethernet-ip-cloud-test.

[14] Tofino native architecture - public version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[15] Tofino product family brochure. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/tofino-product-family-brochure.pdf.

[16] Tofino programmable Ethernet switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[17] Trafgen. http://netsniff-ng.org.

[18] TRex. https://trex-tgn.cisco.com.

[19] Gianni Antichi, Charalampos Rotsos, and Andrew W. Moore. Enabling performance evaluation beyond 10 gbps. *SIGCOMM Comput. Commun. Rev.*, 45(4):369–370, aug 2015.

[20] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, apr 2014.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[22] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[23] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547, 2012.

[24] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *2013 USENIX Annual Technical Conference (ATC)*, 2013.

[25] Kwang Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, 1993.

[26] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. Drmt: Disaggregated programmable switching. In *ACM SIGCOMM (SIGCOMM)*, 2017.

[27] G. Adam Covington, Glenn Gibb, John W. Lockwood, and Nick Mckeown. A packet generator on the netfpga platform. In *17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2009.

[28] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies (CoNEXT)*, 2017.

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[30] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. A network analysis on cloud gaming: Stadia, geforce now and psnow. *Network*, 1(3):247–260, 2021.

[31] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the Internet Measurement Conference (IMC)*, 2015.

[32] Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. Mind the gap - a comparison of software packet generators. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2017.

[33] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM (SIGCOMM)*, 2020.

[34] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[35] Philippe Graff, Xavier Marchal, Thibault Cholez, Stéphane Tuffin, Bertrand Mathieu, and Olivier Festor. An analysis of cloud gaming platforms behavior under different network constraints. In *17th International Conference on Network and Service Management (CNSM)*. IEEE, 2021.

[36] J. Heinanen and R. Guerin. Rfc2698: A two rate three color marker. Technical report, RFC Editor, USA, 1999. https://www.rfc-editor.org/rfc/rfc2698.html.

[37] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets)*, 2019.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[39] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[40] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[41] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM (SIGCOMM)*, 2020.

[42] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM (SIGCOMM)*, 2017.

[43] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. IMap: Fast and scalable in-network scanning with programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[44] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[45] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM (SIGCOMM)*, 2021.

[46] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[47] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference (ATC)*, 2012.

[48] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. Oflops: An open framework for openflow switch evaluation. In *Passive and Active Measurement (PAM)*, 2012.

[49] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM (SIGCOMM)*, 2016.

[50] Junji Takemasa, Ryoma Yamada, Yuki Koizumi, and Toru Hasegawa. Ccngen: A high-speed generator of bidirectional ccn traffic using a programmable switch. In *Proceedings of the 8th ACM Conference on Information-Centric Networking (ICN)*, 2021.

[51] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020.

[52] Michael D. Wong, Aatish Kishan Varma, and Anirudh Sivaraman. *Testing Compilers for Programmable Switches through Switch Hardware Simulation*. 2020.

[53] Zhaowei Xi, Yu Zhou, Dai Zhang, Jinqiu Wang, Sun Chen, Yangyang Wang, Xinrui Li, HaoMing Wang, and Jianping Wu. Hypergen: High-performance flexible packet generator using programmable switching asic. In *ACM SIGCOMM Posters and Demos*, 2019.

[54] Yiling Xu, Qiu Shen, Xin Li, and Zhan Ma. A cost-efficient cloud gaming system at scale. *IEEE Network*, 32(1):42–47, 2018.

[55] Dai Zhang, Yu Zhou, Zhaowei Xi, Yangyang Wang, Mingwei Xu, and Jianping Wu. Hypertester: High-performance network testing driven by programmable switches. *IEEE/ACM Transactions on Networking*, 29(5):2005–2018, 2021.

[56] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2020.

[57] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 26(4):178–183, 2019.

[58] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, et al. Xlink: QoE-driven multi-path QUIC transport in large-scale video services. In *ACM SIGCOMM (SIGCOMM)*, 2021.

[59] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. HyperTester: high-performance network testing driven by programmable switches. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, 2019.

[60] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. NetFPGA: Rapid prototyping of networking devices in open source. In *ACM SIGCOMM (SIGCOMM)*, 2015.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A Non-Atomic State Updating

The state updating we detailed in §4.2 is non-atomic, which brings two drawbacks. First, this inevitably consumes an extra loopback port and increases the packet processing delay. Second, because the updating of the state registers is done in two passes, if the new state ID is too late to be written to the state register, the next stateful packet may read the old one. In this situation, the state register can be mistakenly written with the wrong value. Because the programmable switch ASIC does not provide the ability to schedule packets, this error can only be avoided by locking the state register and delaying the processing of the following stateful packets. This will bring more problems like keeping following stateful packets looping in the internal pipeline buffer and then handling them in order, which makes it impractical. Therefore, a more acceptable solution is to take the continuously incoming packets into account when designing the EFSM.

B Full-Speed Traffic Generation

How to elimate bandwidth consumption of bridged metadata? Norma generates traffic by multicasting the template packets that loop inside the internal pipeline. However, there is no extra bandwidth reserved for the bridged metadata when the loopback port conveys the packet from the egress back to the ingress, so all metadata must be packed into the packet, which increases the length of the packet and forms a bandwidth bottleneck. For example, consider the scenario when we want to send outgoing traffic composed of 64-byte small packets at the rate of 100 Gbps from one front panel port. Assume the egress of the loopback port adds 12-byte metadata to the template packet, so the packet loops at the rate of $\frac{100\text{ Gbps}}{(64+20+12)\times 8} = 130$ Mpps. Although the ingress will parse and remove the metadata header, the packet rate could not increase anymore. As a result, the throughput of outgoing traffic is only $130\text{ Mpps} \times (64+20) \times 8 = 87$ Gbps. This is a severe problem for the network tester, which makes it impossible to test the DUT under 100% workload.

The key idea of Norma is to compress the packet header. We noticed that the bandwidth bottleneck only exists in the internal pipeline, instead of the external pipeline, because only the internal pipeline has egress-to-ingress forwarding. This enables Norma to borrow some header bits to temporally store the metadata in the internal pipeline, and pay them back at the egress of the external pipeline. The key point is to find the "traffic/port-invariant" fields, whose value is determined once the flow ID and outgoing port are given. Practically, Norma compresses the Ethernet header and borrows 12 bytes in total.

- The 2-byte *Ethernet Type* field can be compressed to one byte. This field indicates the type of the next header in the packet. For example, 0x0800 represents the IPv4 header. In cloud scenarios, there are no more than 10 possible values for this field, which can be encoded in one byte.
- The *Src MAC* and *Dst MAC* fields can be compressed together to one byte, which occupy 12 bytes in the original Ethernet header. MAC addresses are usually fixed given the traffic ID and the outgoing port, so storing the 1-byte traffic ID is enough. When the traffic arrives at the egress of the external pipeline, these fields can be recovered.

How many template packets are needed in one loopback port? In the RMT-based programmable switch ASIC, one template packet is not enough to make full use of the hardware pipeline of a loopback port. Multiple template packets are required to guarantee the line-rate looping. But unfortunately, the exact number of template packets we need depends on many factors, including the packet size, the packet header depth, and how the compiler arranges P4 tables, which makes it hard to predict. Based on our experience, the number of template packets can be represented as a function like $y = Ax^{-B} + C$, where A, B, and C are unknown constants, and x is the packet size. When the packet type and the P4 program are fixed, the function can be determined via curve fitting. In general, 10 and 120 packets are enough when packet sizes are 1500 and 64 bytes, respectively.

C Implementation Details of Packet Editors

The major editor can apply step-based or random-based field editing. There are five modes Norma supports:

1) The **direct step mode** simply adds a constant to the initial value. If the value exceeds the bound provided by the user, it will be subtracted by the bound.
2) The **indirect step mode** is similar to the direct step mode. Differently, the value is not directly outputted but used as an index to access a register array, which saves the real value to the output provided by the user.
3) The **cascaded step mode** can be regarded as a combination of two editors working in the direct step mode. The first one works as normal, but the second one is triggered only when the bound excess happens in the first one.
4) The **direct random mode** simply fills some bits of a field with random bits. For example, filling the rightmost 8 bits of the source IP field means randomly choosing an IP address under a /24 prefix.
5) The **ranged random mode** also relies on random bits. Differently, Norma is required to choose a number from a given range. For example, choose a number for the source port field uniformly from the range 2000-3000.

Among all these modes, the ranged random is the most complex one. It is the random bit instead of the random number that is provided by the programmable switch ASIC, which

means the length of the range must be a power of 2. For example, with 8 random bits, we can only get a uniform random number from 0 to 255, instead of any other range. A straightforward way is to keep trying until a number in the required range is acquired. However, it is not suitable for the ASIC, because there is no cheap way to emulate the while-loop. So we need a concise method that does not rely on loops. Norma solves this problem by using the equation $n := (n' + r) \bmod l$, where n' is the random number generated in the last execution, stored in the corresponding field of the template packet. Here, r is a k-bit random number generated by the ASIC satisfying $2^k \leq l$, where l is the length of the range. Note that the mod operator is not supported by the ASIC, but the restriction to k makes it representable with no more than one subtraction and thus can be implemented in the ASIC. We evaluated the quality of generated random numbers as follows.

We applied the packet editor to UDP packets, whose source ports were randomly chosen from the range 0-999, so the length of the range was 1000, and k (*i.e.*, the number of random bits) should be set to a number no more than 9 to satisfy the constraint $2^k \leq 1000$. From Figure 13a and Figure 13b, we can observe that the quality of the generated random number improved as k becomes larger, benefiting from more random bits provided. Figure 13c shows the frequency of each number generated from 100,000 packets. Each number occurred at a frequency of around 0.1% as expected, which indicated the uniformity of the generated numbers.

D Analysis of Blind Measurement

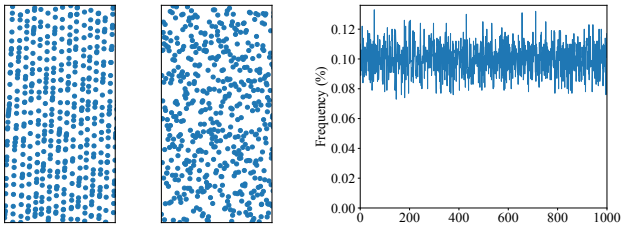
We use the sets $\mathbb{S}$ and $\mathbb{R}$ to represent the packets sent by the egress pipelines, and received from the ingress pipelines, so we have $\mathbb{R} \subseteq \mathbb{S}$, and $drops = |\mathbb{S}| - |\mathbb{R}|$ is the number of packets dropped by the DUT. The real average delay can be represented as $delay = \frac{1}{|\mathbb{R}|}\sum_{i\in\mathbb{R}}(r_i - s_i)$, where the dropped packets are excluded due to incomplete information. However, the blind average delay becomes $delay' = \frac{1}{|\mathbb{R}|}\sum_{i\in\mathbb{R}} r_i - \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}} s_i$. Now, we define the absolute measurement error e as the difference between $delay$ and $delay'$, and we have

$$
\begin{aligned}
e =& |delay' - delay| \\
=& \left| \frac{1}{|\mathbb{R}|}\sum_{i\in\mathbb{R}} s_i - \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}} s_i \right| \\
=& \left| \frac{1}{|\mathbb{S}|}\left(\sum_{i\in\mathbb{R}} s_i + \sum_{i\in\mathbb{S}\setminus\mathbb{R}}\left(\frac{1}{|\mathbb{R}|}\sum_{j\in\mathbb{R}} s_j\right)\right) - \frac{1}{|\mathbb{S}|}\left(\sum_{i\in\mathbb{R}} s_i + \sum_{i\in\mathbb{S}\setminus\mathbb{R}} s_i\right) \right| \\
=& \left| \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}\setminus\mathbb{R}}\left(\frac{1}{|\mathbb{R}|}\sum_{j\in\mathbb{R}} s_j - s_i\right) \right| \leq \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}\setminus\mathbb{R}}\left| \frac{1}{|\mathbb{R}|}\sum_{j\in\mathbb{R}} s_j - s_i \right| \\
\leq& \frac{|\mathbb{S}| - |\mathbb{R}|}{|\mathbb{S}|}\left(\max_{i\in\mathbb{S}} s_i - \min_{i\in\mathbb{S}} s_i\right) = \frac{drops}{pps_{tx}},
\end{aligned}
$$

which means, the more the packets are dropped, or the slower the packets are sent, the larger the error could be.

E Loopback Port Allocation

As shown in Figure 1, Norma uses two internal pipelines and each of which contains 16 loopback ports. Each loopback port has a maximum BPS rate (*e.g.*, 100 Gbps) while each pipeline has a maximum PPS rate shared by all ports belonging to it.



(a) $k = 6$ (b) $k = 9$ (c) Frequency

Figure 13: Pseudo-random ports ranging in 0-999.

Norma models the loopback port allocation as a satisfiability problem, which can be solved efficiently by SMT solvers such as Z3 [29]. Consider there are n flows using loopback ports. We use zero-one variables $x_{i,j,k}$ to indicate whether flow i should be placed to port k in pipeline j. The BPS rate and PPS rate of flow i are represented as b_i and p_i, respectively. The maximum BPS rate of a port and the maximum PPS rate of a pipeline are represented as B and P, respectively. Then loopback port allocation can be modeled as the following integer linear satisfiability problem:

$$\sum_i \sum_k x_{i,j,k} p_i \leq P \qquad \text{for each } j, \tag{1}$$

$$\sum_i x_{i,j,k} b_i \leq B \qquad \text{for each pair of } (j,k), \tag{2}$$

$$\sum_k x_{i,j,k} = \sum_k x_{i',j,k} \qquad \text{if } i \text{ shares data with } i', \tag{3}$$

$$\sum_j \sum_k x_{i,j,k} = 1 \qquad \text{for each } i. \tag{4}$$

Equations (1) and (2) describe the constraints of the PPS rate and BPS rate, respectively. Equation (3) enforces flow i and flow i' in the same pipeline if they share data via registers, such as the enqueuing flow and dequeuing flow in §4.3. Equation (4) guarantees that each flow will be placed to a port and a port is allowed to be shared by more than one flow.

For all of the cases we met, the satisfiability problems were solved by Z3 in less than 1 second. Then Norma can allocate loopback ports according to the solved variables $x_{i,j,k}$.

F Performance of Software Testers

Zhang *et al.* [55] evaluated software testers such as MoonGen [31] and TRex [18]. Results are summarized as follows. First, software testers cannot generate traffic at more than 300 Gbps due to PCIe bandwidth limitations. For small packets, even with 12 CPU cores, software testers can only generate traffic at about 40 Gbps. Second, the rate control of MoonGen relies on NIC meters, which means not all target rates can be precisely configured. TRex uses software timestamps for rate control, which leads to unstable inter-packet gaps. These results show that software testers cannot generate precisely-controlled high-speed traffic.

μMote: Enabling Passive Chirp De-spreading and μW-level Long-Range Downlink for backscatter Devices

Yihang Song[1], Li Lu[1], Jiliang Wang[2], Chong Zhang[1], Hui Zheng[1], Shen Yang[1], Jinsong Han[3], and Jian Li[1]

[1]University of Electronic Science and Technology of China
[2]Tsinghua University
[3]Zhejiang University

Abstract

The downlink range of backscatter devices is commonly considered to be very limited, compared to tremendous long-range and low-power backscatter uplink designs that leverage the chirp spread spectrum (CSS) principle. Recently, some efforts are devoted to enhancing the downlink, but they are unable to achieve long-range receiving and low power consumption simultaneously. In this paper, we propose $\mu Mote$, a μW-level long-range receiver for backscatter devices. $\mu Mote$ achieves the first passive chirp de-spreading scheme for negative SINR in long-range receiving scenarios. Further, without consuming external energy, $\mu Mote$ magnifies the demodulated signal by accumulating temporal energy of the signal itself in a resonator container, and meanwhile it preserves signal information during this signal accumulation. $\mu Mote$ then leverages a μW-level sampling-less decoding scheme to discriminate symbols, avoiding the high-power ADC-sampling. We prototype $\mu Mote$ with COTS components, and conduct extensive experiments. The result shows that $\mu Mote$ spends an overall power consumption of $62.07\mu W$ to achieve a $400m$ receiving range at a $2kbps$ data rate with 1% BER, under $-2dB$ SINR.

1 Introduction

Backscatter communication has significant advantages over active radio in terms of power consumption. The recognized drawback of conventional backscatter devices [10, 46] is their insufficient communication range of only a few meters. In the last decade, a number of long-range backscatter techniques [23, 35, 37, 51, 54, 55] are proposed that can significantly increase the uplink backscatter range to hundreds of meters or even more than $1km$. Their main idea is to reflect the Chirp Spread Spectrum (CSS) signal. When this chirp signal is de-spread by the gateway receiver, it will incur a processing gain for the receiver, and therefore more interference immunity and longer communication range.

Nevertheless, the downlink range of this type of long-range backscatter devices [18, 37] is limited to less than 20 meters [17]. As they still use the conventional envelope detector as a receiver [15, 30], rather than de-spreading the chirp signal. In the last two years, two efforts are devoted to extending the downlink range. Saiyan [17] leverages an RF band-pass filter (with $6dB$ signal strength loss) to assign an envelope to the chirp signal, and subsequently uses an envelope detector to detect the assigned envelope. This demodulation method does not de-spread the chirp signal, so it cannot obtain processing gain and extend the downlink range. Instead, to extend the range, Saiyan uses an LNA (Low Noise Amplifier) and an OP AMP (Operational Amplifier) to amplify the signal [16, 21, 40] and gains a receiving range of $180m$. However, these two amplifiers consume about $88mA$, which is similar to the current consumption of active radio, and 176 times that of the typical backscatter device [46]. Passive-DSSS [31] employs two envelope detection channels to transmit DSSS (Direct Sequence Spread Spectrum) spreading codes, enabling anti-interference receiving. However, it strictly requires a positive SINR. Besides, due to the performance limitation of the envelope detector, its downlink range is limited to about 50 meters, far less than the uplink range of existing long-range backscatter devices. Therefore, to the best of our knowledge, none of existing envelope detector-based receivers can achieve long-range receiving with low power consumption.

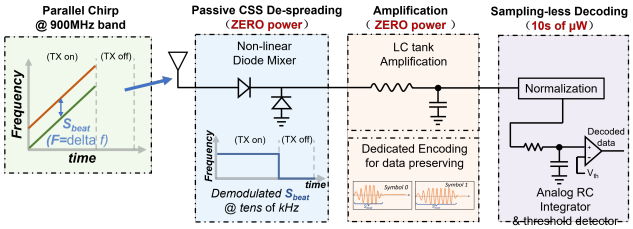


Figure 1: **The high-level block diagram and basic principle of $\mu Mote$.**

In an IoT network consisting of backscatter devices and a gateway, insufficient downlink range will be the bottleneck of network coverage. When a backscatter device is placed far away from the gateway, the incapability of downlink communication can lead to the failure of essential network func-

tions, including ACK and re-transmission [13], multi-access control [13], network association [27], over-the-air firmware switching or updating [59, 60], and device authentication [41].

Conventional methods to improve the downlink range can be classified into two categories. The one is to employ spread spectrum communication to suppress interference, so as to demodulate the signal at low SINR (signal-to-interference-plus-noise ratio) or even negative SINR. The other one is to use amplifiers (i.e., LNAs) [58] to enhance the receiver sensitivity and combat signal strength loss during propagation. De-spreading the spread spectrum signal, e.g., the CSS signal, requires a local-generated carrier and correlated de-spreading signals, which will incur very high power consumption (e.g., more than $7mW$ [6]). As for LNAs, by their nature (see Sec. 2.1), they commonly consume $10mW$ to more than $100mW$ [14, 47]. In summary, these two solutions are very power-consuming, and therefore cannot realize on backscatter devices whose overall power consumption is commonly less than $1mW$ [46].

In this paper, we present *µMote*, a μW-level re*Mote* receiver with hundreds of meters of receiving range, which can effectively work even under negative SINR. The high-level idea incorporates three key designs, as shown in Fig. 1: a novel Passive Chirp Spread Spectrum (Passive-CSS) de-spreading design to combat interference, a magnification scheme to magnify the demodulated signal with zero external power consumption to combat signal strength loss, and an efficient decoding design with only tens of μW.

More specifically, *µMote* addresses the following key challenges.

(1) How to address interference or even negative SINR with extremely low power? We try to leverage Chirp Spread Spectrum (CSS) technique to resist interference, which has been widely used in backscatter transmitters. Existing chirp de-spreading techniques, however, consume very high power consumption, as we describe above. To this end, we present the first Passive Chirp Spread Spectrum (Passive-CSS) technique, which removes the conventional power-consuming chirp demodulation and de-spreading processes. The basic idea is a parallel chirp modulation scheme for the gateway and then a passive chirp de-spreading circuit with zero power. Thus, we can passively decode the chirp signal while retaining the long-range and interference-resilient features.

(2) How to increase the signal amplitude without external power, and meanwhile preserve signal information? Traditional amplification solutions, such as LNAs, commonly consume more than ten mW of power. To solve this problem, we leverage an *LC* resonator (also called *LC* tank) as a container to accumulate the energy of demodulated signal so as to magnify the signal amplitude. However, when the signal energy is stored in such an *LC* tank, the corresponding signal information that is supposed to be in chronological order can be overlapped and distorted. To preserve the information, we expect a new encoding method that can embed information into the accumulated signal. Inspired by PIE (Pulse Interval Encoding), we propose a software/hardware co-designed encoding scheme that leverages the duration of the magnified signal to represent symbols.

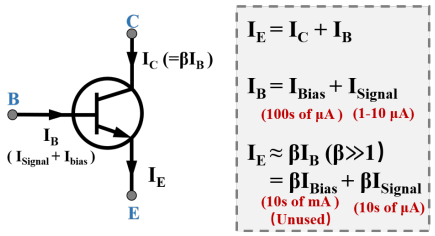


Figure 2: **Amplification principle and current consumption of an LNA.**

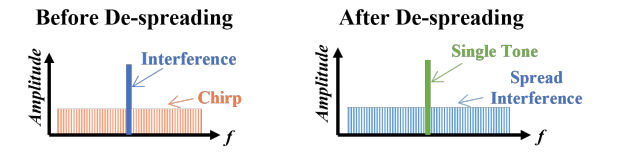
(3) How to efficiently decode symbols conveyed by the magnified signal? Existing receivers usually employ ADC for high-speed sampling and decoding. This accounts for the vast majority of the power consumption of backscatter devices. For example, a widely used low-power integrated ADC IC [49] typically consumes hundreds of μW power. In this paper, we present an analog *RC* integrator to decode without the need for ADC. This significantly reduces the power consumption to several μW. Specifically, we introduce a low-power energy integrator to identify the symbols based on the level of symbol energy. Moreover, the above circuit is designed to be programmable to achieve ADC-free symbol synchronization before decoding. Besides, we present a normalization scheme to address the "dynamic symbol energy" problem caused by the diversity of signal strength.

Summary of the contributions and results:

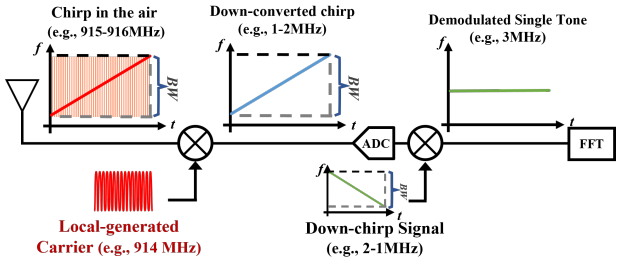
- We propose the design of *µMote*, a novel μW-level low-power receiver with hundreds of meters of receiving range which can effectively resist interference and work even under negative SINR.

- We address the fundamental challenges for realizing the receiver. We realize a passive-CSS communication technique on passive circuits, enabling zero-power chirp de-spreading and demodulating. We present the zero power magnification scheme by accumulating the energy of the demodulated signal itself. And we propose the design of a sampling-less analog energy integrator for μW symbol decoding.

- We prototype *µMote* with COTS components and conduct extensive experiments. The results demonstrate that *µMote* reaches a communication range of more than $400m$ at a 1% Bit Error Rate (BER) with an average working power of $61.07\mu W$. Compared to the literature of existing works [17, 31], *µMote* improves the downlink range by $2.5\times$ to $8.65\times$, with a power consumption reduction of at least 63.2%. Under the case of narrowband



(a) Spectrum of chirp and interference before and after de-spreading



(b) Procedures of conventional chirp de-spreading.

Figure 3: **Principle of conventional chirp de-spreading, in which the power consumption of carrier generation is not affordable for backscatter devices.**

interference, *μMote* can work at $-2dB$ SINR or above. To the best of our knowledge, this is the first receiver that can work under negative SINR with μW-level power.

2 Background and Motivation

Before presenting our design, it is worth investigating why existing receivers cannot achieve long-range receiving with low power consumption. We classify existing approaches to long-range receiving into two categories: amplifying the signal and spreading spectrum technique.

2.1 Amplifying the Signal

Receivers usually use LNAs to amplify the received signal strength, as LNAs' low noise figure can minimize the noise caused by the amplification process. An LNA usually consists of multiple triodes and peripheral analog circuits. Its function of signal amplification is performed by the triodes. Fig. 2 depicts the principle of a triode. By leveraging the external current injected to its collector (I_C), it can amplify small base current (I_B) by β times, and the amplified current is output as emitter current I_E, i.e.,

$$I_E = I_C + I_B = \beta I_B + I_B \approx \beta I_B (\beta \gg 1)$$

where the β inherently depends on the transistor and can be typically large up to 100.

However, the practical power efficiency of such triodes is very low. As shown in Fig. 2, the I_B contains not only the signal current (I_{Signal}), but also a bias current (I_{Bias}) that is significantly larger than I_{Signal}. The I_{Bias} contains no signal information but can ensure that the triode works within the linear active region (amplifying region). After amplification, the amplified I_{Bias} will be removed. In the widely used commercial LNA ICs [7, 8], the I_{Bias} is commonly hundreds of μA or even more than $1mA$. Hence, amplifying the I_{Bias} makes commercial LNA ICs consume several mA to even hundreds of mA of external current. As a comparison, the total power consumption of a typical backscatter device is $470\mu A$ [46].

2.2 Spread Spectrum Technique

Spread spectrum techniques, such as Chirp Spread Spectrum, has been widely used in LPWAN (Low Power Wide-Area Network) and backscatter transmitters due to their interference immunity and long communication range. In brief, the basic idea of CSS is "don't put all the eggs into one basket". Specifically, by transmitting or reflecting the wide-band chirp signal, the information (e.g., a symbol) is distributed to the chirp's band (the red shaded area in Fig. 3(a) and Fig. 3(b)). Hence, the narrow-band interference is unable to cover the entire band, even if it has a stronger signal strength, as shown in Fig. 3(a).

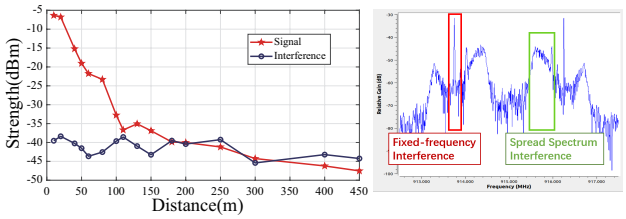
The receiver needs to de-spread the chirp to gather the distributed information. As shown in Fig. 3(b), the chirp is down-converted to a relatively low frequency (e.g., 1–$2MHz$) and then multiplied by a down-chirp signal, incurring a single tone signal which sums the frequency of the down-chirp and the down-converted chirp. Meanwhile, the narrow-band interference is also multiplied by the down-chirp signal, but then its energy is spread into the entire chirp's band. This means the single-tone signal gains high SINR and can be easily recognized by FFT (Fast Fourier Transform) operation (as shown in Fig. 3(a)). With the FFT results, the receiver knows whether a chirp is transmitted or not, even if the chirp has lower strength than that of the interference.

Nevertheless, the conventional chirp de-spreading process is power-consuming, as it requires the receiver to generate the high-frequency carrier for down-conversion and perform FFT operation. Generating the carrier needs power-consuming components, such as the VCO (Variable Crystal Oscillator) and the PLL (Phase Locked Loop), and performing FFT requires high-speed computing. These operations inevitably incur high power consumption that is undesirable for backscatter devices [29, 34]. Hence, chirp de-spreading has never been implemented on backscatter devices.

2.3 Motivation

After revisiting existing solutions and their power consumption, we aim to design a receiver with interference resistance, long-range receiving (high receiving sensitivity), and low-power consumption. We are motivated to take the following strategies to solve those problems, respectively.

Interference resilience: When the receiver is far away from the gateway, the received signal becomes susceptible



(a) (b)

Figure 4: **(a) Site survey for the strengths of ambient interference and downlink signal in the campus environment, and (b)Then measured spectrum of interference signals.**

to interference. This significantly decreases the receiving performance in practice, especially for non-noise-resistant envelope detectors. Hence, we are motivated to present a passive chirp de-spreading design to combat interference.

Low-power Magnification: Amplification can combat signal strength loss and improve receiving range. However, using LNAs can cause enormous power consumption. Thus, we present the LNA-free magnification scheme with zero power consumption, which helps to extend the receiving range.

Low-power decoding: Conventional symbol decoding solution is high-speed ADC sampling. However, it typically consumes hundreds of microwatts, accounting for the major part of the power consumption of a backscatter device [46]. Hence, we expect a power-efficient decoding mechanism without high-speed ADC sampling.

3 Passive Chirp De-spreading

The envelope detector is widely used in low-power receivers as it requires no high-power-consuming components and computing tasks. However, the envelope detector is subject to interference. When the interference is stronger than the downlink signal, it will misidentify the envelope of interference as the envelope of the downlink signal, thereby incurring communication failure. To better understand this, we conduct a preliminary site survey in a campus environment. As illustrated in Fig. 4(a), at the distance of 250m, 350m and above, the interference strength is stronger than that of the signal. In those environments, the envelope detector fails to work.

This motivated us to implement CSS downlink on backscatter devices to combat interference and extend the receiving range. However, as we have introduced in Sec 2.2, the power-limited backscatter device is unable to afford the power consumption of carrier generation and signal down-conversion. Therefore, implementing chirp de-spreading seems to be infeasible on backscatter devices.

3.1 Passive Chirp De-spreading Design

With an understanding of the major cause of high power consumption in conventional chirp de-spreading, preliminarily,

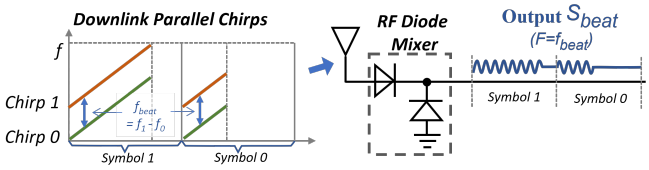


Figure 5: **Transmitted two chirps and circuit for passive chirp de-spreading**.

we explore uploading the task of carrier generation and down-chirp generation to the gateway. Specifically, we arrange the gateway to transmit two wide-band chirp signals into the air. By multiplying one chirp with another one, the receiver can achieve down-converting and de-spreading without the requirement of a locally generated carrier and the down-chirp signal.

The transmitted chirp signals are shown in Fig. 5, which are denoted by 'chirp_0' and 'chirp_1', respectively. They have the same chirp rate (i.e., frequency increasing rate whose unit is Hz/s), and hence there is a constant frequency difference between them. Unlike narrow-band signals, such as ASK signals or BFSK signals, these two wide-band chirp signals will not be covered by narrow-band interference.

These two chirps transmitted to the receiver can be written as:

$$chirp_0(t) = cos\left[2\pi\left(f_0 t + 1/2ut^2\right)\right] \quad (1)$$

$$chirp_1(t) = cos\left[2\pi\left(f_1 t + 1/2ut^2\right)\right] \quad (2)$$

where u is their chirp rate; f_0 and f_1 are their initial frequencies, respectively.

The receiver de-spreads these two chirps by multiplying them with each other using a passive, non-linear diode mixer. In practice, we use two diodes so as to receive the signal from the positive and negative terminals of the antenna. The result of signal mixing is expressed as:

$$\sum_{m=-\infty}^{+\infty}\sum_{n=-\infty}^{+\infty} a_{mn} cos\{2\pi([m(f_1+ut)+n(f_0+ut)]t\} \quad (3)$$

According to the property of non-linear mixing, we ignore the high-order harmonics as they have very low energy levels. Thus, we only consider the low-order harmonics, e.g., the first, second and third-order harmonics. More specifically, we can obtain a high-frequency signal and a low-frequency beat signal (denoted by S_{beat}) whose frequency is equal to the frequency difference $f_1 - f_0$. Instead of using FFT to detect the single tone, we use the LC resonator introduced in the next section to filter out the high-frequency signal.

Therefore, the only signal that can remain is the S_{beat}. If the receiver detects the presence of the S_{beat}, it can infer that the gateway is transmitting that two chirps. Moreover, by measuring the duration of the S_{beat}, the receiver can know the transmitting duration of the two chirps. This further allows

us to encode different symbols by varying this transmitting duration, as shown in Fig. 5.

3.2 Interference Resilience Analysis

From the measured spectrum shown in Fig. 4(b), it can be seen that the interference can be classified into two categories, i.e., narrow-band interference, and the spread spectrum signals. We consider signals using ASK, FSK and PSK modulation to be narrow-band signals, because they are composed of one or multiple narrow-band signal components with fixed frequencies.

Narrow-band interference: As we have introduced, narrow-band signals will be spread into the chirp's band in de-spreading process. More specifically, considering that there is a narrow-band interference with a fixed frequency f_n passes through the mixer with the two chirp signals, the resulting signals contain multiple components, including multiple high-order harmonics, the spread interference, and the envelope of all these signals. For example, if the interference signal is mixed with chirp_0 (Eq. 1), the generated signals are

$$\sum_{m=-\infty}^{+\infty}\sum_{n=-\infty}^{+\infty} a_{mn}cos\left\{2\pi\left[m(f_0+ut)+nf_n\right]t\right\}. \qquad (4)$$

where the high order harmonics and the spread interference with increasing frequency term (i.e., the ut in Eq. 4) cannot pass the following filter (i.e., the LC resonator). As for the envelope, it can hardly pass the filter unless it has the same frequency of $f_1 - f_0$. We argue that this case can hardly occur in practical scenarios.

Moreover, we should note that we also encountered strong out-of-band interference in our experiment (whose strength is about $12dB$ higher than that of chirp signals), and the passive mixer can only output the interference envelope instead of the desired beat frequency. This can be fixed with a SAW (Surface Acoustic Wave) filter.

Spread spectrum interference: Theoretically, our design can be interfered with by a specific type of chirp spread spectrum interference. This chirp signal should have the same chirp rate as the chirp signals transmitted by the gateway. Besides, the frequency difference between this interference chirp and one of the chirp signals transmitted by the gateway should be equal to $(f_1 - f_0)/2^n$ $(where\ n = 0,1,2,3...)$. According to our experiment, facing this type of interference, its anti-interference performance is similar to that of an envelope detector without an anti-interference design.

4 LNA-free Signal Magnification

Amplifying the received signals can enhance the receiving sensitivity and improve the receiving range. Receivers typically employ the LNA to achieve this. As we have introduced in Sec. 2.1, even the latest LNAs we can find, still consume several mA to hundreds of mA of external current. Hence, we are motivated to propose a signal magnification method with ultra-low power to replace the amplifier.

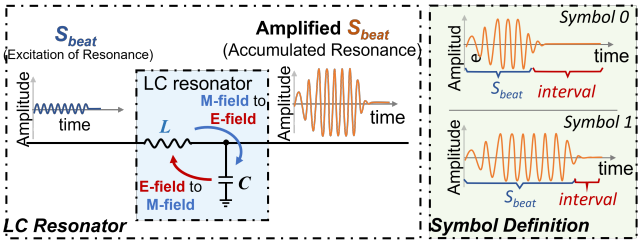


Figure 6: **Block Diagram of LNA-free Magnification.**

To achieve this, we explore accumulating the energy of the signal to implement magnification instead of consuming external current. We leverage an LC resonator to realize a container to accumulate energy. Conventionally, the LC resonator is used as high-frequency RF elements, e.g., on RF receiving antennas. It relies on tuning antenna impedance [3] to improve signal receiving efficiency at a particular frequency [1, 50]. Differing from these conventional uses, $\mu Mote$'s LC resonator works at relatively low frequencies (tens of kHz) and is used to accumulate the signal energy to magnify signal amplitude. Besides, when the signal energy is being accumulated, the signal's physical characteristics (e.g., frequency, amplitude, and phase which are used to carry data) will be destroyed. Thus, we should address the issue of information preservation during energy accumulation.

4.1 Magnifying by Accumulating Energy

In $\mu Mote$, the LC resonator is placed after the passive-CSS and consists of an inductor in series and a shunt capacitor, as shown in Fig. 6. The energy in this circuit will be alternatively transformed in the manner of magnetic and electric fields between the capacitor and inductors. The principle behind is the energy entered the magnetic field of the inductor can be used to charge the electric field on the capacitor plates, and vice versa. Similar to a pendulum or a swing, this "energy reciprocating" has its natural resonance frequency, which can be written as:

$$f_{res} = \frac{1}{2\pi\sqrt{L \cdot C_{res}}}$$

where C_{res} is the capacitance value of the capacitor in the resonant circuit.

The input signal of LC circuit is S_{beat}, the demodulated beat signal output by the passive-CSS circuit. If S_{beat} has the same frequency as $f_{resonant}$, its energy will be accumulated in the form of resonance for magnification, as shown in Fig. 6. Thus, the resonator acts as a magnifier and band-pass filter which only magnifies the voltage of S_{beat} whose frequency is f_{res}.

Table 1: Candidate *LC* resonators for different symbol rates

Symbol Rate	L	C_{res}	R_{inner}	Q	T_{dis}	V_{out}/V_{in}
$1\sim2kbps$	$0.05H$	$270pF$	273Ω	49	$600\mu s$	$40\times$
	$0.05H$	$270pF$	384Ω	35	$250\mu s$	$23\times$
	$0.05H$	**$270pF$**	**734Ω**	**18**	**$100\mu s$**	**$15\times$**
$5kbps$	$0.05H$	$470pF$	$1.1k\Omega$	9	$70\mu s$	$10\times$
	$0.05H$	**$470pF$**	**$1.3k\Omega$**	**7.9**	**$30\mu s$**	**$7\times$**

4.2 Preserve the Information

Unfortunately, this magnification scheme based on *LC* circuit can destroy the physical characteristics of the signal, e.g., the phase, the frequency and the amplitude of the signal. For instance, symbols in PSK (Phase-Shift-Keying) or ASK (Amplitude-Shift-Keying) manner can be destroyed as the phase and amplitude are instantaneous physical characteristics that cannot be preserved in an energy container. Therefore, we need to retain the encoding information in the energy accumulating.

Inspired by PIE (Pulse Interval Encoding), we leverage the duration of the magnified S_{beat} signal (i.e., S_{beat}'s energy) to represent the binary value of a symbol. The symbols representing binary bit "1" and binary bit "0" are shown in the right portion of Fig. 6. For simplicity, we refer to them as symbol "1" and symbol "0" respectively. Besides, in each symbol, there is an interval between two consecutive symbols. As the duration of S_{beat} in symbol "1" is longer than that in symbol "0", the symbol "1" contains more energy than symbol "0". The energy difference can further be distinguished by energy integration schemes introduced in Sec. 5.

To separate symbols, each symbol's energy should be released from *LC* circuit to avoid affecting the next symbol. In our scheme, we can separate symbols by assigning an interval between two symbols, as shown in the right portion of Fig. 6. In detail, during the symbol, the *LC* circuit has energy input and keeps resonating. During the interval, the energy of the current symbol stored in the *LC* resonator should be discharged by the circuit's internal resistance, or controlled discharging, see Sec.8.

4.3 Symbol Rate vs. Magnification Performance

From the symbol definition, we can see that a higher symbol rate requires a shorter interval, and fast discharging of *LC*. If the *LC* is not discharged completely, the residual energy will make the next symbol misidentified. Thus, we need a small Q factor to ensure sufficient discharging. On the other hand, a small Q means the weak magnification capability of the *LC* circuit. There is a trade-off between the magnification times and the symbol rate.

The Q factor is defined as

$$Q = 2\pi \frac{E_{store}}{P_{diss} T}$$

, where P_{diss} is the average dissipation power during a resonant period T, and E_{store} is the energy currently stored in *LC*. Hence, a higher Q factor means more energy can be preserved in *LC*, or more time to release the stored energy, i.e., discharge the capacitor in *LC*. On the other hand, the Q factor of the *LC* resonator can be calculated as:

$$Q = \frac{1}{R_{inner}} \cdot \sqrt{\frac{L}{C_{res}}}$$

where R_{inner} is the inner resistance of the *LC* resonator. From the two formulas, it can be seen that we can hardly calculate or measure the time constant of an *LC* resonator [48] to determine the required time interval for discharging. Further, it is also difficult to find COTS "*L*" and "*C*" components with the exact optimal value in practice. Hence, we empirically study *LC* resonators of different Q factors with available COTS components, and practically measure their discharging time.

We present the parameters of candidate *LC* resonators for different symbol rates in Table 1. In the table, "V_{out}/V_{in}" means the magnification ratio that the resonator can magnify the voltage of input S_{beat}, and the "T_{dis}" is the measured discharging time. For example, for a symbol rate of $2kbps$, we can choose parameters of the third resonator, whose discharging time is more than $100\mu s$. It means that the time interval of symbols should be more than $100\mu s$ also.

5 Low-power Decoding

With LNA-free magnification, we can get the magnified signals as well as symbols. The next step is to extract binary bits from the symbols. Mostly, ADC (Analog-to-Digital Conversion) is used for symbol decoding, but ADC will introduce high power consumption due to two reasons.

First, to ensure decoding accuracy, the ADC has to perform sampling tens of times when decoding each symbol. Based on Nyquist's theorem, the sampling rate should be twice the frequency of S_{beat}. Second, every single sampling operation of ADC consists of several complicated steps, e.g., extracting analog samples from a continuous signal, amplifying those analog samples to improve converting accuracy, integrating the amplified samples, and converting integration results to digital values. All aforementioned steps are controlled by an MCU or custom IC. As a result, even a well-known energy-efficient ADC [49] will consume more than a couple of 100 μW, which is two to three times higher than the total power consumption of our *µMote* including signal amplifying and symbol decoding.

In *µMote*, we carefully make use of the signal duration of S_{beat} and interval to represent symbols, as shown in Fig. 6. For

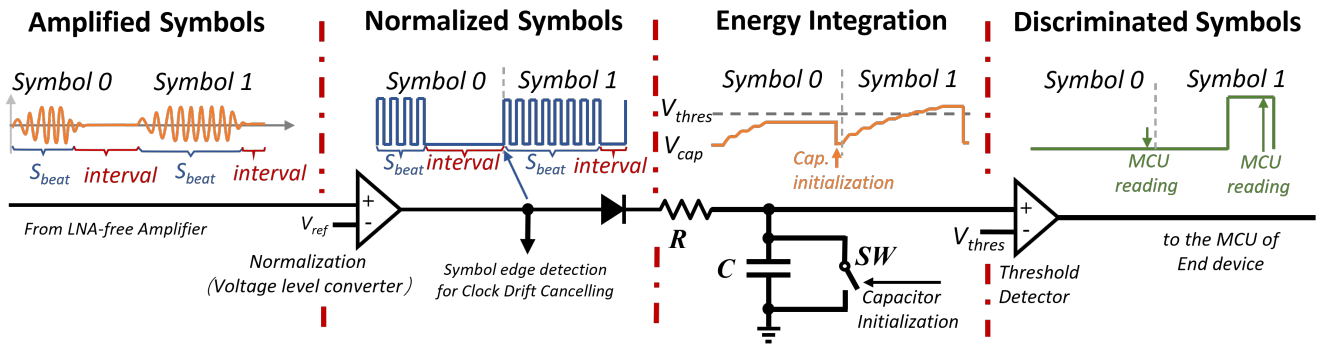


Figure 7: **Procedures for energy integration-based decoding.**

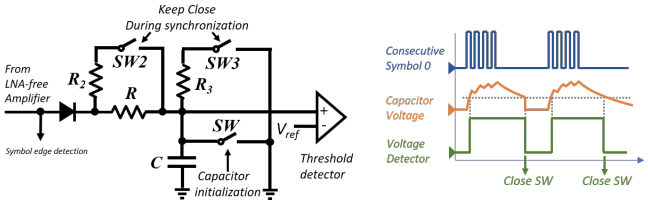


Figure 8: **The practical decoding circuit for symbol synchronization.**

example, symbol "1" has a shorter interval and longer signal duration than symbol "0", thus the energy of symbol "1" is higher than symbol "0". Different symbols to an *energy integrator* can result in different energy levels which can be used to discriminate symbols. Thus, a simple energy integrator can be employed to replace the ADC for decoding each symbol with only a single integration operation and low power consumption. *RC* (Resistance Capacitance) circuit is a typical realization of energy integrator as illustrated in Fig. 7. In order to save power, we can suppress the internal current to several μA with high resistance, and reduce the power consumption of integration to about $10\mu W$. As shown in Fig. 7, the signal S_{beat} can charge the capacitor "C" through the resistor "R" upon decoding. Because of the longer signal duration, symbol "1" can charge the capacitor to a higher peak voltage, which can be detected and converted to "1" by a threshold detector, and vice versa.

Apart from the basic design, there are several practical problems that need to be addressed.

Dynamic signal strength. In practice, the charged peak voltage in capacitor "C" is not only determined by S_{beat}'s duration, but also affected by S_{beat}'s strength. For example, if the receiver is placed close to the gateway, the incident signal power is high and hence the time to charge the capacitor would be short. In this case, a symbol "0" might be identified as a symbol "1". Conversely, if the receiver is placed far away, a symbol "1" may also be incorrectly identified as a symbol "0". To address this problem, we normalize S_{beat}'s amplitude after magnification. Specifically, we place a voltage level converter after the *LC* circuit and before the *RC* integrator, normalizing the amplitude of S_{beat} to a predefined reference voltage, as illustrated in the left part of Fig. 7. The detailed implementation of this circuit is introduced in Sec. 6 and its power consumption is discussed in Sec. 7.4.

Capacitor discharging. The binary value of each symbol is determined by how much S_{beat} charges the capacitor C. So the capacitor should be discharged fast and completely before the arrival of the next symbol to avoid inter-symbol interference. Intuitively, we can leverage the self-discharging of the capacitor. However, this is quite slow and thereby significantly slowing down the symbol rate. To address this problem, we add an NMOS switch to connect the positive plate of the capacitor to Ground (GND). At the end of each symbol, we make the capacitor directly connected to the Ground for fast discharging.

Symbol synchronization. To decode the symbol, the MCU should know exactly the end time of a symbol and read the output of the threshold detector at the time. Otherwise, the MCU may get the wrong integration results and misidentify symbols. In other words, it should be synchronized with the symbols sent by the gateway. Intuitively, we can make the gateway transmit preamble symbols for synchronization. However, if it is not synchronized, the decoding circuit and MCU cannot decode any symbols.

To achieve synchronization, the decoding circuit is programmed to switch to another structure which can detect preamble symbols. Specifically, two resistors and two NMOS switches are added to the circuit, as shown in the left portion of Fig. 8. Upon receiving the preamble, $SW2$ and $SW3$ are closed by the MCU so that resistors R_2 and R_3 are connected to the circuit. At this time, R and R_2 form a smaller resistor, accelerating the charging speed of the capacitor. With these resistors, a symbol "0" can rapidly charge the capacitor to the threshold of the detector, and the charged voltage can be released via R_3 and $SW3$ before the next symbol "0" arrives, as illustrated in Fig. 8. Thus, we can use a series of symbol "0" as the preamble, and the detector will output corresponding high voltage levels (the green line shown in Fig. 8) to inform the MCU that the preamble has arrived. The reason we do not use symbol "1" is that the duration of its S_{beat} signal is

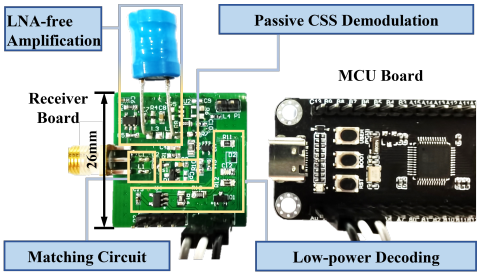


Figure 9: ***µMote* prototype.**

too long, so even with the presence of R_3, the voltage in the capacitor cannot be fully released before the next symbol "1" arrives.

Clock drift canceling. The clock on the backscatter device drifts over time, leading to synchronization and decoding errors. To address this, we present a method to cancel the clock drift. At the end of a symbol, we detect the first rising edge of the following symbol, as shown in Fig. 7. This rising edge represents the exact starting time of a symbol, which can be used to calibrate the clock on the device.

Power consumption. The decoding process requires three operations at the end of a symbol, i.e., reading the threshold detector, discharging the capacitor, and canceling the clock drift. These operations can be accomplished with less than 100 instructions, which incurs $< 1\%$ duty cycle and several μW power consuming. The necessary timing circuit to wake up the MCU and control the duty cycle is included in the MCU hardware, and the MCU only consumes nW-level power in low power mode [49].

Power optimization of *RC* circuit. We reduce the charging current to save power. The charging current can be written as

$$I_{charge} = \frac{V_{ref} \cdot exp(-t/RC)}{R}$$

, where V_{ref} is the charging voltage on the capacitor and is equal to the reference voltage in S_{beat} normalization. To suppress I_{charge}, we increase R and reduce C, and keep the product RC unchanged. For a symbol rate of $5kbps$, the proper values of R and C are $220k\Omega$ and $330pF$, and the power consumption of the RC circuit is $12\mu W$. The concrete evaluation results can be found in Sec. 7.

6 Implementation

We prototype $\mu Mote$ with COTS components on a $2.4cm \times 2.6cm$ PCB, as shown in Fig. 9. We introduce the hardware implementation as follows.

Matching circuit and passive-CSS demodulation: The matching circuit is composed of a series capacitor and a shunt inductor, as RF signals can easily pass through the capacitor but cannot go through the inductor to the Ground (GND). The matching circuit achieves reflection loss lower than $-20dB$ in the $913-916MHz$ frequency band, which is wide enough

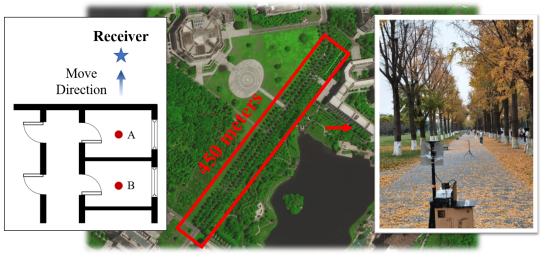


Figure 10: **Indoor and outdoor experiment environment.**

for receiving the two chirps. We use Skyworks SMS7630-005LF RF diodes as the passive mixer for down-conversion and de-spreading.

LNA-free Magnification: We implement $\mu Mote$ prototypes of three symbol rates. For $1kbps$ and $2kbps$, we choose the third resonator shown in Table 1. For $5kbps$, we choose the fifth one. The corresponding resonant frequencies of these two are $43kHz$ and $32.8kHz$. Before the LC resonator, we place a capacitor in series for DC-AC conversion, as the LC resonator requires AC input.

Normalization circuit: We employ a voltage level conversion circuit to normalize the magnified S_{beat}. In practical realization, we have two choices: (i) NCS2200, a comparator which has a relatively low power consumption of $18\mu W$ and can work on passive backscatter devices [64].(ii) MAX9914, which is actually a power-efficient operational amplifier which can also work as a voltage level converter. It can provide an extra sensitivity gain of $8dB$ with a power consumption of $51.17\mu W$, therefore further extending the receiving range. We use MAX9914 to realize the $\mu Mote$ prototype to explore the maximum range of our design. Besides, we use NCS2200 to realize a low power version of $\mu Mote$, which is named as "$\mu Mote-$" (i.e., "*µMote minus*"). Compared with $\mu Mote$, $\mu Mote-$ loses $8dB$ sensitivity but gains $33.17\mu W$ power saving.

RC Decoding circuit: We employ different combinations of R and C to build RC circuits for different symbol rates. The detailed values are introduced in Sec. 7.4. After the RC circuit, we use an S-1000C16 and an NLSV1T244 to build the threshold detector with a total power consumption of $1\mu W$. And we use GPIO ports of STM32L476 MCU to record the decoded data and control the circuit via three DMG2302UK NMOS switches. The MCU runs at a 1% duty cycle as we introduced in Sec. 5.

7 Evaluation

In this section, we first introduce the experiment setup in Sec. 7.1. Then we introduce benchmarks for subsequent evaluation in Sec. 7.2. Further, we evaluate communication ranges of $\mu Mote$ and benchmarks in Sec. 7.3. The power consumption comparison among our design and benchmarks, as well as $\mu Mote$ power breakdown, are illustrated in Sec. 7.4. Then we introduce interference experiments using interference signals

recorded in practice environments in Sec. 7.5.

7.1 Experiment Setup

We leverage a USRP-2922 to build the gateway's transmitter with an ADL5605-EVALZ [9] RF amplifier. Thus, the gateway can transmit up to $30dBm$ power, near the ImpinJ R420 RFID reader. The two downlink chirp signals are in $915MHz$ ISM band and their bandwidths are $1MHz$. The frequency difference (i.e., the frequency of S_{beat}) between the two chirp signals is set to the resonant frequencies of LC circuit, i.e., $43.3kHz$ for $1kbps$ and $2kbps$ symbol rates, and $32.8kHz$ for $5kbps$.

7.2 Benchmarks

- WISP5 [45]: a very widely known backscatter tag with an envelope detector-based receiver and an ADC for sampling. It has no amplification and anti-interference design. To favor the communication range test for the WISP5, we connect it with a $1.8V$ power supply module, instead of using its RF energy harvester which has an energy harvesting range of only several meters.

- Saiyan [17]: the very recent work that achieves LoRa symbol receiving with an envelope detector. Saiyan converts chirp symbols to amplitude-modulated signals via a SAW (Surface Acoustic Wave) filter, and uses an LNA for amplification. Following the literature [17] and the BOM (Bill of Material) list of their project document [16], we implement Saiyan hardware with a TQP3M9008 LNA and an OPA810 operational amplifier for signal amplification.

- Passive-DSSS [31]: an envelope detector-based design which employs two envelope detection channels for DSSS spreading codes to address interference. Besides, passive-DSSS employs a TLV9001 operational amplifier of $180\mu W$ to improve sensitivity and range.

Those benchmarks use their original encoding mechanisms, respectively, i.e., PIE for WISP5, chirp symbols for Saiyan, and DSSS spreading codes for passive-DSSS. For comparison, we manually set the symbol rate of those benchmarks to $1kbps$.

7.3 Communication Range

In this section, we first evaluate the receiving sensitivity of $\mu Mote$ and benchmarks using laboratory tests. Then we conduct outdoor experiments to evaluate their practical communication ranges in LOS (Line-of-Sight) scenario, i.e., a roughly straight road on campus, as shown in Fig. 10. Finally, we evaluate the communication range and packet throughput of

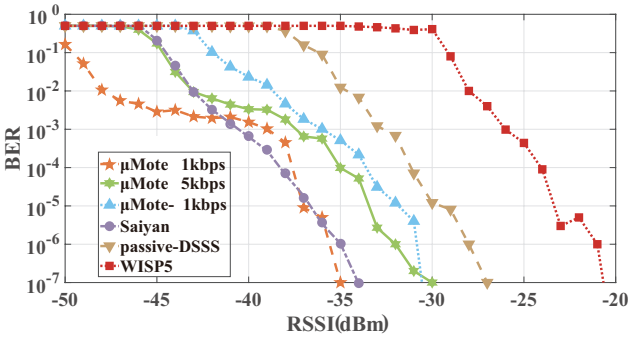


Figure 11: **Receiving sensitivity of $\mu Mote$ and benchmarks.**

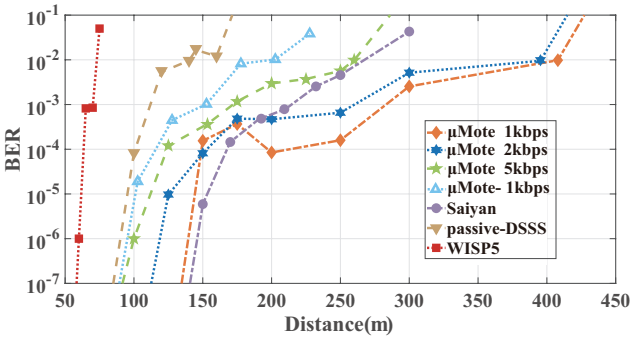


Figure 12: **Communication range in the outdoor LOS scenario.**

$\mu Mote$ in the NLOS (Non-Line-of-Sight) scenario. The NLOS experiment field is illustrated in the left portion of Fig. 10.

Receiving Sensitivity: Without interference, the receiving sensitivity theoretically determines the receiving range. In experiments, we measure the receiving sensitivity as the lowest received signal strength with a corresponding physical layer Bit Error Rate (BER) lower than 1%. To precisely control the signal strength, we connect the transmitter (i.e., USRP) and receiver with an RF cable and employ one or two $30dB$ RF attenuators in the cable. Thus, the receiving signal strength can be precisely controlled by setting the transmitting signal strength and the number of employed attenuators. Besides, to ensure accuracy, we leverage a professional signal strength meter [28] to calibrate the transmitting power of USRP. For each round, we measure the BER by sending $1,280,000$ bits.

The measured receiving sensitivities are shown in Fig. 11. It can be seen that the $\mu Mote$ has the best receiving sensitivity of $-48dBm$ at $1kbps$ symbol rate. According to theoretical estimation [10], its receiving range can reach 500 meters with the maximum transmitting power of our transmitter ($30dBm$). The receiving sensitivities of WISP5, replicated Saiyan, and passive-DSSS are $-28dBm$, $-43dBm$, and $-35dBm$. Similarly, we estimate their theoretical receiving ranges are $80m$, $250m$, and about $160m$, respectively. The sensitivity of $5kbps$ $\mu Mote$ prototype is $-43dBm$, due to the relatively low Q factor of LC circuit. The future solution to this is discussed in Sec. 8. Besides, $\mu Mote-$ has a sensitivity of about $-38dBm$, which is similar to Saiyan and better than passive-DSSS.

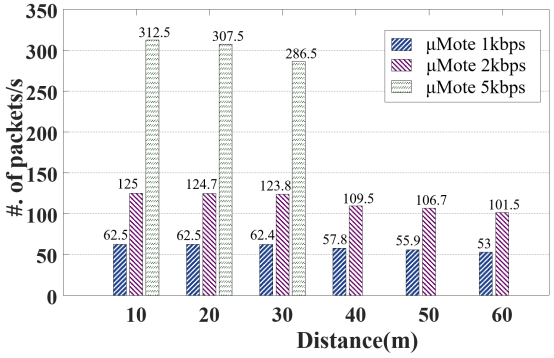


Figure 13: **Throughput of *µMote* for different bit rates when twin-CSS signal penetrates one wall.**

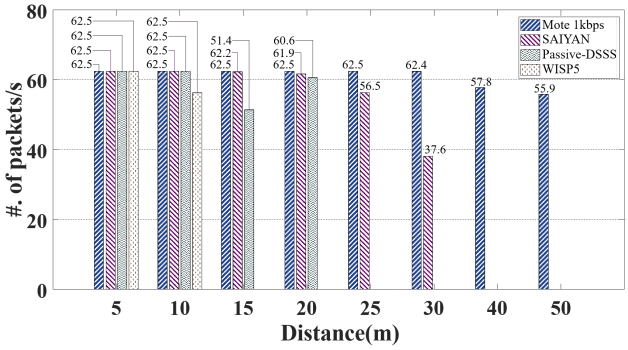


Figure 14: **Throughput of *µMote* 1*kbps* and benchmarks when the signal penetrates one wall.**

LOS Experiments: To evaluate the practical communication ranges of *µMote* and three benchmarks, we conduct outdoor experiments in LOS scenarios. The USRP transmits 30*dBm* RF signals through a 6*dBi* gain antenna, while each of the receivers uses a receiving antenna with 3*dBi* gain. We test *µMote* at different symbol rates of 1*kbps*, 2*kbps*, and 5*kbps*. Each BER value is obtained by counting misidentified bits in $1,280,000$ received bits.

The practical receiving ranges are shown in Fig. 12. At symbol rates of 1*kbps* and 2*kbps*, *µMote* achieves a similar receiving range of 400 meters, because at these symbol rates, the *LC* circuit has the same high *Q* factor and similar magnification performances. We do not reach the estimated 500*m* range, because we encountered strong in-band interference that came from unknown wireless devices and exceeded the anti-interference capability of our prototype. The practical ranges of WISP5, Saiyan and passive-DSSS are 70*m*, 250*m*, and 140*m*, which are similar to our estimations. We think that the reason is at these ranges, they do not face interference signals significantly stronger than signals transmitted by USRP. And the range of *µMote* 5*kbps* is 260*m* which is an expected result considering its sensitivity. Besides, *µMote*− achieves a maximum range of about 200*m*.

NLOS Experiments: We also measure the practical receiving ranges of *µMote* and benchmarks in the indoor NLOS scenarios, as a number of IoT devices are deployed in doors

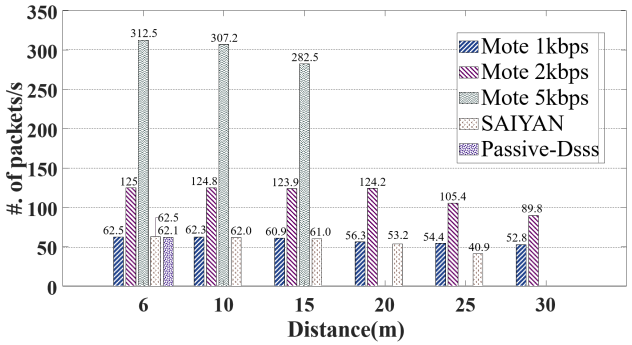


Figure 15: **Packet throughput of *µMote* and benchmarks when the signal penetrates two walls.**

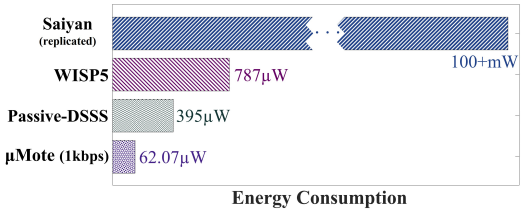


Figure 16: **The power consumption of *µMote* prototype and benchmarks.**

and signals have been attenuated by walls. The dots *A* and *B* represent where the gateway's USRP transmitter is located, and the star indicates where the receiver is placed. When the gateway is placed at the *A*, the transmitted signals have to penetrate one wall. When the gateway is at the dot *B*, the signals penetrate two walls. The star can be moved far or close to the transmitter to make different downlink ranges. We test three symbol rates with *µMote* and one symbol rate with benchmarks, and record each of the error bits in $1,280,000$ received bits.

Leveraging the recorded error bits, we calculate the packet throughput at different ranges, which can better represent the communication performance in actual environments. For example, even if there is only one error bit, the packet will become unrecognizable and is therefore discarded. The packet is defined as a 16-bit string, and thus the *packet throughput* refers to the number of received packets being correctly recognized. The test data is a packet which is sent for $10,000$ times in a loop.

When there is one wall, the measured packet throughput of *µMote* at different symbol rates are shown in Fig. 13. With 60*m* downlink range, the *µMote* receiver achieves 53 packets/*s* at 1*kbps*, and 101.5 packets/*s* at 2*kbps*. The maximum range for 5*kbps* is 30*m* and the corresponding throughput is 286.5 packets/*s*. We can learn from this result that for *µMote*, a *LC* resonator with high *Q* factor is crucial to achieving long range.

We also measure the packet throughput of benchmarks when their downlink signals penetrate one wall, and then compare them with *µMote* at 1*kbps*. The results are plotted

Table 2: Power breakdown of two prototypes

Modules	Despreading	Magnification	Regulator	*RC* Decoding (1*kbps*)	MCU (1% duty cycle)	Total
μMote	0μW	0μW	51.17μW	3.4μW	7.5μW	62.07μW
μMote−	0μW	0μW	18μW	3.4μW	7.5μW	28.9μW

Table 3: Parameters of *RC* decoding circuit for different symbol rates

Bit rate	***R* value**	***C* value**	***E*/symbol**	**Power**
1kbps	910$k\Omega$	330pF	2.40nJ/bit	3.4μW
2kbps	470$k\Omega$	330pF	2.36nJ/bit	3.72μW
5kbps	200$k\Omega$	330pF	2.51nJ/bit	13.55μW

in Fig. 14. Among these benchmarks, Saiyan has the longest range of 30m with a packet throughput of 37.6 packets/s, this range is half the range of *μMote* (i.e., 60m at 1$kbps$).

When there are two walls between the gateway and receivers, the receiving ranges are further decreased, as shown in Fig. 15. At the range of 30m, *μMote* has acceptable throughput values of 52.8 packets/s at 1$kbps$ and 89.8 packets/s at 5$kbps$. Saiyan and passive-DSSS can receive at the ranges of 25m and 6m, with packet throughput values of 61.6 packets/s and 62.1 packets/s, respectively. WISP5 cannot receive its downlink signal.

7.4 Power Consumption

μMote vs. Benchmarks: To demonstrate the power-efficiency of our design, we measure the practical power consumption of *μMote* prototypes and benchmarks using a sub-μA-level power monitor [52]. The results are plotted in Fig. 16. We can see that the total power consumption of *μMote* (1$kbps$) is 61.07μW, which is the lowest. WISP5 and passive-DSSS have a power consumption of several hundreds of μW, as we take the power consumption of ADC sampling and the MCU for ADC control into account. We replicated Saiyan prototype according to the BOM extracted from their project files [16, 17]. At its maximum receiving range, the power consumption of the Saiyan prototype is 446mW. We believe the cause of this high power consumption is their use of two amplifiers [21, 40].

μMote power breakdown: Table 2 shows the power breakdown of the *μMote* prototype. The de-spreading circuit and magnification circuit consume zero power as they are driven by the received signal. For different symbol rates, the corresponding power consumption of *RC* decoding circuit (including the threshold detector) is illustrated in Table 3. It can be observed that at different symbol rates, the energy budget for decoding one bit remains roughly unchanged. The cause is that the energy budget is related to capacitance. The MCU average power consumption is about 7.5μW with 1% duty cycle.

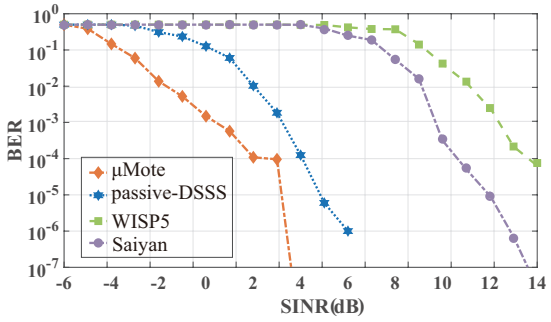


Figure 17: **Receiving BER of *μMote* and benchmarks under LoRa interference.**

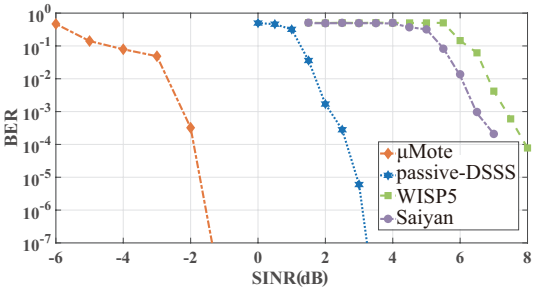


Figure 18: **Receiving BER of *μMote* and benchmarks under RFID interference.**

7.5 Interference Resilience

We evaluate the interference resilience of *μMote* under interference in 900MHz ISM band, such as RFID and LoRa. We use the interference-resilient receiver, i.e., passive-DSSS, as benchmarks. We also evaluate WISP5 and Saiyan for comparison, which are not interference-resilient. We choose the impinj R420 RFID reader and EBYTE E32-915T30S Lora transceiver as the interference source. Then we use USRP to record practical signals transmitted by these two devices, so we can manually amplify or attenuate the recorded signals to make interference with different strengths. Finally, we leverage RF cables to transmit twin-CSS signals as well as those interference signals to *μMote* receiver to measure receiving BER values. Both *μMote* receiver and passive-DSSS receiver are set to 1$kbps$ symbol rate. The bandwidths of RFID and LoRa interference are set to 100kHz and 500kHz, respectively.

Fig. 17 and Fig. 18 show the BER measurement under LoRa and RFID interference signals. The results demonstrate that the *μMote* prototype can operate under negative SINR caused by RFID or LoRa interference. In detail, at BER of

1%, the corresponding SINRs caused by LoRa or RFID signal are about $-1dB$ and $-2.7dB$, whereas at BER of 0.1% the SINRs caused by LoRa or RFID signal are about $-1dB$ and $-2.2dB$. We can learn *µMote* can operate under negative SINR caused by LoRa and RFID interference. Passive-DSSS has SINR improvement over classic WISP5 which has no anti-interference design, but it cannot work under negative SINR.

8 Discussion and Limitations

Accessing multiple devices or gateways: To date, *µMote* cannot support concurrent transmissions from the gateway to multiple backscatter devices, as *µMote* has no frequency division or other multiple access designs. A feasible solution is to employ time division on the MAC layer, similar to RFID protocols. In addition, *µMote* fails to work in the face of the hidden node problem, which occurs when multiple gateways want to talk with a certain receiver, but they are unaware of each other's existence.

Strength loss: The gateway transmitter needs to split its power across both chirps, and therefore it may require more transmitting power than it would have been needed to transmit to a device at the same distance using a conventional active receiver. Although *µMote* mixes the two chirps, the generated harmonics will also result in a loss of signal energy.

Spectrum efficiency: The bandwidth consumed in our design is about $1.04MHz$ ($1MHz$ chirp and $43kHz$ S_{beat}). Most spectrum bandwidth is spent on spectrum spread to resist interference. Compared with LoRa receivers, our *µMote* consumes similar spectrum bandwidth but achieves power consumption three orders of magnitude lower than LoRa.

Improving symbol rate: As the trade-off introduced in Sec. 4, in this paper, we cannot achieve the highest symbol rate and best magnification performance simultaneously. Improving the magnification performance requires increasing the interval time for discharging the *LC* resonator, while improving the symbol rate requires decreasing the interval time. Hence, we explore a discharging mechanism to the *LC* resonator. Specifically, when the symbol synchronization is completed, we can discharge the *LC* resonator at the middle (to discharge the energy of symbol "0") and the end of a symbol. Thus, the energy in *LC* can be discharged more rapidly to allow a higher symbol rate, even for *LC* of very high *Q* factor to gain high magnification performance. Moreover, we also plan to explore whether we can add more threshold detector modules ($< 1\mu W$ per module) with different thresholds, and then the symbols can be quantified into more characters, thereby increasing the bit rate.

9 Related Work

Diode-based mixer. Similar to *µMote*, there are two existing works [11, 42, 43] that use the diode mixer [4] to mix two RF signals and achieve low-power subcarrier generation or very high-speed receiving. The differences between our work and them are three-fold. The first difference is waveform modulation. In *µMote* design, we employ Chirp Spread Spectrum (CSS) signals as downlink signals, and thus our work can combat interference and even work under negative SINR. The second difference lies in *µMote*'s system design. The passive chirp de-spreading scheme should be used in conjunction with the *LC* resonator which acts as a magnifier and filter, otherwise the signal strength loss due to chirp mixing would decrease the receiving range, and the receiver cannot filter or detect the generated beat signal. The third difference is our design has the benefit of a relatively long receiving distance.

Long-range backscatter communication. To resist interference and improve communication range in low-power IoT systems, tremendous backscatter innovations [23, 35, 37, 51, 54, 55] leverage CSS modulation on LoRa, extending the uplink range to more than one kilometer. Nevertheless, these designs cannot receive spread spectrum signals on the downlink, as spread spectrum signal demodulation and de-spreading involve high-power local carrier generation and computing-intensive correlation for synchronizing local de-spreading codes or de-spreading signals. This fact causes unbalanced communication ranges and robustness on uplink and downlink. Moreover, tunnel diode can be employed in backscatter transmitters, further increasing the communication range of backscatter devices [56].

Saiyan [17] employs a SAW filter to re-shape the envelope of chirp signals, enabling an envelope detector to receive LoRa signals on the downlink. But this method does not de-spread chirps so it cannot gain the communication range and robustness improvements of CSS, and instead, it has to use LNA to boost range. Passive-DSSS [31] receives both spreading codes and synchronized de-spreading codes from the transmitter, first achieving the DSSS communication on envelope detector-based receivers. Compared with *µMote* which employs a passive mixer to de-spread two parallel chirps, Passive-DSSS relies on envelope detectors to receiver signal, and hence it cannot receive CSS signals and work under negative SINR. Moreover, Passive-DSSS does not contain any low-power signal magnifying scheme (e.g., the LC resonator) and low-power ADC-free decoding scheme, resulting in a relatively limited receiving range and relatively high decoding power.

Signal magnification techniques. The most commonly used signal amplification means is the LNA. As we introduce in Sec. 2.1, even the latest commercial LNA ICs consume more than several *mW* of power. To magnify signals with low power, recent researches leverage the principle of *Voltage transforming* or *Impedance tuning* to get signals with mag-

nified voltages. For example, XSHIFT [43] leverage a coil voltage transformer to magnify the voltage of demodulated signals by 5 times. Besides, *LC* circuits are also used to boost the voltage of received RF signal by adjusting the antenna impedance [1, 50]. However, these two methods cannot magnify the instantaneous power of RF signals or demodulated signals, so their improvements in receiving sensitivities are relatively limited, compared to our design.

Backscatter communication. Our work is also related to backscatter communication techniques, as *µMote* characteristics have potential benefits existing backscatter devices. For LoRa backscatter devices [23, 35, 37, 51, 54, 55], it can make up for their limited downlink performance in terms of range and robustness against interference, with only μW-level power. For other tremendous backscatter innovations features of high data rate [53], high-throughput [24, 25, 42], robustness [57], large system scale [19], simplified hardware [30, 43, 62], and ambient-signal-compatible [2, 5, 12, 22, 26, 27, 32, 33, 35, 36, 38, 54, 61, 63], *µMote* can provide downlink connection with benefits of interference-resistant, low-power, and cost-efficiency, at the same time.

ADC-free decoding. Similar to *µMote*, RFID tags [10] can decode PIE symbols without ADC. But differing from *µMote*, the PIE decoder of RFID tags can extract the synchronization clock from PIE symbols by simply inverting the PIE waveform (which is illustrated in the citation [28]). However, according to the symbol definition of our design, our ADC-free decoding circuit cannot extract the synchronization clock in the same way as RFID tags. Hence, we have to explore a dedicated synchronization scheme for our decoding circuit.

Wake-up receiver. Due to the power benefit of integrated circuits, envelope detector-based wake-up ICs achieve power consumption of less than $100\mu W$ and good receiving sensitivity of lower than $-60dBm$. These receivers typically has a low power consumption of less than $100\mu W$, and provide sensitivities lower to $-60dBm$ [20, 39, 44]. There are two distinctions between our work and these ICs. First, our design enables interference-resistant communication with μW-level power. Second, employing the LC resonator and the dedicated encoding mechanism can magnify the amplitude of demodulated signals, which is a potential alternative technique for receivers to enhance the receiving sensitivity.

10 Conclusion

In this paper, we systematically investigate the issue of low-power and long-range receiving, the critical bottleneck of communication of backscatter devices. We present *µMote*, an μW-power receiver with 400 meters range. We design and implement the first passive chirp de-spreading method with a simple mixer by offloading high-power carrier de-chirp generating to the gateway. Then we present a novel zero-power magnification method with a *LC* resonant circuit, effectively improving the receiving sensitivity. We propose an energy integration-based decoding mechanism instead of high-power ADC sampling to reduce power consumption. We conduct extensive experiments in different scenarios. The evaluation shows that *µMote* can support up to 400m receiving range with $62.07\mu W$ power consumption at $2kbps$ symbol rate. Compared to benchmarks, *µMote* improves the downlink range by $2.5\times$ to $8.65\times$, with a power consumption reduction of 63.2% to even three orders of magnitude.

Acknowledgments

We sincerely thank our shepherd and all reviewers for their time in reviewing our paper and providing valuable feedback. This paper is partially supported by the National Natural Science Foundation of China under Grant U21A20462; National Key R&D Program of China under Grant 2022YFC3801302; National Natural Science Foundation of China under Grant U22A2031; National Key R&D Program of China under Grant 2021QY0703; National Natural Science Foundation of China under Grants 61872061, 61932013 and 61872285.

References

[1] Mohamed R Abdelhamid, Ruicong Chen, Joonhyuk Cho, Anantha P Chandrakasan, and Fadel Adib. Self-reconfigurable micro-implants for cross-tissue wireless and batteryless connectivity. In *MobiCom'20: Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020.

[2] Ali Abedi, Farzan Dehbashi, Mohammad Hossein Mazaheri, Omid Abari, and Tim Brecht. Witag: Seamless wifi backscatter communication. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 240–252, 2020.

[3] All about circuits. Applications of resonance. https://www.allaboutcircuits.com/textbook/alternating-current/chpt-6/applications-of-resonance/, 2021.

[4] Mark R Barber. Noise figure and conversion loss of the schottky barrier mixer diode. *IEEE Transactions on Microwave Theory and Techniques*, 15(11):629–635, 1967.

[5] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 283–296, New York, NY, USA, 2015. ACM.

[6] Estarija Dan David, Katreena Gabrielle Juntado, Miguel Lorenzo Panagsagan, Anastacia Alvarez, Maria Theresa De Leon, Marc Rosales, John Richard Hizon, and Christopher Santos. Design and implementation of a baseband lora demodulator using de-chirp method. In *2019 International Symposium on Multimedia and Communication Technology (ISMAC)*, pages 1–4, 2019.

[7] Analog Devices. Adl9005. https://www.analog.com/media/en/technical-documentation/data-sheets/adl9005.pdf, 2021.

[8] Analog Devices. Hmc8412chips. https://www.analog.com/media/en/technical-documentation/data-sheets/hmc8412chips.pdf, 2021.

[9] Analog Devices. Evalz-adl5605. https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-adl5605.html$#$eb-overview, 2022.

[10] Daniel Dobkin. *The rf in RFID: uhf RFID in practice*. Newnes, 2012.

[11] Joshua F Ensworth, Alexander T Hoang, and Matthew S Reynolds. A low power 2.4 ghz superheterodyne receiver architecture with external lo for wirelessly powered backscatter tags and sensors. In *2017 IEEE International Conference on RFID (RFID)*, pages 149–154. IEEE, 2017.

[12] Joshua F Ensworth and Matthew S Reynolds. Every smart phone is a backscatter reader: Modulated backscatter compatibility with bluetooth 4.0 low energy (ble) devices. In *2015 IEEE International Conference on RFID (RFID)*, pages 78–85, April 2015.

[13] EPCglobal. Epc radio-frequency identity protocols class-1 generation-2 uhf rfid protocol for communications at 860 mhz - 960 mhz version 1.2.0, 2008.

[14] Inc. Freescale Semiconductor. Practical considerations for low noise amplifier design - white paper. https://www.nxp.com/docs/en/white-paper/RFLNAWP.pdf, 2013.

[15] Jeremy Gummeson, Shane S Clark, Kevin Fu, and Deepak Ganesan. On the limits of effective hybrid micro-energy harvesting on mobile crfid sensors. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 195–208. ACM, 2010.

[16] Xiuzhen Guo. Saiyan-the design and implementation of a low-power demodulator for lora backscatter systems. https://github.com/ZangJac/Saiyan, 2022.

[17] Xiuzhen Guo, Longfei Shangguan, Yuan He, Nan Jing, Jiacheng Zhang, Haotian Jiang, and Yunhao Liu. Saiyan: Design and implementation of a low-power demodulator for {LoRa} backscatter systems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 437–451, 2022.

[18] Xiuzhen Guo, Longfei Shangguan, Yuan He, Jia Zhang, Haotian Jiang, Awais Ahmad Siddiqi, and Yunhao Liu. Aloba: Rethinking on-off keying modulation for ambient lora backscatter. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 192–204, 2020.

[19] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 271–283, USA, 2019. USENIX Association.

[20] Xiongchuan Huang, Pieter Harpe, Guido Dolmans, Harmke de Groot, and John R Long. A 780–950 mhz, 64–146 μw power-scalable synchronized-switching ook receiver for wireless event-driven applications. *IEEE Journal of Solid-State Circuits*, 49(5):1135–1147, 2014.

[21] Texas Instruments. Opa810. https://www.ti.com/product/OPA810/part-details/OPA810IDCKR, 2022.

[22] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 356–369, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Jinyan Jiang, Zhenqiang Xu, Fan Dang, and Jiliang Wang. Long-range ambient lora backscatter with parallel decoding. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 684–696, 2021.

[24] Meng Jin, Yuan He, Xin Meng, Dingyi Fang, and Xiaojiang Chen. Parallel backscatter in the wild: When burstiness and randomness play with you. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 471–485, 2018.

[25] Meng Jin, Yuan He, Xin Meng, Yilun Zheng, Dingyi Fang, and Xiaojiang Chen. Fliptracer: Practical parallel decoding for backscatter communication. *IEEE/ACM Transactions on Networking*, 27(1):330–343, 2019.

[26] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*,

SIGCOMM '14, pages 607–618, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 151–164, USA, 2016. USENIX Association.

[28] Keysight. E4418b epm series single-channel power meter. https://www.keysight.com/us/en/product/E4418B/epm-series-singlechannel-power-meter.html, 2021.

[29] A. Kokkeler, N. B. Molenkamp, and S. H. Gerez. A comparison of fft processor designs. 2013.

[30] Songfan Li, Chong Zhang, Yihang Song, Hui Zheng, Lu Liu, Li Lu, and Mo Li. Internet-of-microchips: direct radio-to-bus communication with spi backscatter. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[31] Songfan Li, Hui Zheng, Chong Zhang, Yihang Song, Shen Yang, Minghua Chen, Li Lu, and Mo Li. Passive {DSSS}: Empowering the downlink communication for backscatter systems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 913–928, 2022.

[32] Yan Li, Zicheng Chi, Xin Liu, and Ting Zhu. Passive-zigbee: Enabling zigbee communication in iot networks with 1000x+ less power consumption. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, pages 159–171, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 39–50, New York, NY, USA, 2013. Association for Computing Machinery.

[34] microsemi. Zl30260-zl30263. https://www.microsemi.com/document-portal/doc_download/136349-zl30260-zl30263-datasheet, 2022.

[35] Rajalakshmi Nandakumar, Vikram Iyer, and Shyamnath Gollakota. 3d localization for sub-centimeter sized devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, pages 108–119, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Aaron N. Parks, Angli Liu, Shyamnath Gollakota, and Joshua R. Smith. Turbocharging ambient backscatter communication. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 619–630, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianshang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. Plora: A passive long-range data network from ambient lora transmissions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 147–160, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Carlos Pérez-Penichet, Frederik Hermans, Ambuj Varshney, and Thiemo Voigt. Augmenting iot networks with backscatter-enabled passive sensor tags. In *Proceedings of the 3rd Workshop on Hot Topics in Wireless*, pages 23–27. ACM, 2016.

[39] Nathan M Pletcher, Simone Gambini, and Jan M Rabaey. A 2ghz 52 μw wake-up receiver with-72dbm sensitivity using uncertain-if architecture. In *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*, pages 524–633. IEEE, 2008.

[40] Qorvo. Tqp3m9008 high linearity lna gain block. https://www.qorvo.com/products/p/TQP3M9008, 2022.

[41] Aanjhan Ranganathan, Boris Danev, and Srdjan Capkun. Low-power distance bounding. *arXiv preprint arXiv:1404.4435*, 2014.

[42] Mohammad Rostami, Xingda Chen, Yuda Feng, Karthikeyan Sundaresan, and Deepak Ganesan. Mixiq: re-thinking ultra-low power receiver design for next-generation on-body applications. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 364–377, 2021.

[43] Mohammad Rostami, Karthik Sundaresan, Eugene Chai, Sampath Rangarajan, and Deepak Ganesan. Redefining passive in backscattering with commodity devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–13, 2020.

[44] Camilo Salazar, Andreia Cathelin, Andreas Kaiser, and Jan Rabaey. A 2.4 ghz interferer-resilient wake-up receiver using a dual-if multi-stage n-path architecture. *IEEE Journal of Solid-State Circuits*, 51(9):2091–2105, 2016.

[45] Aaron Parks & samannp. Wisp 5 - hw. https://github.com/wisp/wisp5-hw, 2017.

[46] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, 2008.

[47] Bill Schweber. Understanding the basics of low-noise and power amplifiers in wireless designs. https://www.digikey.com/en/articles/understanding-the-basics-of-low-noise-and-power-amplifiers-in-wireless-designs, 2013.

[48] CADENCE PCB SOLUTIONS. What is the time constant of an rlc circuit? https://resources.pcb.cadence.com/blog/2020-what-is-the-time-constant-of-an-rlc-circuit, 2020.

[49] STMicroelectronics. Stm32l4 series. https://www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html, 2021.

[50] Mark Stoopman, Shady Keyrouz, Hubregt J Visser, Kathleen Philips, and Wouter A Serdijn. Co-design of a cmos rectifier and small loop antenna for highly sensitive rf energy harvesters. *IEEE Journal of Solid-State Circuits*, 49(3):622–634, 2014.

[51] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R. Smith, and Shyamnath Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3), September 2017.

[52] Taobao. Eka950 0.1μa-level power monitor. https://https://m.tb.cn/h.U1A802y?tk=1u3A2wmBIRx, 2021.

[53] Stewart Thomas and Matthew S Reynolds. Qam backscatter for passive uhf rfid tags. In *2010 IEEE International Conference on RFID (IEEE RFID 2010)*, pages 210–214. IEEE, 2010.

[54] Ambuj Varshney, Oliver Harms, Carlos Pérez-Penichet, Christian Rohner, Frederik Hermans, and Thiemo Voigt. Lorea: A backscatter architecture that achieves a long communication range. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2017.

[55] Ambuj Varshney, Carlos Pérez Penichet, Christian Rohner, and Thiemo Voigt. Towards wide-area backscatter networks. In *Proceedings of the 4th ACM Workshop on Hot Topics in Wireless*, pages 49–53, 2017.

[56] Ambuj Varshney, Wenqing Yan, and Prabal Dutta. Judo: addressing the energy asymmetry of wireless embedded systems through tunnel diode based wireless transmitters. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 273–286, 2022.

[57] Jue Wang, Haitham Hassanieh, Dina Katabi, and Piotr Indyk. Efficient and reliable low-power backscatter networks. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 61–72, New York, NY, USA, 2012. Association for Computing Machinery.

[58] Wikipedia. Low noise amplifier. https://en.wikipedia.org/wiki/Low-noise_amplifier, 2022.

[59] Die Wu, Muhammad Jawad Hussain, Songfan Li, and Li Lu. R2: Over-the-air reprogramming on computational rfids. In *2016 IEEE International Conference on RFID (RFID)*, pages 1–8, 2016.

[60] Wenyu Yang, Die Wu, Muhammad Jawad Hussain, and Li Lu. Wireless firmware execution control in computational rfid systems. In *2015 IEEE International Conference on RFID (RFID)*, pages 129–136, 2015.

[61] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using 54 commodity wifi. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 259–271, New York, NY, USA, 2016. Association for Computing Machinery.

[62] Pengyu Zhang, Pan Hu, Vijay Pasikanti, and Deepak Ganesan. Ekhonet: High speed ultra low-power backscatter for next generation sensors. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 557–568. ACM, 2014.

[63] Renjie Zhao, Fengyuan Zhu, Yuda Feng, Siyuan Peng, Xiaohua Tian, Hui Yu, and Xinbing Wang. Ofdma-enabled wi-fi backscatter. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, pages 20:1–20:15, New York, NY, USA, 2019. ACM.

[64] Yi Zhao, Joshua R Smith, and Alanson Sample. Nfc-wisp: A sensing and computationally enhanced near-field rfid platform. In *2015 IEEE International Conference on RFID (RFID)*, pages 174–181. IEEE, 2015.

Channel-Aware 5G RAN Slicing with Customizable Schedulers

Yongzhou Chen[1], Ruihao Yao[1], Haitham Hassanieh[2], Radhika Mittal[1]
[1]UIUC, [2]EPFL

Abstract. This paper focuses on 5G RAN slicing, where the 5G radio resources must be divided across slices (or enterprises) so as to achieve high spectrum efficiency, fairness and isolation across slices, and the ability for each slice to customize how the radio resources are divided across its own users. Realizing these goals requires accounting for the channel quality for each user (that varies over time and frequency domain) at both levels – inter-slice scheduling (i.e. dividing resources across slices) and enterprise scheduling (i.e. dividing resources within a slice). However, a cyclic dependency between the inter-slice and enterprise schedulers makes it difficult to incorporate channel awareness at both levels. We observe that the cyclic dependency can be broken if both the inter-slice and enterprise schedulers are greedy. Armed with this insight, we design RadioSaber, the first RAN slicing mechanism to do channel-aware inter-slice and enterprise scheduling. We implement RadioSaber on an open-source RAN simulator, and our evaluation shows how RadioSaber can achieve 17%-72% better throughput than the state-of-the-art RAN slicing technique (that performs channel-agnostic inter-slice scheduling), while meeting the primary goals of fairness across slices and the ability to support a wide variety of customizable enterprise scheduling policies.

1 Introduction

Network slicing is one of the key new features introduced in the 5G standards [3, 9, 37]. It refers to dividing network resources among different services or groups of users to create virtual customizable networks. Such virtualization enables cellular networks to expand beyond the classical "individual mobile user" use-case to a more general "groups of users" use-case which can support new applications with different requirements. These groups of users (typically referred to as enterprises in 5G) enter into service level agreements (SLA) with the network operator, which provides two features: (1) It governs the type of service and the total amount of resources allocated to each slice. For example, it can provide an ultra-reliable low-latency communication for first responders, connected vehicles, or hospitals performing remote surgeries [20]. It can provide cheap and scalable IoT connectivity for farmers using sensors to monitor crops and cities deploying sensors for traffic or air quality monitoring [18]. It can also provide high throughput connectivity for companies with multiple users as well as educational and training institutions using VR/AR [36]. (2) It allows each slice (or enterprise) to customize their virtual networks and dynamically manage their resources [22, 41]. For example, a farming enterprise might want to give higher bandwidth to drones collecting aerial images of crops as opposed to soil moisture sensors, or a hospital might want to prioritize traffic for critical remote surgeries at different times of the day.

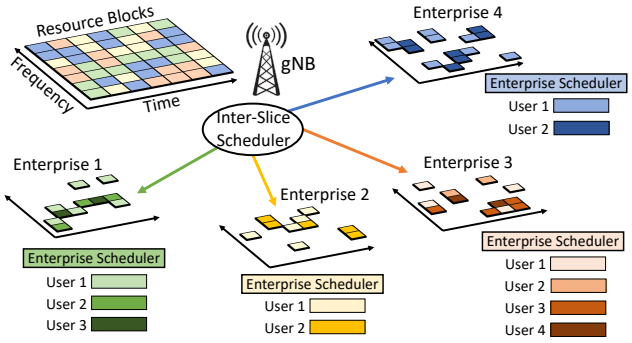


Figure 1: RAN Slicing of resources across 4 enterprises.

Network slicing has two components: 5G RAN (Radio Access Network) slicing and 5G core slicing [11, 29, 30, 44]. This paper focuses on RAN slicing as the RAN is typically the bottleneck in cellular networks [11, 27]. The goal is to divide physical layer resources at the base station (referred to as gNB in 5G) among different enterprises with devices connected to that gNB. These resources include time slots as well as frequency sub-bands used for transmission as shown in Fig. 1. Ideally, the RAN should schedule these resources in a manner that:

(1) Achieves high spectrum efficiency.
(2) Ensures fairness among enterprises subject to their SLAs.
(3) Allows enterprises to customize their scheduling policies.

Realizing the above goals, however, can be challenging in wireless networks because the throughput achieved by using a certain resource block is highly dependent on which user gets that block. In particular, the quality of the wireless channel can change drastically between frequency bands, between users, and over time. This well known phenomena is called frequency selective fading and is shown in Fig. 2 where the capacity can vary by up to 9× across 100 resource blocks (frequency sub-bands) for two users in a 20 MHz LTE bandwidth. Frequency selective fading is even more prominent in 5G where the total bandwidth is expanded to 100 MHz and up to 400 MHz [6]. Allocating resources in a channel agnostic manner can lead to inefficient spectrum usage and unfairness among different slices (enterprises) [10, 22].

Past work has considered the problem of channel aware spectrum allocation [2, 4, 38, 45], channel aware hierarchical resource scheduling in WiMax [21], and RAN virtualization in the context of MVNOs (Mobile Virtual Network

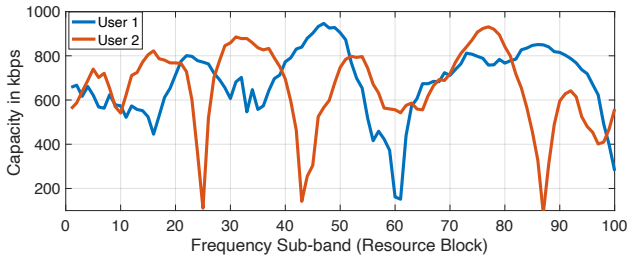


Figure 2: The channel quality across 100 RBs for two users in a 20MHz LTE downlink.

Operators)[1] [9, 22]. However, state-of-the-art techniques can only account for the channel quality between users within the same slice (as detailed in § 2). In other words, only after the inter-slice scheduler allocates resources to different slices irrespective of channel quality, each enterprise can run a channel aware scheduler. Hence, a slice can end up with resources that have bad channel quality for its users which significantly degrades the throughput as we demonstrate in §3.1.

Enabling channel aware scheduling at both the inter-slice scheduler and enterprise scheduler leads to a chicken and egg problem. The enterprise scheduler cannot allocate resources in a channel aware manner before it knows all the resources the inter-slice scheduler will give it, and the inter-slice scheduler cannot allocate resources among slices in a channel aware manner if it does not know to which user in the slice the enterprise scheduler will give a certain resource. One way to break this deadlock is to enumerate through all possible resource allocations and run the enterprise scheduler for each one. However, enumerating all possibilities leads to exponential complexity and is intractable. As a result, state-of-the-art work only uses a channel agnostic inter-slice scheduler [9, 21, 22].

In this paper, we present RadioSaber, a RAN slicing policy that enables channel aware resource allocation at both the inter-slice scheduler and the enterprise scheduler while allowing each enterprise to customize their scheduling policy. RadioSaber's design is based on a simple idea. Since both the inter-slice scheduler and the enterprise scheduler must run at the base station for real-time scheduling, the inter-slice scheduler can use the enterprise scheduling algorithms as subroutines in its algorithm. Both the inter-slice and enterprise scheduler can be channel aware if the inter-slice scheduler can call the enterprise scheduling algorithm with the following query: "If we give resource X to slice A, which user in slice A will get resource X?" If the enterprise scheduler is able to reply to this query independent of what other resources the inter-slice scheduler will allocate to its slice, the inter-slice scheduler can be channel aware. Specifically, the inter-slice scheduler can query the enterprise scheduler of each slice, find the user to which the resource X will be allocated and determine the slice in which X will deliver the best channel quality.

Being able to answer this query, however, limits the space of scheduling policies that an enterprise can run. In particular, the enterprise scheduler must decide how to allocate a resource X independent of the remaining resources that have not yet been allocated to it as we show in § 4.1. Hence, its algorithm must greedily allocate one resource at a time. Most common practical policies, however, like Max. throughput [4], proportional fairness [45], QoS-aware scheduling [2, 4, 38, 39] etc., tend to use greedy algorithms that satisfy the above requirement. It is worth noting here that when allocating a resource X, the algorithms can still account for resources that have already been allocated to the slice in the past (e.g. in order to provide some notion of fairness or demand-awareness), but not account for resources that are yet to be allocated in the future. Hence, RadioSaber is able to accommodate policies that are both channel aware and flexible to the needs of different slices.

Realizing RadioSaber in practice, however, requires addressing algorithmic questions like how do we incorporate the SLA between the operator and the enterprise into the inter-slice scheduler? In what order does the inter-slice scheduler query the resource blocks to decide which slice gets the block? How do we support customizable enterprise scheduling, while restricting the schedulers to be greedy? How does the enterprise scheduler balance between channel quality and other metrics such as flow priorities? We also have to address several system level questions like what is a good and simple interface that we can provide to the enterprises to set their scheduling policies? How do we incorporate RadioSaber into the current 5G standards? We address the above questions in detail in §4.2 and § 4.3.

We have implemented RadioSaber and evaluated it using trace driven simulations. We used an open-source 5G core network [1] and a popular RAN simulator [34] that is capable of simulating both the physical layer and higher layers at the RAN. We evaluate our system using traces of cellular signals [46], that were collected using software defined radios. We compare with:(1) NVS [9, 21, 22], a popular algorithm which enables RAN slicing with channel aware enterprise scheduler and (2) a global channel aware scheduler that schedules all users without slicing.

Our results reveal that RadioSaber is able to outperform NVS, achieving 17%-72% better throughput for backlogged flows, 2× to 4× lower FCT (flow completion time) for non-backlogged flows, and 24× lower packet delays for real-time flows with constant bitrates. Unlike the global channel aware (but slicing unaware) scheduler that fails to provide any isolation across slices, RadioSaber is able to achieve desired isolation and fairness. Finally, RadioSaber is able to accommodate enterprise schedulers with various policies and number of users. Hence, RadioSaber is able to achieve the three goals of spectrum efficiency, fairness, and customizability.

[1] Examples of MVNOs include Straight Talk, Virgin Mobile, & Xfinity mobile, which do not operate their own networks but rather run their traffic through Verizon, AT&T, & T-mobile networks.

The paper makes the following contributions:

- We present the first RAN slicing that is channel-aware both at the inter-slice scheduler and enterprise scheduler.
- We present a new framework for RAN virtualization that abstracts physical layer scheduling and provides an interface for enterprises to set their own schedulers.
- We implement our techniques and demonstrate significant improvement in efficiency and fairness.

2 Background & Related Work

In this section, we provide a brief background on the radio access network (RAN) in cellular networks as well as the related work for channel aware scheduling and RAN slicing.

1. Resource Blocks: In 5G RANs, the user or device is referred to as UE (User Equipment) and the base station is referred to as gNB (next generation Node B). The gNB uses OFDMA (Orthogonal Frequency Division Multiple Access) at its PHY and MAC layers in order to divide radio resources across UEs. In OFDMA, the frequency bandwidth is divided into sub-carrier frequencies that are orthogonal (i.e. do not interfere) and time is divided into equal slots called TTIs (Transmission Time Interval). A resource block (RB), which is the smallest resource unit that can be allocated to a UE, is formed of 12 frequency sub-carriers and 1 TTI slot. Hence, the RBs are organized into a 2D grid as shown in Fig. 1. In practice, however, network operators typically schedule resources in the granularity of resource block groups (RBGs) to minimize control overhead. Each RBG contains a fixed number of consecutive RBs ranging from 1 to 4 [8, 42]. In 4G, each TTI has a fixed length of 1ms and each sub-carrier has a fixed width of 15 kHz. Thus, the RB spans $12 \times 15 = 180$ kHz. 5G, on the other hand, supports 5 configurable TTI and sub-carrier intervals such that the TTI is $2^{-\mu}$ ms and the sub-carrier interval is $2^{\mu} \times 15$ kHz. μ is commonly referred to as the numerology and chosen from the values $0, 1, 2, 3, 4$ depending on the band of operation [7, 35]. For example, the 5G sub-6GHz band, supports sub-carrier width of 60 KHz with a TTI of 0.25 ms for $\mu = 2$ [33, 43]. In this case, the RB spans 720 kHz.

2. Data Rate: The data rate at which a UE can transmit in a given RB depends on the channel quality which is typically defined by the SNR (signal-to-noise-ratio). The SNR determines the capacity of the wireless link and varies across time, RBs, and users as shown in Fig. 2. The SNR can be computed at the UE for each OFDM sub-carrier. For a RB or RBG made of many sub-carriers, the "effective SNR" is typically computed[2] as described in [16, 25, 31]. The effective SNR is then mapped to a discrete value called channel quality indicator (CQI) and periodically reported to the gNB [8]. The CQI is then used to determine the modulation and coding scheme (MCS) for the UE which in turn determines the data rate of the UE. The higher the SNR and channel quality, the higher order MCS can be used to increase the data rate.

[2]The effective SNR is not the average across sub-carriers but rather a weighted exponential average that typically gives a value close to the minimum SNR across sub-carriers. This ensures that the chosen data transmission rate does not exceed the capacity of the wireless link at any sub-carrier which would otherwise result in a very high packet loss rate.

UE	User Equipment	RAN	Radio Access Network
gNB	5G Base station	TTI	Transmission Time Interval
RB	Resource Block	RBG	Resource Block Group
SNR	Signal-to-Noise Ratio	SLA	Service Level Agreement
PF	Proportional Fair	CQI	Channel Quality Indicator
MT	Max. Throughput	MCS	Modulation & Coding Scheme

Table 1: Terms used in 5G networks

3. Channel Aware Scheduling: The need for channel-aware scheduling in wireless networks has led to a number of techniques that propose allocating resources across individual users in a channel-aware manner (e.g. [2, 4, 13, 14, 38, 39, 45]). In most cases, the scheduling problem is NP-Hard and a greedy heuristic is adopted whereby RBs are allocated one at a time to the UE that scores highest on some given metric [4]. These strategies ensure low scheduling overhead and enabling fast decisions at timescales of a single TTI. Some of the most common techniques include:

• *Maximum Throughput (MT):* Assigns the RB to the UE with the largest CQI to maximize data rate irrespective of fairness.
• *Proportional Fair (PF):* Extends MT to incorporate fairness across users by weighing the CQI with the UE's historical allocation. This is a very popular strategy in cellular networks that aims to optimize the sum of the log of throughput of UEs [17, 24, 45]. The PF metric can also be parameterized to vary the relative weights of the CQI versus historical allocation [45].
• *Incorporating QoS and delay:* The PF metric can be further extended to incorporate (i) QoS values that increase the weights of higher priority UEs [4, 13, 14, 39], or (ii) packet delays that increase weights of UEs that have been waiting longer [2, 38, 39].

Non-greedy heuristics have also been proposed for optimizing proportional fairness [17, 24]. The scheduling problem is abstracted as an NP-hard integer linear program and solved using a sub-optimal non-greedy algorithm. Such algorithms, however, are computationally very expensive and must use GPUs to compute their solutions [17].

4. RAN Slicing: The inter-slice scheduler provides each enterprise with a slice of the RAN by allocating a set of virtual RBs which it maps to physical RBs. In order to support channel aware scheduling, it also provides the enterprise scheduler with the CQIs of the UEs of this enterprise for each virtual RB. Each enterprise is then able to customize how it allocates the virtual RBs to its UEs by using virtual control functions at the gNB to specify its own scheduling policy [9, 10]. Note that both the inter-slice scheduler and enterprise scheduler run on the gNB which enables RadioSaber to expose the enterprise scheduling algorithms to the inter-slice scheduler as described in more details in §4.1.

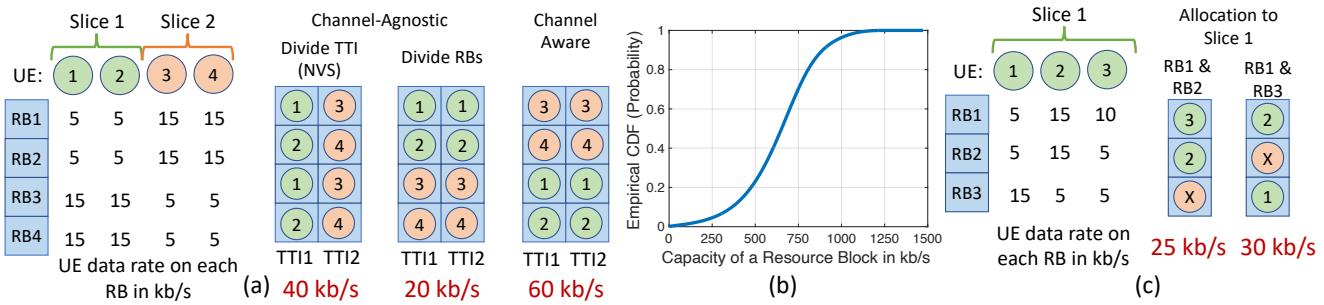


Figure 3: (a) Illustrative example showing the importance of a channel aware inter-slice scheduler. (b) Distribution of the capacity of RBs from real traces. (c) Example showing the challenge in enabling a channel aware inter-slice scheduler.

The closest to our work is NVS (Network Virtualization Substrate) [21], a popular inter-slice scheduler that is used by multiple RAN slicing schemes [9, 21–23, 26]. NVS allocates all RBs in a given TTI to a single slice independent of channel quality. It then rotates between slices in a weighted round-robin manner to satisfy the target throughput of each slice as specified by SLA. While NVS does allow the enterprise scheduler to run a channel aware policy, the inter-slice scheduler remains agnostic to channel quality which changes over time and resource blocks. We compare with NVS in §6 and show that our channel aware inter-slice scheduler can significantly improve performance.

Past work on slicing radio resources also explores supporting dynamic demands across slices [40, 47] and deadlines across slices [12, 15]. All past work, however, constraints the inter-slice scheduler to allocate virtual RBs to each slice in a channel-unaware manner. Our goal, in contrast, is to enable channel-aware scheduling at both inter-slice and enterprise levels, while still giving each enterprise enough flexibility to allocate RBs across its users.

3 Motivation and Challenges

In this section, we will explain, using illustrative examples, the importance of channel aware resource allocation at both the inter-slice scheduler and enterprise scheduler as well as the challenge in making the inter-slice scheduler channel aware.

3.1 Need for Channel-Aware Slicing

To best illustrate the need for channel-aware slicing at both the inter-slice scheduler and enterprise scheduler, consider the toy example in Fig. 3(a). In this example, there are 4 RBs $\{R_1, R_2, R_3, R_4\}$, 2 slices $\{S_1, S_2\}$, and each slice has 2 UEs: $\{u_1, u_2\} \in S_1$ and $\{u_3, u_4\} \in S_2$. The channel quality is shown in the left 2D grid in terms of the data rate each UE can achieve on each RB in kb/s. The enterprise scheduler of both slices are channel-aware and run a proportional fairness (PF) policy in order to maximize throughput while ensuring fairness between UEs.

Consider a channel unaware inter-slice scheduler. The scheduler could allocate all RBs in the first TTI to S_1 and all RBs in the second TTI to S_2 similar to NVS [21]. In this case, the enterprise scheduler of S_1 would allocate $\{R_1, R_3\}$ to u_1 and $\{R_2, R_4\}$ to u_2. Similarly, the enterprise scheduler of S_2 would allocate $\{R_1, R_3\}$ to u_3 and $\{R_2, R_4\}$ to u_4. This allows each UE to achieve 10 kb/s over the two TTIs for a total of 40 kb/s. An alternative channel unaware inter-slice scheduler could have also divided the RBs between the two slices by allocating $\{R_1, R_2\}$ to S_1 and $\{R_3, R_4\}$ to S_2 for all TTIs. In this case, the enterprise scheduler of S_1 would allocate $\{R_1\}$ to u_1 and $\{R_2\}$ to u_2 and that of S_2 would allocate $\{R_3\}$ to u_3 and $\{R_4\}$ to u_4. This allows each UE to achieve a data rate of 5 kb/s for a total of 20 kb/s. Since the inter-slice scheduler is channel-unaware, it has no way of figuring out that such an allocation is very inefficient.

Consider, on the other hand, a channel-aware inter-slice scheduler. This scheduler would allocate $\{R_3, R_4\}$ to S_1 and $\{R_1, R_2\}$ to S_2 for all TTIs. In this case, the enterprise scheduler of S_1 would allocate $\{R_3\}$ to u_1 and $\{R_4\}$ to u_2 and that of S_2 would allocate $\{R_1\}$ to u_3 and $\{R_2\}$ to u_4. This allows each UE to achieve a data rate of 15 kb/s for a total of 60 kb/s. Hence, the channel-aware inter-slice scheduler enables 1.5× to 3× higher throughput.

There are two factors in this example that enable a channel-aware scheduler to achieve better performance than a channel-agnostic scheduler: (i) The channel quality differs across RBs for a given slice, and (ii) The two slices have complementary channel quality distribution across RBs (i.e. slice 1 has a better channel quality for the first two RBs, and slice 2 has better channel quality for the last two RBs). It is the combination of these factors that enables a smarter (channel-aware) resource allocation policy to achieve better performance. Moreover, such factors are quite common in practice as can be seen from the real traces shown in Fig. 2.

While this toy example illustrates the insight behind RadioSaber, the variation in channel quality in real systems can be quite significant as was shown in Fig. 2. Fig. 3(b) plots the cumulative distribution function (CDF) of capacity of all the RBs in a real trace collected from 4G measurements (obtained from [46]). The figure shows that the channel in real traces can vary significantly leading to a capacity that can be as high as 2.2× the median value and as low as 1/20 of the median value. This diversity which is a result of frequency

selective fading in the wireless channel is expected to further increase in 5G as the bandwidth increases from 10-20 MHz to 100-400 MHz [6].

Our evaluation in §6 shows that due to this diversity, our simple insight from the toy example generalizes to more complex scenarios.

3.2 Challenge in Channel-Aware Slicing

In order to allocate RBs across slices in a channel-aware manner, we first need to know what channel quality each slice would achieve for each RB. While in the above toy example it is easy to see what the best channel aware allocation is, the problem is challenging in more general settings. In particular, if the inter-slice scheduler gives RB R_i to slice S_j, then the channel quality of R_i will depend on which UE belonging to S_j will use R_i which in turn depends on the enterprise scheduling policy. However, the enterprise scheduling policy itself could be channel-aware, in which case, the allocation of R_i to a given UE will depend on which RBs the inter-slice scheduler has allocated to S_j. This creates a deadlock as the inter-slice scheduler needs to know how the enterprise scheduler will allocate R_i to determine its channel quality and whether to give R_i to this slice. At the same time, the enterprise scheduler needs to know what RBs the inter-slice scheduler will give it so it can allocate them in a channel-aware manner.

Fig. 3(c) illustrates this through a toy example. Consider a slice with three UEs $\{u_1, u_2, u_3\}$. Suppose the inter-slice scheduler must allocate two RBs to this slice (as per its weighted share) and suppose the enterprise scheduling policy assigns a RB to the user which has maximum data rate for the RB, while limiting each user's allocation to a single RB. If the inter-slice scheduler allocates $\{R_1, R_2\}$ to the slice, then the enterprise scheduler would first allocate R_2 to u_2 and then allocate R_1 to u_3. However, if the inter-slice scheduler allocates $\{R_1, R_3\}$ to the slice, the enterprise scheduler would allocate R_1 to u_2 and R_3 to u_1. The data rate associated with R_1 for this slice is 10 kbps in the first case and 15 kbps in the second case. Hence, while determining whether to allocate R_1 to this slice or not, the inter-slice scheduler does not know the data rate that R_1 will deliver as it depend on whether the other RB allocated to the slice is R_2 or R_3.

One way out is to enumerate through all possible combinations of inter-slice allocations, and run the enterprise scheduler for each slice for each allocation. However, this is clearly intractable with the number of possible allocations increasing exponentially with the number of slices and resource blocks.

3.3 RadioSaber's Approach

In order to break the deadlock challenge outlined in §3.2, we leverage the following insights. First, both the inter-slice and enterprise scheduler must run on the base station (gNB) to guarantee real-time scheduling. Hence, the inter-slice scheduling algorithm can use the enterprise scheduling algorithm as a subroutine and query it to figure out how it will allocate a certain RB. Second, we can break the deadlock if the enterprise scheduler is able to reply to the following query from the inter-slice scheduler: "If I give resource R_i to slice S_j, which UE in slice S_j will get resource R_i?".

For the enterprise scheduler to be able to reply to this query, its algorithm should determine how to allocate a RB independent of other RBs that the inter-slice scheduler might allocate to it. Restricting the enterprise scheduling algorithm to greedily allocate one RB at a time makes it independent of the remaining RBs that will be allocated to it. It may still need to account for the historical allocation of RBs (i.e. RBs already assigned to the slice) to correctly estimate the remaining demand of a user or to account for fairness. A greedy inter-slice scheduler, that assigns RBs to slices one at a time, enables the enterprise scheduler to update its scheduling state based on its allocation so far. Restricting both scheduler to be greedy thus enables the enterprise scheduler to effectively answer the query while still accounting for historical allocation, and for the inter-slice scheduler to use the result of the query to assign the RB to the slice with the best channel quality.

4 RadioSaber's Design

Objectives. RadioSaber tackles the problem of dividing N RBs (over one or more TTIs) across K slices, and then dividing the RBs allocated to each slice across the UEs within that slice, such that the following objectives are met:

(i) Weighted fairness across slices. Each slice must get its weighted fair share of resource blocks. We assume that the weights are known and are proportional to each slice's demand (based on the SLA between the slice owner and the cellular network operator). Prior work on RAN slicing [22] shows how SLAs can be specified either in terms of number of RBs or overall throughput, and how both can be translated to dynamic per-slice weights. We use these weights to compute a quota of RBs for each slice, as detailed in §4.1.

(ii) High spectrum efficiency. The system must allocate RBs across slices so as to use the spectrum efficiently and achieve high overall throughput. For this, RadioSaber uses the approach outlined in §3.3 to greedily determine which RBs are allocated to a slice to fulfill its quota in a channel-aware manner. We detail RadioSaber's greedy channel-aware inter-slice scheduling algorithm in §4.1.

(iii) Customizable enterprise scheduler. Each slice can have a different policy for dividing the RBs allocated to it across its UEs and flows. The system should provide an expressive interface for slice operators to specify their desired policies, and should be able to enforce them. §4.2 describes RadioSaber's framework for supporting a variety of greedy enterprise schedulers.

After describing individual components of RadioSaber's design, we end this section with describing our overall workflow for RAN slicing in §4.3.

4.1 Inter-slice Scheduler

We divide inter-slice scheduling logic into two steps: (i) computing the quota of RBs for each slice in a TTI, and (ii) greedily allocating RBs to slices as per their quota in a channel-aware manner. [3] We detail these steps below.

Computing Per-slice Quotas. In each TTI, RadioSaber first computes the quota of RBGs (the granularity at which RBs are allocated) for each slice, based on per-slice weights. Let the number of RBs and the number of RBGs in each TTI be $|\mathcal{RB}|$ and $|\mathcal{RBG}|$ respectively. A naive strategy is to simply compute the quota of slice s as $w_s|\mathcal{RBG}|$, where w_s is the normalized weight of the slice. However,there are a few practical considerations: (i) The quota for a slice computed in this manner could be non-integral (and possibly less than 1, depending on the number of other slices and their weights). We cannot have non-integral allocation of RBGs. (ii) If $|\mathcal{RB}|$ is not a perfect multiple of the default number of RBs in each group (say k), then the last RBG will have fewer RBs.

RadioSaber accounts for these aspects by maintaining an offset from the (ideal) target share of RBs for each slice, that rolls over to the next TTI. Algorithm1 presents the pseudo-code. We first compute the target share of each slice (in number of RBs) as its absolute weighted share in a TTI (given by $w_s|\mathcal{RB}|$) subtracted by its rolling offset from the previous TTIs (initialized to zero for a new slice). We then set the quota for the slice (in number of RBGs) as its target share divided by k, and round down the result. Because of the rounding down, the sum of quota across all slices would be less than the available number of RBGs. We then increment the quota of a random set of slices by one, so as to account for all of the extra RBGs. This can result in a few slices getting less than their fair share of RBs, and a few slices getting more. We capture this by updating the offset for each slice. This offset is then taken into account when computing the quotas in the next TTI – the slices that get more than their fair share of RBs in the current TTI will have a positive offset and a lower share in the next TTI, while the slices that get less than their fair share of RBs in the current TTI will have a negative offset and a higher share in the next TTI.

As mentioned above, one of the RBGs allocated to a slice may have fewer than k RBs. We account for this by adjusting the offset of the slice that is allocated the aberrant RBG when assigning RBGs to slices (we skip mentioning this step when discussing our assignment algorithm below).

Assigning RBGs to Slices. Given per-slice quotas, we next need to assign RBGs to slices in a channel-aware manner, so as to maximize spectrum efficiency. Even if we assume that the channel-quality (or the data-rate) associated with each slice for each RBG is known in advance (which, as illustrated in §3, is not the case), computing the optimal assignment of RBGs that maximizes the total data-rate, while adhering to the quota on RBGs for each slice is an NP-hard problem. [4] A greedy heuristic is therefore a natural choice for finding (a potentially sub-optimal) solution to this problem. But more importantly, as discussed in §3.3, a greedy approach allows the inter-slice scheduler to effectively query the enterprise scheduler and determine the channel quality for each RBG.

The basic allocation logic then becomes relative straightforward: In each TTI, RadioSaber greedily picks a RBG and assigns it to the slice that achieves the maximum channel quality for that RBG. Once a slice has been allocated its quota of RBGs, it is no longer considered for subsequent RBGs allocation in that TTI.

The order in which the inter-slice scheduler allocates

Algorithm 1 Calculating RBGs quota for slices

```
1:  variable rbs_offset_
2:  procedure SLICEQUOTA
3:      rbs_share = []
4:      rbgs_quota = []
5:      k ← rbs_per_rbg()
6:      for s in S do
7:          rbs_share[s] ← |RB| × w_s - rbs_offset_[s]
8:          rbgs_quota[s] ← ⌊ rbs_share[s] / k ⌋
9:      end for
10:     extra_rbgs = |RBG| - sum(rbgs_quota)
11:     while extra_rbgs > 0 do
12:         rbgs_quota[S.rand()] += 1
13:         extra_rbgs -= 1
14:     end while
15:     for s in S do
16:         rbs_offset_[s] = rbgs_quota[s] × k - rbs_share[s]
17:     end for
18:     return rbgs_quota
19: end procedure
```

Figure 4: Allocating three RBs to three slices with same weights using different strategies.

[3] RadioSaber follows the standard practice of making radio resource allocation decisions at timescales of a TTI (§2). Consequently, any temporal variations in a UEs CQI is naturally accounted for when recomputing the schedule over each subsequent TTIs.

[4] It reduces to an Integer Linear Programming problem [45].

Algorithm 2 Assigning RBGs to slices

```
1:  procedure RBALLOCATION
2:      rbgs_quota ← sliceQuota()
3:      rbgs_assignment = []
4:      slice_user = []        ▷ Users that each slice decides to schedule for all RBs
5:      slice_cqi = [] ▷ Channel qualities of each slice(based on scheduled users) for all RBs
6:      slice_rbgs = []
7:      for i in range(|RBG|) do
8:          for s in S do
9:              u* ← s.enterpriseScheduler(i)
10:             slice_user[i][s] ← u*
11:             slice_cqi[i][s] ← u*.channelQuality(i)
12:         end for
13:     end for
14:     while rbs_assignment.size() < |RBG| do
15:         i*, s* ← argmax_{i,s} slice_cqi and i not in rbgs_assignment and slice_rbgs[s] < rbgs_quota[s]
16:         u* ← slice_user[s*]
17:         u*.allocRB(i*)
18:         rbgs_assignment[i*] = u*
19:         slice_rbgs[s*] += 1
20:         if slice_rbgs[s*] >= rbgs_quota[s*] then
21:             Continue
22:         end if
23:         for i in range(|RBG|) do
24:             u' ← s*.enterpriseScheduler(i)
25:             slice_user[i][s*] ← u'
26:             slice_cqi[i][s*] ← u'.channelQuality(i)
27:         end for
28:     end while
29: end procedure
```

the RBGs can impact spectrum efficiency. Fig. 4 illustrates this with an example. It shows the channel quality (datarate) associated with three slices $\{s_1, s_2, s_3\}$ for three RBGs $\{R_1, R_2, R_3\}$. Each slice has a quota of 1 RBG. Assigning RBGs sequentially in the order $\{R_1, R_2, R_3\}$ to the slice that has maximum datarate (until it exhausts its quota) results in a total datarate of 50Kb/s. In comparison, allocating RBGs in decreasing order of channel quality achieves a higher datarate of 60Kb/s as it greedily selects the best RBG-slice mapping (in terms of datarate) in each iteration. RadioSaber adopts this strategy for inter-slice scheduling.

While this greedy strategy results in a better outcome than the sequential allocation, it does not produce the optimal result. In fact, the optimal allocation for the example in Fig. 4 would have resulted in a higher datarate of 65kb/s. This suboptimality is expected from any polynomial time algorithm, given the NP-hardness of our resource allocation problem.

Algorithm2 presents the pseudo-code for RadioSaber's inter-slice scheduler. It computes maximum channel quality across slices for each unallocated RBG (lines 7-13). It allocates the RBG with the highest channel quality (say i^*) to the corresponding slice (s^*) that has the highest channel quality for it (line 15). If the quota for slice s^* is not exhausted, it recomputes the channel quality for the remaining RBGs for that s^* (lines 20-24). This recomputation is needed because the previous allocation may influence how the enterprise scheduler in s^* schedules UEs for subsequent RBGs allocated to it (e.g. if a UE has met its demand or to ensure fairness across UEs). The above steps are then repeated to proceed with the allocation of the next RBG.

This algorithm has a polynomial time complexity of $O(|\mathcal{RBG}|^2(|\mathcal{S}|+T)+|\mathcal{RBG}||\mathcal{S}|T)$ in the worst case, where T is the time complexity of the enterprise scheduler for assigning an RBG to a UE (this is typically $O(|\mathcal{U}|)$ for $|\mathcal{U}|$ UEs in a slice for the greedy enterprise scheduler described in §4.2).

4.2 Customizable Enterprise Scheduling

We now describe RadioSaber's framework for supporting different enterprise scheduling policies. Restricted by the need for the enterprise scheduler to be greedy (§3), and inspired by existing channel-aware wireless schedulers (described in §2), we adopt a metric-based allocation strategy. The enterprise scheduler computes a metric for each UE for the RBG it is assigned (or queried about by the inter-slice scheduler) and selects the UE that scores highest on that metric. These metrics are parameterized – RadioSaber exposes these parameters to the slice operators, allowing them to tune the parameters in order to express different scheduling policies. [5]

At a high-level RadioSaber allows the slice operator to specify a variety of policies based on flow priorities, fairness across UEs, channel-quality, and packet delays. It supports two scheduling paradigms:

Paradigm 1: Select User First. This paradigm assigns a given RBG i to a UE that scores the highest on the following metric (corresponding to the generalized PF [45]), and schedules the highest priority flow belonging to that UE.

$$metric(u,i) = d_{u,i}^{\varepsilon}/R_u^{\psi}$$

Here, $d_{u,i}$ is the instantaneous data rate of UE u for RBG i, and R_u captures the historical RBG allocation for the UE u as an exponential weighted moving average of the user's throughput, based on its datarate for the RBGs it has been assigned so far. The parameters, ε and ψ, are integers that determine the relative weightage of channel quality and historical allocation. For example, setting $\varepsilon = \psi = 1$ instantiates the default

[5] RadioSaber makes an implicit assumption that each slice operator wishes to implement a customized scheduler, and accordingly specifies the scheduling parameters that allow RadioSaber's enterprise scheduling framework to answer the inter-slice scheduler's query and to allocate resources. For slices that do not wish to implement a customized policy, RadioSaber can initialize the scheduling parameters to reflect the default scheduling policy decided by the network operator.

PF(proportional fairness) scheduler, or setting $\varepsilon = 1$ and $\psi = 0$ instantiates the MT(maximum throughput) scheduler.

Paradigm 2: Select Highest Priority First. This paradigm always assigns a given RBG i to the flow with highest priority. When the highest priority level (p) has multiple flows (across multiple UEs), it selects a flow (or a UE) [6] based on the following metric (from [2,4]):

$$metric(u,p,i) = (\beta D_{u,p} + (1-\beta))(d_{u,i}^{\varepsilon}/R_u^{\psi})$$

Here, $D_{u,p}$ is the queuing delay experienced by the packet at the head of the queue corresponding to UE u and priority p. β is a binary parameter that determines whether the head packet delay ($D_{u,p}$) influences the choice of UE. $d_{u,i}$, R_u, ε and ψ are as described above. Setting $\beta = \varepsilon = \psi = 1$ instantiates the M-LWDF(Modified-Largest Weighted Delay First) Scheduler [2].

A binary parameter α allows a slice operator to pick between the two paradigms ($\alpha = 0$ picks the first paradigm and $\alpha = 1$ picks the second paradigm).

In summary, there are four parameters that RadioSaber exposes for tuning the enterprise schedulers:
(1) α that determines whether a UE is selected first or the priority level.
(2) ε that determines the weightage of channel quality.
(3) ψ that determines the weightage of historical allocation.
(4) β that determines whether the head packet delay is a factor.
A slice operator can tune these parameters differently to express different policies, as per their requirements and workload. These parameters are initialized when the slice operator creates the slice, and are stored in the data repository in the 5G core (as detailed in §4.3). [7]

While our enterprise scheduling approach is heavily inspired from existing wireless scheduling techniques, we introduce some new aspects. In particular, cellular schedulers typically do not consider different priority levels for a given UE. Instead, they map all flows belonging to a UE to a single queue (the default radio bearer). We introduce the notion of different priority levels for each UE, as we believe that is an important requirement for emerging applications where each UE may have different types of flows.

4.3 RAN Slicing Workflow

We now explain the RAN slicing workflow for RadioSaber.

[6] We assume each UE has a single flow at each priority level. Even if a UE has multiple flows for a given priority level, they are served in a FIFO order with respect to one another, and we can logically treat them as a single flow.

[7] We assume that the slice operators are fine with sharing their coarse-grained scheduling policies with the network operators (in the form of these parameters). Extending our system to provide a secure environment for the slice operators to schedule their UEs and to answer the inter-slice scheduler's query in manner that does not reveal the scheduling policies is an interesting future direction, that is beyond the scope of this paper.

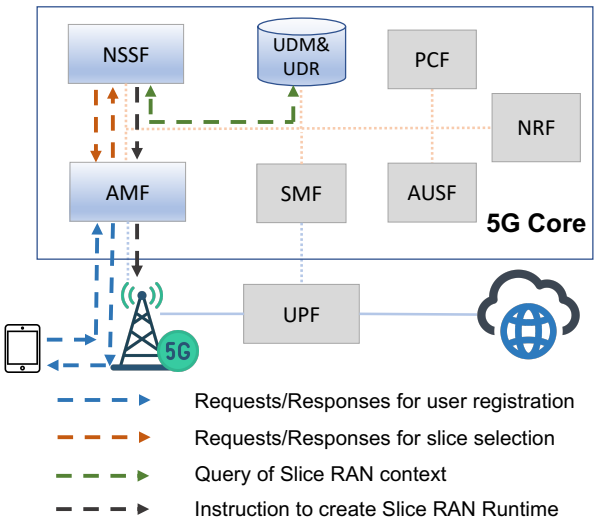


Figure 5: 5G Core network architecture, and the workflow for relaying slice context from 5G core to gNB.

4.3.1 Relaying Slice Context to gNB

The relevant slice context for RadioSaber includes the slice ID, a list of its users, the slice SLA (or weight), and the enterprise scheduling parameters. When the slice operator registers a new slice, this context is initialized and stored in the 5G core. When a user belonging to this slice attaches to a gNB, the relevant context must be relayed from the 5G core to the gNB for RAN slicing. We now describe the workflow for it.

Fig. 5 shows different modules of 5G core. The modules relevant for RadioSaber's workflow are shaded in blue. UDM & UDR (Universal Data Management & Repository) manages and stores all user-related data including slice contexts. AMF (Access and Mobility Function) manages different aspects of a UE (other than data forwarding) – these include connection, reachability, mobility, authentication, authorization, and location services. NSSF (Network Slicing Selector Function) handles slice selection request and manages the life cycle of a network slice instance.

Standard implementations for 5G core provide a framework to support core slicing. We extend it for RAN slicing with RadioSaber by adding the following workflow:
(1) When a UE tries to connect, it issues a registration request to the gNB, which then forwards the request to the AMF.
(2) The AMF in turn issues a request to the NSSF to fetch the slice context associated with the UE.
(3) If the corresponding slice instance is not available at the NSSF, it means that the UE is the first to register for the slice. The NSSF then constructs the slice instance based on the slice context stored in the UDM, and passes the context to the AMF in response. If an instance for the slice is already available at the NSSF, it directly responds to AMF.
(4) The AMF then relays the slice context obtained from NSSF to gNB through the NGAP(NG Application Protocol).
(5) The gNB uses the information received from the AMF

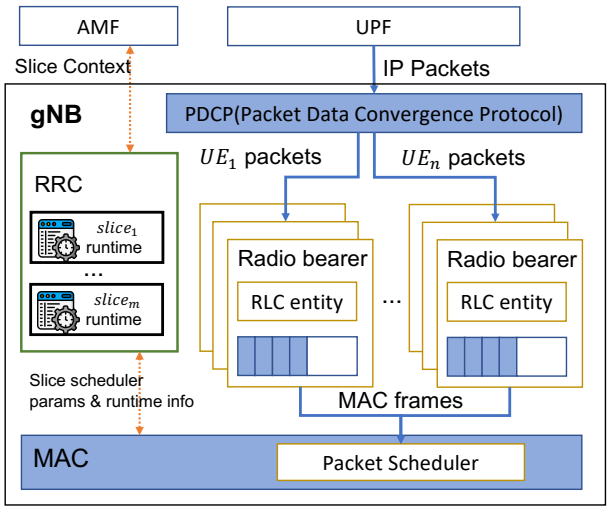


Figure 6: 5G gNB architecture: it shows how RRC maintains slice runtimes, and controls the MAC scheduler behavior.

to create a RAN slice runtime instance if that UE is the first to register for that slice at the gNB. Otherwise, it uses the information to determine the slice ID for the UE, and map the UE to the corresponding RAN slice instance.

4.3.2 Scheduling at gNB

The gNB stores the slice context obtained from the 5G core at the RRC (Radio Resource Control). The IP packets arrive at the gNB from the UPF (User Plane Function), and are intercepted by the PDCP (Packet Data Convergence Protocol). PDCP is responsible for compressing IP headers and ciphering data for integrity protection. For RadioSaber, it parses the packet priority from the DSCP field in the IP header.

Typically, a gNB maintains a single queue for each UE associated with a single QoS level (radio bearer) [32]. To support multiple priority levels within a UE, we simply allow multiple discrete priority queues (radio bearers) for each UE. [8]

Upon parsing the priority level from the packet header, the PDCP can then forward the packets to the radio bearer with the corresponding UE and priority level.

The packet scheduler in the gNB uses the slice context stored in the RRC (that includes slice weights and enterprise scheduling parameters) and the per-UE context (including its periodically updated CQI and historical allocations) to run both the inter-slice and the enterprise schedulers.

5 Implementation

5G Core Support. We extend Open5GS (an open-source 5G core) [1] to add support for the RadioSaber workflow described in §4.3.1. Open5GS already provides a framework for core slicing. We add the serialization and deserialization procedure of our RAN slice context, and the communication between the AMF and gNB for the construction and destruction of RAN slice runtime. This implementation comprises 530 lines of code in total.

RAN Slicing and Scheduling Logic. Due to high overhead and massive cost of deploying a base station and scaling to large number of users, we evaluate our system using trace-driven simulations. We implement RadioSaber's RAN slicing and scheduling logic in an open-source RAN simulator [34]. By default, the simulator was configured to use LTE settings. We extend it to support 5G configuration. In particular, we configure the downlink bandwidth to 100MHz and the TTI to 250μs. We set the guard band to 3920KHz so there are 128 RBs and 32 RBGs in total. We also add support for some practical considerations (e.g. CQI reporting intervals). We ensure that UEs report subband CQI every 40ms, at a granularity that covers 4 RBs. We extend the simulator to support multiple priority levels for each UE. Moreover, we implement multiple scheduling policies in the simulator, including RadioSaber's inter-slice scheduler, the inter-slice scheduling logic in NVS [22], and the enterprise scheduling policies.

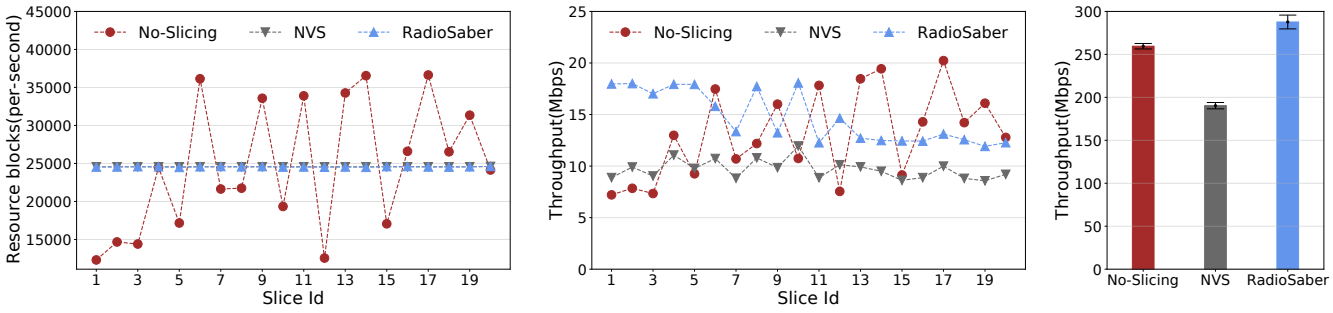
As detailed in §6, we use this simulator to do trace driven simulations to evaluate RadioSaber, using traces from LTScope [46]. These traces measure LTE signal strength (SNR over time and sub-carriers) for major US mobile carriers. We extend those measurements to 5G by concatenating five randomly selected distinct measurements of the 20 MHz LTE bandwidth to generate measurements over a 100 MHz 5G bandwidth.

6 Evaluation

We use the simulator described in §5 to evaluate RadioSaber. Our evaluation results in the following key takeaways:

- RadioSaber achieves higher overall throughput than the state-of-the-art (channel-unaware) inter-slice scheduler, NVS [9, 21, 22] (§6.1).
- RadioSaber can correctly enforce isolation and weighted fairness across slices (§6.1).
- RadioSaber is able to support diverse and customizable enterprise scheduling policies. The throughput wins with RadioSaber over NVS translate to better performance on the corresponding slice-specific metrics (§6.2).
- Complementary CQI distribution across slices is required for channel awareness to produce throughput gains (§6.3).
- RadioSaber's performance benefits hold as we vary the number of slices and UEs/slice (§6.4).
- RadioSaber performs better than NVS with non-greedy PF enterprise schedulers(§6.5).
- RadioSaber performance is very close to our (contrived) upperbound (§6.6).
- The scheduling latency of RadioSaber is within a TTI and the runtime overhead is low (§6.7).

[8] We expect a typical deployment to support 4-8 priority levels.



(a) per-slice RBs per second (b) per-slice throughput (c) sum total throughput

Figure 7: Comparing RadioSaber with NVS and no slicing. Number of slices is fixed to 20, with 5-15 UEs in each slice. The first 10 slices use MT and the next 10 use PF enterprise schedulers (except for no slicing, which uses a global PF scheduler).

6.1 Spectrum Efficiency and Fairness

We first evaluate how RadioSaber can achieve both high spectrum efficiency and fairness between slices. We configure our experiment to use 20 slices, and randomly assign between 5 to 15 UEs to each slice. While we expect there to be hundreds of slices, and hundreds of UEs associated with each slice, only a few of them will be active (and in the geographical vicinity) of a given gNB.

We configure all slices to have the same weight. To evaluate total spectrum efficiency of RadioSaber, we instantiate each UE with a single backlogged flow. The enterprise scheduler-ing policy is configured to follow MT scheduling for the first 10 slices (with $\beta = 0$, $\varepsilon = 1$, and $\psi = 0$), and PF scheduling for the last 10 slices (with $\beta = 0$, $\varepsilon = 1$, and $\psi = 1$).

We compare RadioSaber with two baselines in this experiment: (i) NVS, with a channel-agnostic inter-slice scheduler described in §2, and the same (channel-aware) enterprise schedulering policy for each slice as described above; (ii) No-Slicing, which uses a global PF scheduler to schedule UEs without any notion of slicing.

Fig. 7(a) shows the number of resource blocks allocated to each slice over intervals of 1 second. Both RadioSaber and NVS allocate same shares of radio resources to slices since each slice is configured to have the same weight. In contrast, No-Slicing cannot guarantee the same level of fairness across slices. With No-Slicing, the allocated RBs for a slice is roughly proportional to the number of UEs in the slice.

Fig. 7(b) shows the aggregate average throughput achieved by each slice. We see that, even though NVS and RadioSaber allocate the same number of RBs to each slice, RadioSaber achieves better throughput than NVS for each slice by enabling a better (channel-aware) assignment of RBs to slices. As expected, for both RadioSaber and NVS, the first 10 slices (that use MT schedulers) achieve higher average throughput than the last 10.

Fig. 7(c) shows the overall throughput computed as the total number of bytes transmitted across all UEs and over all TTIs and divided by the total number of TTIs in the experiment run. RadioSaber achieves 51% higher throughput than NVS and 11% higher throughput than No-Slicing. The higher throughput compared to NVS comes from channel-aware inter-slice scheduling with RadioSaber. The higher throughput compared with No-Slicing stems from the first 10 slices applying MT enterprise schedulers as compared to all UEs being scheduled as per the PF policy with No-Slicing (that does not have any notion of customizable per-slice scheduling).

We also evaluated a setting where slices differ in their weights. Both NVS and RadioSaber adhered to the specified per-slice weights when allocating RBs, while No-Slicing could not, and RadioSaber achieved better throughput than NVS. We present these results in Appendix§A.1.

If No-Slicing used MT instead of PF, it would have necessarily achieved a better overall throughput than RadioSaber, but would have failed to provide weighted fairness across users. In contrast, RadioSaber strives to maximize throughput while meeting this basic objective of isolation and fairness across RAN slices.

Slices	Scheduler $(\alpha, \beta, \varepsilon, \psi)$	Traffic generation	Metrics
1-5	PF(0,0,1,1)	a backlogged flow	average throughput
6-10	PF(1,0,1,1)	heavy-tail distributed flows	FCT(Flow Completion Time)
11-15	PF(1,0,1,1)	heavy-tail distributed flows(25% prioritized)	FCT of prioritized flows
16-20	M-LWDF (1,1,1,1)	a 1280kbps real-time video flow	average queueing delay

Table 2: Scheduling configuration, workloads per user, and metrics to evaluate in different slices.

6.2 Diverse Enterprise Schedulers

We now evaluate how RadioSaber supports diverse and customizable enterprise schedulers, and maintains isolation between slices. We compare against the same baselines: NVS and No-Slicing. We consider a total of 20 slices, and randomly assign between 5-15 users to each slice. We group the slices into 4 categories, with 5 slices in each category. Table 2 summarizes the configuration for slices in each category, including the enterprise scheduler, workloads per user, and

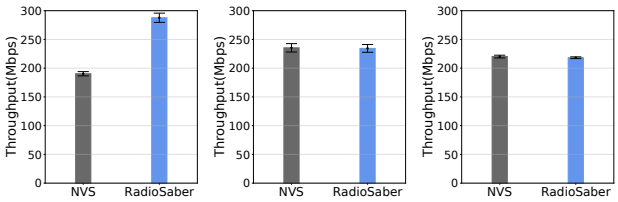
Slices	Metrics	RadioSaber	NVS	No-Slicing
1-5	throughput (Mbps)	13.02	8.45	**17.41**
6-10	average FCT(s)	**2.606**	5.708	5.714
11-15	average FCT(s)	**0.489**	1.686	2.988
16-20	average queueing delay(s)	**0.061**	1.493	0.696

Table 3: Experiment results w.r.t different metrics in all slices of RadioSaber and baselines.

metrics: (i) Slices 1-5 use PF based enterprise scheduling policy. Every user in a slice instantiates a backlogged flow, and we measure the average aggregate throughput as the metric. (ii) Slices 6-10 also use PF based enterprise scheduling policy. Each slice in this category generates flows with Poisson inter-arrival time, arriving at an average rate of 12Mbps, following a heavy-tailed Internet flow distribution [28]. It randomly and evenly assign these flows to its UEs. We measure the FCT(Flow Completion Time) of flows as the evaluation metric. (iii) Slices 11-15 use the same workload generator as the previous category, but assigns a higher priority to 25% of the flows (that are randomly selected). We measure the FCT of prioritized flows as the key metric. (iv) For slices 16-20, each user streams a real-time video flow with 1280kbps average bitrate. Each slice applies the M-LWDF policy for enterprise scheduling, to ensure low packet delays for the real-time streaming. We compute per-packet queuing delays as the key evaluation metric. We configure the weights of slices 16-20 to be 2× higher than the weights of the other slices.

Table 3 summarizes the results (aggregating the slice-specific metrics across all slices in each category). We also present the CDF of different metrics, to highlight the performance differences at different percentiles in the Appendix§A.2.

We find that RadioSaber consistently outperforms NVS across all slice-specific metrics, with 1.5× higher throughput for slices 1-5, 2-4× lower FCT for slices 6-15 and 24× lower packet delays for slices 16-20. This shows that the throughput wins of RadioSaber directly translate to other relevant metrics such as flow completion time and packet delays. No-Slicing achieves higher throughput than RadioSaber for slices 1-5 which have backlogged flows, but fares significantly worse in the performance metrics for other slices (with 2-7× higher FCT in slices 6-15 and 10× higher packet delays in slices 16-20). This is because: (i) No-Slicing cannot provide isolation across slices and correctly enforce weighted fairness. Instead, it ends up allocating a larger share of RBs to the first category of slices (1-5) which have more number of active users at any given time (as they are all backlogged). (ii) No-Slicing's PF policy is not compliant with the requirements of other slices (i.e. prioritization for slices 11-15 and low packet delays for slices 16-20)



(a) Real Trace (b) Synthetic: no variation in sub-band CQI (c) Synthetic: non-complementary subband CQI

Figure 8: Experiments with different types of CQI traces

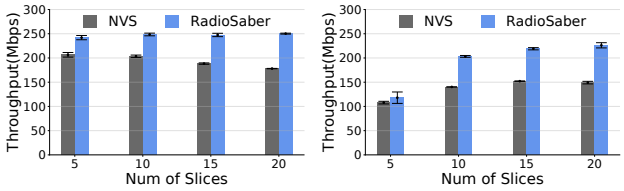
6.3 What Makes RadioSaber Win Over NVS?

We now validate the intuition discussed in §3, that the throughput gain from channel-aware inter-slice scheduling originates from the complementary channel quality distribution between slices. We use the same setting as that in §6.1, except that we generate synthetic CQI reports using the default channel propagation module in our RAN simulator that models losses due to multipath [19], path loss, penetration and shadowing [34]. Fig. 8(a) shows experiment results from real traces for comparison (these results are the same as those presented in §6.1).

In the first experiment, we exclude the multipath loss to make the channel quality same across all RBs (although the channel quality still varies across different UEs). Channel quality variations over time happen at granularity of 40ms (the CQI reporting interval), which translates to 160 TTIs. Fig. 8(b) shows that NVS and RadioSaber both achieve the same throughput. If we consider a 2D grid spanning 20 TTIs (number of slices in our setup), the channel quality for a UE neither varies in the frequency domain, nor in the time domain – so, as long as all slices are assigned RBs as per their quotas, it does not matter which RBs they get assigned.

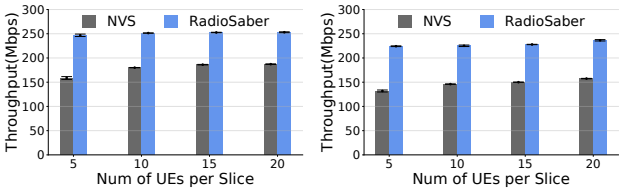
In the second experiment, we synthetically make the sub-band channel distribution non-complementary. For this, we exclude the multipath loss, and manually decrease SNR of the first 50MHz channel by 10dB and increase SNR of the lower 50MHz channel by 10dB. What this implies is that all UEs have equally high channel quality for the first half of the RBs in the frequency domain, and equally low channel quality for the second half. Fig. 8(c) shows that NVS and RadioSaber again achieve similar throughput with this "non-complementary" CQI distribution. This is because, with both NVS and RadioSaber, each RB will produce the same datarate (that is either high or low), no matter which slice/UE it gets assigned to.

These results highlight that RadioSaber wins over NVS when (i) the CQIs differ across UEs in the frequency domain, and (ii) the CQIs for a RB across different UEs complement one another (i.e. if some UEs have low CQIs for a given RB, there are other UEs having high CQIs for it). We found this to be the case for the LTE cellular traces used in our experiments, and expect these trends to be stronger for 5G with larger bandwidths.



(a) backlogged flows (b) heavy-tail distributed flows

Figure 9: Varying the number of slices (5-15 UEs per slice).



(a) backlogged flows (b) heavy-tail distributed flows

Figure 10: Varying the number of UEs per slice for 20 slices.

6.4 Varying Number of Slices and UEs per Slice

We next evaluate the robustness RadioSaber's throughput wins over NVS, as we vary the number of slices and the number of users in each slice. For simplicity of analyzing the results, we configure each slice to apply the PF enterprise scheduling policy. We experiment with two types of workloads: (i) each user maintains a backlogged flow; (ii) the heavy-tail distribution of flows across UEs (as described in §6.2), and an average load of 24Mbps per slice.

Varying number of slices. We increase the number of slices from 5 to 20 and randomly assign between 5-15 users in each slice. In Fig. 9(a), we find that with backlogged flows, the aggregate throughput remains almost unchanged in RadioSaber as we increase the number of slices, but the aggregate throughput keeps dropping in NVS. It shows that the spectrum efficiency of RadioSaber remains unaffected and scales well with the number of slices. RadioSaber achieves 17.2%-40.5% higher throughput than NVS.

We find some interesting trends in Fig. 9(b), with non-backlogged flows. RadioSaber is able to make better use of the spectrum as the number of slices increase, which increases the number of active users (with diverse and complementary subband CQI distributions) that can be scheduled per TTI. NVS does not experience a similar trend, and in fact, has a significantly lower aggregate throughput compared to the backlogged flows scenario in Fig. 9(a). This is because NVS allocates all RBs in a TTI to the same slice, and since there are fewer active users in a slice in this scenario, the multi-user diversity gain reduces, which reduces the throughput achieved with the PF scheduler. RadioSaber achieves 37.3%-50% higher throughput than NVS in this context for 10-20 slices. Note that when there are 5 slices, the throughput is limited by the total incoming network traffic.

Varying number of UEs. In the second experiment, we fix the number of slices to 20, and increase the number of UEs per slice. In Fig. 10(a), the aggregate throughput of NVS increases a little when the number of users per slice increases. This is reasonable since more users bring more diverse subband channel quality distributions for the PF enterprise scheduler in NVS to allocate RBs smartly. However, RadioSaber still outperforms NVS by 35.2%-56.8%. From Fig. 10(b), RadioSaber outperforms NVS by 49.7%-72.2% when the flows are heavy-tail distributed.

6.5 Non-greedy Enterprise Schedulers

Our evaluation so far only considered greedy enterprise schedulers. Indeed, most proposed (and potentially deployed) radio resource schedulers employ a greedy heuristic to achieve low runtime overheads [4]. We now compare how RadioSaber (using a greedy enterprise scheduler for PF) compares against NVS using a non-greedy heuristic for PF [17]. We briefly explain the non-greedy heuristic before presenting our results.

A UE can only use a single MCS(Modulation and Coding Scheme) across all RBs that are allocated to it. The typical practice is to first assign RBs to users, and then compute the MCS based on effective SNR across these RBs. The final data-rate so achieved is less than the sum of the data-rates achieved if one could use the optimal MCS value for each RB (our experimental results take this effect into account). Given this effect, GPF [17] uses a non-greedy heuristic to co-optimize the MCS assignment and the RB allocation – it samples 300 plausible mappings between each user and a corresponding MCS value (assuming that all RBs are available for a given user), and computes the best RB allocation for each of these MCS mappings. It then selects the MCS mapping and RB allocation that achieves the highest PF metric.

We evaluate how NVS using the above non-greedy PF scheduler for each slice compares against NVS and RadioSaber using greedy PF enterprise schedulers. We fix the number of slices to 20 and vary the number of UEs in each slice. Fig. 11 reports the overall throughput across the three schemes. We find that NVS with non-greedy PF achieves 13%-19% higher aggregate throughput than NVS with greedy PF. This better spectrum efficiency results from the tactical assignment of RBGs and MCS to users. However, the aggregate throughput of NVS (non-greedy PF) is still 14%-18% lower than RadioSaber since NVS is channel-unaware.

6.6 Is There a Better Inter-Slice Scheduler?

We now evaluate two questions:

(i) What is the impact of assigning RBGs in the order of decreasing channel quality? For this, we modify RadioSaber's inter-slice scheduler to greedily assign RBGs to the slices with maximum channel quality in a sequential order (from top to bottom). We term this strategy as "Sequential". With this approach, the inter-slice scheduler can query enterprise schedulers to determine the channel quality when it

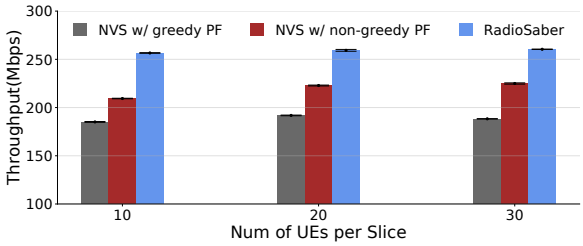


Figure 11: Comparing RadioSaber (using a greedy PF scheduler) with NVS (using a non-greedy PF scheduler). We fix number of slices to 20, and vary the number of UEs per slice.

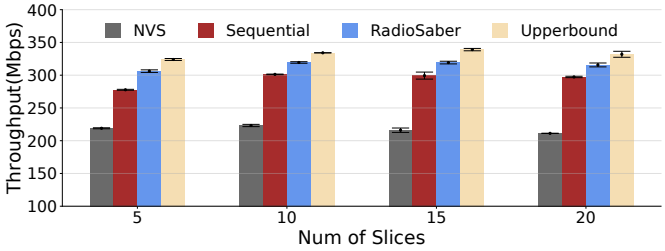


Figure 12: Comparing RadioSaber with a simple greedy inter-slice scheduler, and with a contrived upperbound. We vary the number of slices, with 5-15 random users in each slice, and use MT scheduling policy in each slice.

assigns RBGs, which avoids the need for recomputing channel quality after each allocation (as in the original inter-slice scheduling algorithm), resulting in a lower time complexity of $O(|\mathcal{RBG}||S|T)$.

(ii) Could we have achieved a better allocation? For this we compare RadioSaber with an impractical scheme that gives us an upper-bound on the spectrum efficiency that any inter-slice scheduler can achieve. In this scheme, we greedily allocate the RBG with the best channel quality to each slice until the slice's quota is exhausted. However, in doing so we allow a given RBG to be repeatedly allocated to multiple slices. As a result, each slice gets its best set of RBGs, independent of the allocation for other slices. Note that this upperbound still enforces fairness between slices, and allows customizable enterprise scheduling within each slice.

To make it easier to analyze the inter-slice schedulers we configure the enterprise scheduler in each slice to use the MT policy. We evaluate the overall throughput as we vary the number of slices, with 5-15 users in each slice. Fig. 12 shows that RadioSaber performs only 4%-6% worse than Upperbound and 6%-10% better than Sequential. We see similar trends across other settings (e.g. using PF policy in each slice, varying the number of users, etc).

The key takeaways are: (a) RadioSaber's inter-slice scheduling policy is close to optimal, and (b) Even the simple channel-aware greedy inter-slice scheduler achieves 25%-36% better throughput than channel-agnostic NVS, and is a good option if lower time complexity is required, at a cost of some throughput penalty compared to our current algorithm.

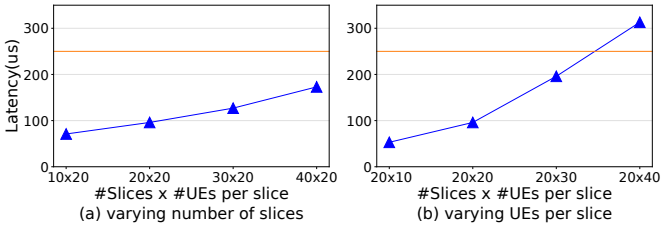


Figure 13: RadioSaber's scheduling latency

6.7 Scheduling Latency and Overhead

To evaluate RadioSaber's scheduling latency and runtime overhead, we implement its MAC scheduling logic on a system using a single Intel Xeon core. For simplicity of analysis, we configure each slice to apply the PF scheduling policy with backlogged users. Fig. 13(a) shows how the scheduling latency increases linearly with the number of slices when the number of UEs per slice is fixed to 20. Similarly, Fig. 13(b) shows how the scheduling latency increases linearly with the number of UEs per slice when the number of slices is fixed to 20. In both cases, the scheduling system can support as many as 600 users and make the scheduling decision within the stringent TTI constraint (250us).

7 Limitations and Future Work

• So far, we have only considered scheduling for downlink radio resources and not for uplink. The uplink channel applies SC-FDMA [5], which is similar to OFDMA and allows radio resources allocation in both time and frequency domain. This indicates that RadioSaber can be extended to uplink scheduling. However, it requires more complicated control information exchange between UEs and the base station. We leave a detailed exploration of this to future work.

• We only consider scheduling radio resources in a channel-aware manner for a single gNB. An interesting future direction is to extend our work in the context of multiple gNBs with small cells. The problem expands to a 3D allocation of frequency, time, and space resources in a channel aware manner.

Acknowledgement

We would like to thank our shepherd, Ganesh Ananthanarayanan, and the anonymous NSDI reviewers for their insightful comments and feedback. We're grateful to Ammar Tahir, Emerson Sie for their feedback in the camera-ready version. We would also like to thank Yaxiong Xie for sharing the LTE SNR traces with us. This work was supported by Intel, Facebook, AG NIFA under grant 2021-67021-34418, and UIUC's Smart Transport Infrastructure Initiative.

References

[1] Open5gs is a c-language open source implementation for 5g core and epc. https://github.com/open5gs, 2021.

[2] M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, P. Whiting, and R. Vijayakumar. Providing quality of service over a shared wireless link. *IEEE Communications Magazine*, 39(2):150–154, 2001.

[3] G. S. Association. 5g network slicing self-management white paper. https://www-file.huawei.com/-/media/corporate/pdf/news/5g-network-slicing-self-management-white-paper.pdf?la=en-us, 2020.

[4] F. Capozzi, G. Piro, L. Grieco, G. Boggia, and P. Camarda. Downlink packet scheduling in lte cellular networks: Key design issues and a survey. *IEEE Communications Surveys Tutorials*, 15(2):678–700, 2013.

[5] ETSI. 5g; 5g system; network slice selection services;. https://www.etsi.org/deliver/etsi_ts/129500_129599/129531/15.01.00_60/ts_129531v150100p.pdf, 2018.

[6] ETSI. 5g; nr; base station (bs) radio transmission and reception. https://www.etsi.org/deliver/etsi_ts/138100_138199/138104/15.02.00_60/ts_138104v150200p.pdf, 2018.

[7] ETSI. 5g nr: Physical channels and modulation(3gpp ts 38.211 version 16.2.0 release 16). https://www.etsi.org/deliver/etsi_ts/138200_138299/138211/16.02.00_60/ts_138211v160200p.pdf, 2020.

[8] ETSI. Evolved universal terrestrial radio access (e-utra); physical layer procedures. https://www.etsi.org/deliver/etsi_ts/136200_136299/136213/15.10.00_60/ts_136213v151000p.pdf, 2020.

[9] X. Foukas, M. K. Marina, and K. Kontovasilis. Orion: Ran slicing for a flexible and cost-effective multi-service mobile network architecture. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 127–140, New York, NY, USA, 2017. Association for Computing Machinery.

[10] X. Foukas, N. Nikaein, M. M. Kassem, M. K. Marina, and K. Kontovasilis. Flexran: A flexible and programmable platform for software-defined radio access networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 427–441, New York, NY, USA, 2016. Association for Computing Machinery.

[11] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina. Network slicing in 5g: Survey and challenges. *IEEE Communications Magazine*, 55(5):94–100, 2017.

[12] J. García-Morales, M. C. Lucas-Estañ, and J. Gozalvez. Latency-sensitive 5g ran slicing for industry 4.0. *IEEE Access*, 7:143139–143159, 2019.

[13] M. Gidlund and J.-C. Laneri. Scheduling algorithms for 3gpp long-term evolution systems: From a quality of service perspective. In *2008 IEEE 10th International Symposium on Spread Spectrum Techniques and Applications*, pages 118–123, 2008.

[14] M. K. Giluka, N. Rajoria, A. C. Kulkarni, V. Sathya, and B. R. Tamma. Class based dynamic priority scheduling for uplink to support m2m communications in lte. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 313–317, 2014.

[15] T. Guo and A. Suárez. Enabling 5g ran slicing with edf slice scheduling. *IEEE Transactions on Vehicular Technology*, 68(3):2865–2877, 2019.

[16] M. B. Hcine and R. Bouallegue. Analytical downlink effective sinr evaluation in lte networks. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, pages 376–381, 2015.

[17] Y. Huang, S. Li, Y. T. Hou, and W. Lou. Gpf: A gpu-based design to achieve 100 us scheduling for 5g nr. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 207–222, New York, NY, USA, 2018. Association for Computing Machinery.

[18] H. Hui, Y. Ding, Q. Shi, F. Li, Y. Song, and J. Yan. 5g network-based internet of things for demand response in smart grid: A survey on application potential. *Applied Energy*, 257:113972, 2020.

[19] W. C. Jakes and D. C. Cox. *Microwave Mobile Communications*. Wiley-IEEE Press, 1994.

[20] E. Kahuha. 5 real life use cases of 5g ultra-reliable low-latency communication (urllc). https://www.section.io/engineering-education/five-real-life-use-cases-of-5g-ultra-reliable-low-latency-communication-urllc/, 2021.

[21] R. Kokku, R. Mahindra, H. Zhang, and S. Rangarajan. Nvs: A virtualization substrate for wimax networks. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, page 233–244, New York, NY, USA, 2010. Association for Computing Machinery.

[22] R. Kokku, R. Mahindra, H. Zhang, and S. Rangarajan. Nvs: A substrate for virtualizing wireless resources in cellular networks. *IEEE/ACM Trans. Network.*, 20(5), oct 2012.

[23] R. Kokku, R. Mahindra, H. Zhang, and S. Rangarajan. Cellslice: Cellular wireless resource slicing for active ran sharing. In *2013 Fifth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–10, 2013.

[24] R. Kwan, C. Leung, and J. Zhang. Proportional fair multiuser scheduling in lte. *IEEE Signal Processing Letters*, 16(6):461–464, 2009.

[25] H. Liu, L. Cai, H. Yang, and D. Li. Eesm based link error prediction for adaptive mimo-ofdm system. In *2007 IEEE 65th Vehicular Technology Conference - VTC2007-Spring*, pages 559–563, 2007.

[26] R. Mahindra, M. A. Khojastepour, H. Zhang, and S. Rangarajan. Radio access network sharing in cellular networks. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2013.

[27] C. Marquez, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez. How should i slice my network? a multi-service empirical evaluation of resource sharing efficiency. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 191–206, New York, NY, USA, 2018. Association for Computing Machinery.

[28] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively cautious congestion control. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 373–385, USA, 2014. USENIX Association.

[29] A. Nakao, P. Du, Y. Kiriha, F. Granelli, A. A. Gebremariam, T. Taleb, and M. Bagaa. End-to-end network slicing for 5g mobile networks. *Journal of Information Processing*, 25:153–163, 2017.

[30] N. Nikaein, E. Schiller, R. Favraud, K. Katsalis, D. Stavropoulos, I. Alyafawi, Z. Zhao, T. Braun, and T. Korakis. Network store: Exploring slicing in future 5g networks. MobiArch '15, page 8–13, New York, NY, USA, 2015. Association for Computing Machinery.

[31] A. Oborina, T. Henttonen, V. Koivunen, and M. Moisio. Efficient computation of effective sinr. In *2012 46th Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6, 2012.

[32] M. Olsson, S. Sultana, S. Rommer, L. Frid, and C. Mulligan. Chapter 6 - session management and mobility. In M. Olsson, S. Sultana, S. Rommer, L. Frid, and C. Mulligan, editors, *SAE and the Evolved Packet Core*, pages 97–140. Academic Press, Oxford, 2010.

[33] L. Peterson and O. Sunay. *5G Mobile Networks: A Systems Approach*. USA, 2020.

[34] G. Piro, L. A. Grieco, G. Boggia, F. Capozzi, and P. Camarda. Simulating lte cellular systems: An open-source framework. *IEEE Transactions on Vehicular Technology*, 60(2):498–513, 2011.

[35] Qualcomm. Future of 5g: Building a unified, more capable 5g air interface for the next decade and beyond. https://www.qualcomm.com/media/documents/files/making-5g-nr-a-commercial-reality.pdf, 2020.

[36] I. Qualcomm Technologies. Vr and ar pushing connectivity limits. https://www.qualcomm.com/media/documents/files/vr-and-ar-pushing-connectivity-limits.pdf, 2018.

[37] A. Rao. 5g network slicing: crossdomain orchestration and management will drive commercialization. https://www.cisco.com/c/dam/en/us/products/collateral/cloud-systems-management/network-services-orchestrator/white-paper-sp-5g-network-slicing.pdf, 2020.

[38] J.-H. Rhee, J. Holtzman, and D.-K. Kim. Scheduling of real/non-real time services: adaptive exp/pf algorithm. In *The 57th IEEE Semiannual Vehicular Technology Conference, 2003. VTC 2003-Spring.*, volume 1, pages 462–466 vol.1, 2003.

[39] B. Sadiq, R. Madan, and A. Sampath. Downlink scheduling for multiclass traffic in lte. *EURASIP J. Wirel. Commun. Netw.*, 2009, mar 2009.

[40] J. X. Salvat, L. Zanzi, A. Garcia-Saavedra, V. Sciancalepore, and X. Costa-Perez. Overbooking network slices through yield-driven end-to-end orchestration. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 353–365, New York, NY, USA, 2018. Association for Computing Machinery.

[41] K. Samdanis, X. Costa-Perez, and V. Sciancalepore. From network sharing to multi-tenancy: The 5g network slice broker. *IEEE Communications Magazine*, 54(7):32–39, 2016.

[42] sharetech. Resource allocation type. https://www.sharetechnote.com/html/Handbook_LTE_RAType.html, 2020.

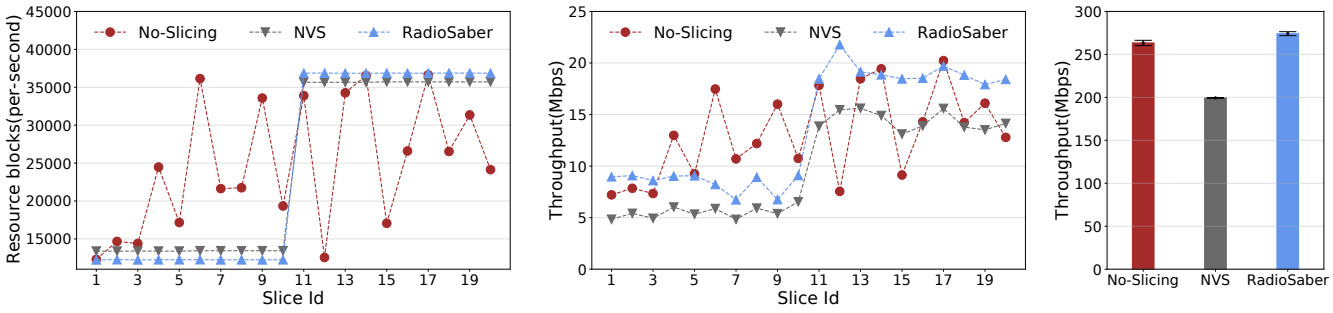
[43] Techplayon. 5g nr resource block definition and rbs calculation. https://www.techplayon.com/nr-resource-block-definition-and-rbs-calculation/, 2019.

[44] N. Van Giang and Y. H. Kim. Slicing the next mobile packet core network. In *2014 11th International Symposium on Wireless Communications Systems (ISWCS)*, pages 901–904, 2014.

[45] C. Wengerter, J. Ohlhorst, and A. von Elbwart. Fairness and throughput analysis for generalized proportional fair frequency scheduling in ofdma. In *2005 IEEE 61st Vehicular Technology Conference*, volume 3, pages 1903–1907 Vol. 3, 2005.

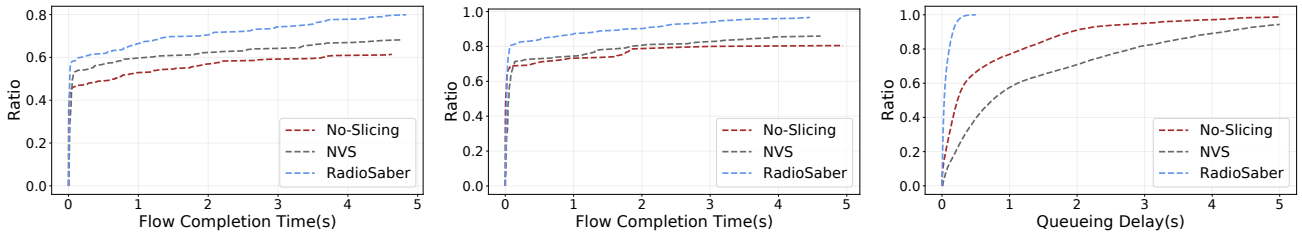
[46] Y. Xie, F. Yi, and K. Jamieson. Pbe-cc: Congestion control via endpoint-centric, physical-layer bandwidth measurements. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 451–464, New York, NY, USA, 2020. Association for Computing Machinery.

[47] M. Yan, G. Feng, J. Zhou, Y. Sun, and Y.-C. Liang. Intelligent resource scheduling for 5g radio access network slicing. *IEEE Transactions on Vehicular Technology*, 68(8):7691–7703, 2019.



(a) per-slice RBs per second (b) per-slice throughput (c) sum total throughput

Figure 14: (a) the average allocated RBs per second to slices; (b) the average aggregate throughput of slices; (c) the sum total throughput of three systems



(a) All flows in slice 6-10 (b) Prioritized flows in slice 11-15 (c) All real-time video flows

Figure 15: CDFs of (a) completion time of flows in slices 6-10, (b) completion time for the higher prioritized flows in slices 11-15, and (c) queuing delay of packets in slices 16-20. We cut the graphs at 5s to better highlight the trends.

A Supplemental Evaluation

A.1 Slices with Different Weights

§6.1 evaluated the scenario where every slice has the same weight. Here we do the same experiment but slightly modify SLAs of slices: slices with PF schedulers have 3X higher weights than slices with MT schedulers. Fig. 14 shows the experiment results. From Fig. 14(a) and Fig. 14(b), it's obvious that the last 10 slices get 3X more RBs than the first 10 slices, and approximately 3X higher throughput. Meanwhile, it's impossible for No-Slicing to meet the SLAs of slices since it doesn't support network slicing between groups of users at all.

A.2 CDF Graphs of FCT and Queueing Delay

We provide more evaluation results in §6.2. Fig. 15(a) shows the CDF of flow completion time in the slice 6-10, and Fig. 15(b) shows the CDF of flow completion time for the higher priority flows in slices 11-15, and Fig. 15(c) shows the CDF of queuing delay for slices 16-20. We cut the graphs at 5s to better highlight the trends. For flows in slices 6-15, the FCT in No-Slicing is the longest since No-Slicing cannot enforce fairness for slices in which flows arrive intermittently. NVS suffers from the longest queuing delay in slices 16-20 due to low spectrum efficiency.

RF-CHORD: Towards Deployable RFID Localization System for Logistics Network

Bo Liang[ℙ𝔸], *Purui Wang*[𝕄], *Renjie Zhao*[𝕌], *Heyu Guo*[ℙ], *Pengyu Zhang*[𝔸], *Junchen Guo*[𝔸]
Shunmin Zhu[𝕋𝔸], *Hongqiang Harry Liu*[𝔸], *Xinyu Zhang*[𝕌], *Chenren Xu*[ℙℤ𝕂✉] *

[ℙ]*Peking University* [𝔸]*Alibaba Group* [𝕄]*Massachusetts Institute of Technology* [𝕌]*University of California San Diego* [𝕋]*Tsinghua University*
[ℤ]*Zhongguancun Laboratory* [𝕂]*Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Abstract – RFID localization is considered the key enabler of automating the process of inventory tracking and management for the high-performance logistic network. A practical and deployable RFID localization system needs to meet *reliability*, *throughput*, and *range* requirements. This paper presents RF-CHORD, the first RFID localization system that simultaneously meets all three requirements. RF-CHORD features a multisine-constructed wideband design that can process RF signals with a 200 MHz bandwidth in real-time to facilitate one-shot localization at scale. In addition, multiple SINR enhancement techniques are designed for range extension. On top of that, a kernel-layer near-field localization framework and a multipath-suppression algorithm are proposed to reduce the 99th long-tail errors. Our empirical results show that RF-CHORD can localize up to 180 tags 6 m away from a reader within 1 second and with 99th long-tail error of 0.786 m, achieving a 0% miss reading rate and ~0.01% cross-reading rate in the warehouse and fresh food delivery store deployment.

1 Introduction

Today's major e-commerce companies like Alibaba and Amazon need to handle a package volume that is tens of billions per year [1], calling for increasingly high-performance automated logistics operations in their network. Considering a typical warehouse in which tens or even hundreds of packages pass through each checkpoint – the packages need to be verified, recorded, sorted, and tracked when checking in/out. In widely adopted barcode-based logistic networks, the worker spends 1~3 seconds on scanning one package. Although this operation can be automated by robots [2], the line-of-sight and field of view requirements of vision-based approaches limits work range and scalability fundamentally. RFID technology, since its invention, has been carrying the vision of replacing inefficient labor and automating inventory management with zero power, near-zero cost, and high throughput.

Towards a highly practical and deployable RFID empowered automated logistic network shown in Fig. 1, there are three key considerations: *i) Reliability.* The classic ROI (range of interest) reading task requires the reader to scan all the RFID tags within the ROI (*i.e.,* near-zero miss-reading rate) while excluding any tag out of the ROI (*i.e.,* near-zero cross-reading rate); *ii) Throughput.* The packages come to the checkpoint in a burst (*i.e.,* 100~200 per pallet) [1] while all the logistic operations, including verification and recording need to be finished within 2~3 seconds before check-in/out; *iii) Range.* A single reader should cover tags within 3~5 m, which is the typical width of the check-in/out aisle.

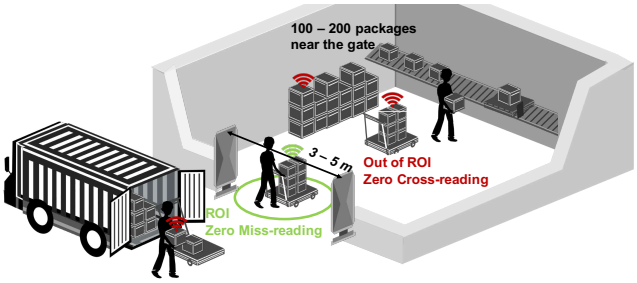


Figure 1: In a typical logistic scenario, the packages are discharged from the truck, scanned at an inventory gate and sorted for warehouse check in. The RFID-based inventory gate should meet *reliability*, *throughput*, and *range* requirements at the same time.

Unfortunately, today's read-or-not inventory systems, both industrial products and research prototypes, all have limitations in meeting these three requirements simultaneously. Industry-grade RFID systems (*e.g.,* Impinj) suffer from miss-reading and cross-reading when deployed in the logistic warehouses. RFGo [3] reports 99.8% recall with 10 carrier-level synchronized antennas and neural network based classifier but limits its operating range to sub-meter. NFC+ [4] achieves a sharp inventory boundary with magnetic resonance engineering that meets the reliability (*i.e.,* miss-reading rate of 0.03% and cross-reading rate of 0%) and range (~3 m) requirements but cannot achieve the desired throughput. No current inventory-based solutions can support automatic package management in a practical logistics network.

RFID localization technique offers an alternative approach toward the same goal by filtering out the reading outside the ROI. Compared with the inventory-based system, the tag location brings a new dimension of information, which can realize a more flexible and accurate ROI reading. The reliability of

*Bo Liang and Purui Wang are the co-primary student authors. Purui Wang was affiliated to Peking University and Alibaba Group during which he contributed to this work.

✉: chenren@pku.edu.cn

[1] Even though one trailer can carry up to 50 packages, the reader should be able to cover all the tags (100~200 tags) near the gate (including passed trailer and undischarged packages) to ensure to read all the passing packages.

ROI reading depends on localization accuracy. However, the legitimate narrow frequency band (*i.e.,* 26 MHz ISM band within 902~928 MHz) of RFID fundamentally limits its capacity of combating multipath and ambiguity [5]. To improve the localization accuracy, approaches like fingerprinting [6] and synthetic aperture radar (SAR) based hologram [7, 8] have been proposed. However, they suffer from prolonged latency due to lots of tag inventory, especially at scale. Cross-frequency based approaches utilize higher frequency band to overcome the bandwidth limitation (*e.g.,* 2.4 GHz [9, 10], millimeter wave [11], UWB [12, 13]) but introduce extra tag manufacturing cost due to wider frequency response and higher power attenuation. More recently, sniffer-based RFID architecture [14, 15] has been proposed to leverage the advantage of wideband (*e.g.,* 100~200 MHz) near 915 MHz to boost location accuracy without violating FCC regulation. Despite the potential, these systems either suffer from latency issues due to the lack of hardware support on multi-band parallel information capture [14], or report limited sub-meter range [15].

This paper introduces the design and implementation of RF-CHORD, an active sniffer-based wideband RFID localization system that tackles the above challenges. RF-CHORD exploits wideband signal and a hologram-based localization algorithm to realize *high reliability*. It employs lossless data stream compression and a GPU-based decoder to guarantee real-time decoding and channel estimation for *high throughput*. It utilizes a customized wideband waveform, full packet matching integration, fine-grained clock offset mitigation, and channel diversity decoding to improve SINR for *long range*.

RF-CHORD ensures *high reliability* (*i.e.,* near zero miss reading and cross reading) by high-accuracy localization. Our study (§5.1) shows that the multipath profile causes long-tail localization errors. Therefore, we design the fine-grained distance resolution hardware and multipath-suppression algorithm to handle these long-tail errors. Considering that the distance resolution is inversely proportional to bandwidth (*i.e.,* $\frac{c}{2B}$), the distance resolution of a conventional UHF RFID reader, which works on a 26 MHz wide ISM band, is only 5.78 m. RF-CHORD introduces an extra active sniffer-based reader to help UHF RFID reader realize 200 MHz parallel wideband localization (§3.2). However, the distance resolution of 200 MHz (0.75 m) is still not enough in all situations. RF-CHORD exploits a kernel-layer-based near-field localization algorithm framework to handle corner cases. The kernel function characterizes the location estimation from a single channel, and layer functions coherently combine multiple channels into a final location estimation. This framework supports choosing different kernel and layer functions suitable for various deployment scenarios to achieve multipath suppression and ambiguity reduction (§5.3). For example, in RF-CHORD's deployment in the warehouse, the work range is fixed so it can be taken as prior information for direct path enhancement to effectively suppress the multipath effect (§5.4).

RF-CHORD ensures *high throughput* by one-shot channel measurement and one-shot location estimation. The hardware supports concurrent phase and amplitude capture across multiple antennas and wide bandwidth. Therefore, RF-CHORD can obtain the necessary information (*i.e.,* wideband channel estimation across multiple antennas) for localization within only one shot measurement. It is challenging because: i) directly capturing the wideband signal from a large array will result in a huge amount of real-time data (~64 Gbps); ii) the commercial reader does not support real time synchronization (*i.e.,* synchronizing with our sniffer-based reader at each slot [18]). Utilizing the essence that the wideband backscattered signal is a combination of scattered narrowband signals, RF-CHORD distills 4 MHz valid bandwidth from 200 MHz bandwidth to reduce the data rate by 50x without information loss (§3.4). Meanwhile, we develop a GPU-based wideband decoder to ensure real time decoding and channel estimation. In other words, the sniffer-based reader has an independent decoder and does not depend on any specific commercial reader interface. It makes our design adaptive to any ISM band commercial reader, which primarily serves as a power activator and multiple access handler (§4). Finally, RF-CHORD supports one-shot localization with 8 antennas and 16 frequencies across 200 MHz in ~5 ms.

RF-CHORD ensures *long range* (up to 6 m) with multisine waveform sniffer and sophisticated wideband channel information estimation. To follow the FCC regulation, the strength of the sniffer excitation signal needs to be *smaller than -13.3 dBm* (see §A for the calculation), which is 50 dB weaker than that of commercial readers. RF-CHORD features the following designs for signal-to-interference-plus-noise ratio (SINR) enhancement without modifying the tag chip: i) It exploits a multisine waveform, which constructs a whole 200 MHz band by taking samples with multiple narrow bands, to significantly reduce the noise bandwidth (§4.1); ii) It handles the high dynamic range requirements introduced by self-interference through high-resolution digital channelization and a low crest factor waveform design (§4.2); iii) It further exploits the integration gain of full packet matching (§4.3) and performs accurate tag clock offset mitigation (§4.4) and decoding with channel diversity (§4.5).

We deploy RF-CHORD and our results show that RF-CHORD presents the first RFID (localization) system meeting all the requirements (*i.e.,* reliability, throughput, and range) in the logistic network (Tab. 1). The key results are:

• **Reliability.** We evaluate RF-CHORD's performance at 384 locations and collect over 20k tag responses in the lab environments. Its 99% localization error is 0.786 m. We deploy RF-CHORD in the dock door of a warehouse and the scanning gate of a fresh food delivery store. We find that it could read 100% of the tags passing the checkpoint (0% miss-reading rate). Its cross-reading rate is only 0.0025%~0.0154%, which is up to 12x improvement compared to state-of-the-art [3, 4].

• **Throughput.** RF-CHORD can localize up to 180 tags per second, which is very close to pure inventory devices [16] and

Solutions \ Requirements	Throughput (> 100 tags/s)	Range (> 3 m)	Reliability (Near Zero Miss-reading & Cross-reading)	Commercial Tag
Barcode (widely deployed)	No (~1 tag per second)	No (~1 m)	High (depend on the human labor)	Yes
xSpan [16] (Inventory based)	Yes (~185 tags/s with 142 mode)	Yes (~10 m)	Low (~6% miss reading and ~2% cross reading)	Yes
RFgo [3]	No (TDMA-based)	No (sub-meter)	High (99.8% recall)	Yes
NFC+ [4]	No data reported	Yes (~3 m)	High (0% miss reading and ~0.03% cross reading)	No
PinIt [6]	No data reported	Yes (> 5 m)	Median (a few decimeters)	Yes
RF-IDraw [17]	No data reported	Yes (> 5 m)	Low (sub-meter)	Yes
Tagoram [7]	No (0.2 second for one tag)	No (~2 m)	Median (a few decimeters)	Yes
MobiTagbot [8]	No data reported	No (~1.5 m)	High (a few centimeters)	Yes
NLTL tags [9]	No (depend on switching)	No (~1 m)	High (a few millimeters)	No
mmwave RFID [11]	No data reported	No data reported	Median (a few decimeters)	No
RFind [14]	No (6.4 second for one tag)	Yes (> 5 m)	High (a few centimeters)	Yes
TurboTrack [15]	No data reported	No (sub-meter)	High (a few centimeters)	Yes
RF-CHORD (Our system)	**Yes (180 tags/s)**	**Yes (6 m)**	**High (0% miss reading and ~0.01% cross reading)**	**Yes**

Table 1: Comparing RF-CHORD with state-of-the-art wireless systems for logistic network requirements.

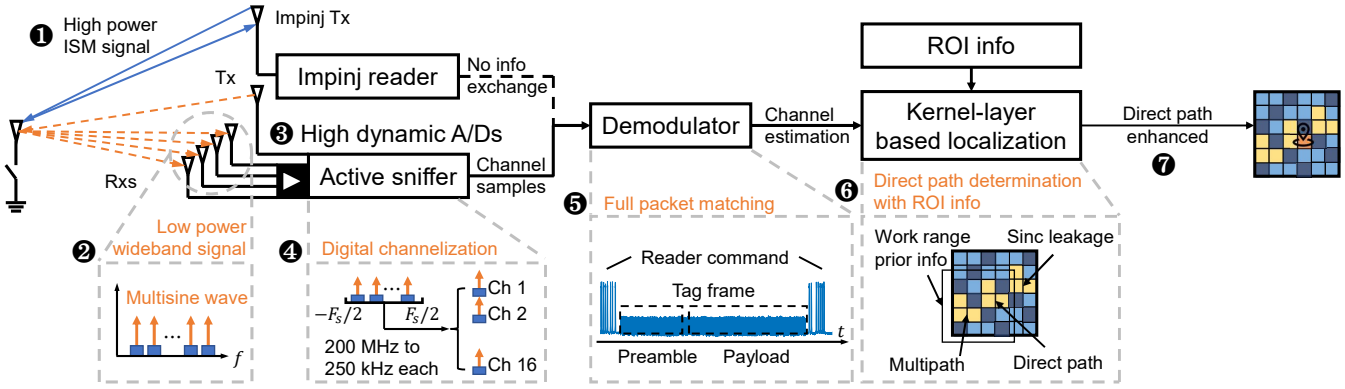


Figure 2: RF-CHORD system overview.

two to three orders of magnitude faster than state-of-the-art localization systems [7, 14].

• **Range.** RF-CHORD can localize tags 6 m away from the reader with transmit power below -15 dBm. There is no obvious throughput and reliability loss with distance increasing.

We open sourced the RF-CHORD's hardware and software as well as the evaluation dataset in https://soar.group/projects/rfid/rfchord.

2 RF-CHORD's System Overview

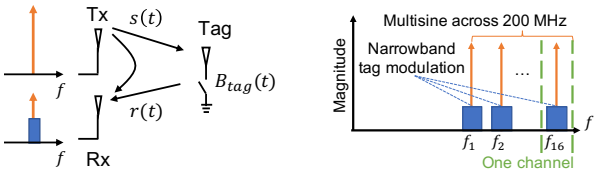
A high level operational flow of RF-CHORD is shown in Fig. 2. RF-CHORD embraces any ISM-band reader ❶ as the tag activator that is capable of charging, coordinating multiple access over EPC Gen II tags. Active sniffer reader observes tags by emitting a low power (-15 dBm) wideband multi-sine waveform to pick up tag responses over a wide frequency band. Specifically, we build the RF frontend and FPGA hardware ❷ as a scalable platform that can receive the tag response from 8 antennas and 16 frequencies of carriers simultaneously. Furthermore, despite the strict legal emission power limit, we still achieve a long range in sniffing the tag response in the wideband without exchanging any information (*e.g.,* EPC ID) with the ISM-band reader. RF-CHORD achieve independent decoding and channel estimation by using dynamic range optimization ❸, digital channelization ❹ in hardware, and a real-time full packet matching ❺ in software. After one-shot tag inventory, RF-CHORD obtains adequate information from both frequency and spatial domains, which are important for robust localization in a multipath-rich environment. RF-CHORD also uses a kernel-layer-based near-field localization algorithm to suppress the multipath effect. This algorithm identifies the direct path with the time of flight profile and prior knowledge (region of interest or ROI information in our paper) ❻. Then it enhances the direct path and estimates the location with a summation layer (a form of near-field AoA+ToF localization) ❼.

3 One-shot Wideband with Multisine Wave

This section explains why we select multisine wave as the wideband signal and how RF-CHORD acquires fine-grained tag responses in one shot. We review the primer of the backscatter signal model and its fundamental narrowband constraint. Then we present our design of constructing a wideband backscatter signal with the multisine waveform on Tx and slicing it for real-time parallel processing on Rx.

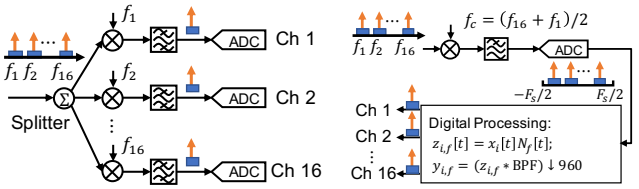
3.1 Backscatter Signal Model Primer

The basic backscatter operation in RFID systems is shown in Fig. 3a. A device emits a high-power single-tone excitation signal $s(t)$ to power the tag and act as a carrier. This carrier will be modulated by the baseband signal $B_{\text{tag}}(t)$ of the tag.



(a) Single-tone Backscatter. (b) Multisine Excitation Signals.

Figure 3: Model of Multisine Backscatter.



(a) Analog Channelization. (b) Digital Channelization.

Figure 4: Two channelization approaches.

The resulting (mixed) backscattered signal is:

$$r(t) = s(t) \cdot B_{\text{tag}}(t)$$

Note that the bandwidth of $r(t)$ is the summation of that of s(t) and $B_{\text{tag}}(t)$, and $B_{\text{tag}}(t)$ is typically a narrowband signal[2] for low power purpose according to the EPC Gen II standard. Therefore, the backscattered signal $r(t)$ will also be narrowband given that $s(t)$ is a single tone.

3.2 Backscattering with Wideband Multisine Wave

When applying a wideband signal $s(t)$, one can retrieve a wideband backscatter signal $r(t)$. Following this idea, RF-CHORD adopts a multisine signal as $s(t)$. The multisine signal is a combination of multiple single tones across wide band with the same amplitude $s(t) = \sum_i \sin(f_i t + \phi_i)$. The backscattered signal will be $r(t) = \sum_i B_{\text{tag}}(t) \cdot \sin(f_i t + \phi_i)$. RF-CHORD adopts 16 carriers with different frequencies across a 200 MHz band in the practical implementation. Fig. 3b shows the spectrum of multisine signal $s(t)$ with backscatter signal $r(t)$. Since the difference between each carrier frequency is much larger than the bandwidth of $B_{\text{tag}}(t)$, the received signal can be treated as multiple copies of $B_{\text{tag}}(t)$ modulated on different carrier f_i. Therefore, on Rx, $r(t)$ can be sliced to 16 individual narrowband channels without information loss, and then the channel information at each carrier frequency f_i can be extracted by using a well-explored RFID processing mechanism (*e.g.,* mixing and demodulating) in parallel. In a nutshell, we sample the wideband with multiple narrowband signals, enabling RF-CHORD to construct the wideband channel information within one shot.

3.3 Why Multisine Wave

The multisine waveform has two advantages. First, the multisine waveform is adaptive to conventional narrowband decoding and channel estimation because the signal in each channel is still narrowband. Extracting these narrowband signals can achieve excellent data rate compression (§3.4). Second, the multisine waveform is amenable to noise and interference reduction because of the low noise bandwidth and low chances of being interfered with, resulting in SINR enhancement, which improves the work range (§4.1). Compared with the two alternative well-known wideband waveform choices, frequency hopping [14] and OFDM signal [15], the multisine waveform is more efficient because it avoids the time overhead in switching between carriers introduced by the former one, and uses the same bandwidth as the (tag) modulation bandwidth, which is 250 kHz out of the full 200MHz bandwidth used by the latter one. In fact, this wideband but narrow sample signal can introduce 29 dB gain on the SINR compared to the full wideband signal (see §4.1), which means around 5× range under the same transmit power. Furthermore, since the multisine wave captures all the backscatter signals in the time domain, the whole packet of tags can be fully utilized for integration gain to improve the SINR (see §4.4).

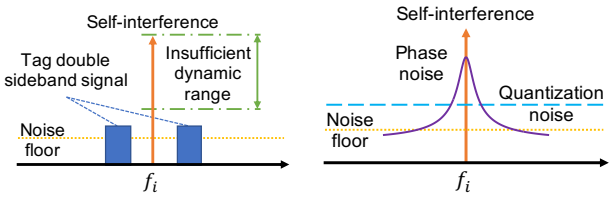
[2]We take 250 kHz as the bandwidth $B_{\text{tag}}(t)$ for the whole paper according to the standard [18].

3.4 Digital Wideband Channelization

RF-CHORD utilizes channelization, which enables one-shot capturing of wideband signals across multiple antennas and reduces the amount of data to be processed during real-time operation. Channelization is a process of extracting effective narrowband signals from a received signal. When a wideband tag signal is received, the aggregated bandwidth of 8 antennas will be 1.6 GHz, resulting in a total of 64 Gbps data (16-bit IQ sample, 1.25× Nyquist). It is challenging to process such massive data in real time. However, recall that with a multisine excitation signal, the effective tag signal is only located around the carrier frequencies, as shown in Fig. 3b. Therefore, the effective bandwidth of the system should be 8 × 16 × 250 kHz = 32 MHz, only 1/50 of the full 200 MHz bandwidth, so that channelization can compress the data validly without information loss.

There are two channelization schemes to extract these narrowband signals: analog channelization and digital channelization. As shown in Fig. 4a, the sniffer with analog channelization has multiple RF chains for the corresponding channels. Each RF chain uses one carrier frequency f_i as its local oscillator (LO) for down-conversion and a filter at the baseband to filter the signal from other channels out. Alternatively, digital channelization finishes all the aforementioned functions in the digital domain as shown in Fig. 4b.

RF-CHORD adopts digital channelization – the sniffer will generate and capture the whole multisine wave with one RF chain. On the Rx side, an ADC/DAC with a 245.76 MHz sampling rate captures all tag signals simultaneously. Further channel extraction can be achieved by digital down-conversion and digital filtering. Digital channelization offers two significant benefits over analog channelization: First, it has better scalability because it only needs one RF chain for each antenna, regardless of the number of channels (and sine



(a) Dynamic Range. (b) Unpredictable Noise.

Figure 5: Two issues caused by self-interference.

tones) are required, while in analog channelization, each channel needs an exclusive RF chain with bulk components (*e.g.,* mixer, PLL, and VCO). Second, it is precisely synchronized among different tones in the multisine wave, while analog channelization needs extensive engineering efforts to synchronize among a large amount of ADCs/DACs and LOs. Nevertheless, analog channelization still has it own advantages, including the convenience of extending or switching bandwidth by changing the carrier frequency and the lower requirements of ADC bandwidth. RF-CHORD also embraces these advantages through the high-speed ADC and low crest factor multisine waveform, which will be introduced in §4.

4 SINR Improvement for Long Range

This section first presents how RF-CHORD improves SINR under long work range by reducing the external noise and canceling self-interference. It next explains how RF-CHORD exploits the full tag packet to incorporate the integration gain, which is based on the multiple channel decoder with clock offset mitigation.

4.1 External Noise Suppression

To follow the FCC regulation, the signal strength of each frequency component in the multisine is -15 dBm, which is 51 dB lower than the 36 dBm excitation signal in the ISM band (see details in §A). With the low signal strength limitation but the long range requirement, we need to reduce the external noise and interference as much as possible.

RF-CHORD adopts the tag signal with reduced bandwidth for lower chances of in-band interference and lower noise. The relationship between thermal noise P_{noise} and signal bandwidth B at room temperature can be expressed as $P_{\text{noise}} = -174 + 10\log_{10}(B)$ [19]. As described in §3.4, the digital channelization at the receiver separates a combined 200 MHz wideband signal into multiple 250 kHz narrowband signals. This means that the thermal noise can be reduced from -91 dBm to -120 dBm (29 dB gain). Furthermore, the reduced bandwidth also reduces the probability of being interfered with by other devices working in the same band.

4.2 Self-interference Canceling

Besides the external interference from other devices, the self-interference caused by the natural full-duplex operation of our active sniffer will also limit the SINR. RF-CHORD's multisine waveform and low power transmission reduce the complexity of self-interference cancellation. As shown in Fig. 5a, the self-interference in one channel is just a single tone after channelization. A commercial tag uses double-sideband modulation with a subcarrier to differentiate the tag signal from the single-tone excitation signal. Therefore, RF-CHORD uses filters to cancel the self-interference caused by the single tone.

However, given the wideband signals are too weak (*i.e.,* -15 dBm), there remain two practical challenges. First, the dynamic range of the receiver may not be large enough to detect the tag signal. Second, any unpredictable noise, such as phase noise and circuit noise from Tx, will be transmitted along with the $s(t)$ and may bury the wideband signal. Then we'll go over how to deal with these issues.

Dynamic Range. Dynamic range is the ratio between the largest and smallest values that the received signal can assume. Specifically, the largest value is the self-interference, and the smallest signal is the targeted wideband tag signal. As shown in the Fig. 5a, even though the tag signal strength is higher than the noise floor and interference, it can still be buried if the dynamic range is not large enough. RF-CHORD meets the requirement of dynamic range by adopting the following strategies: First, it adopts a high-resolution ADC because the dynamic range of the receiver will be bottlenecked by the dynamic range of the ADC. The theoretical dynamic range of the receiver is 6.02 N + 1.76 dB [20], where N is the resolution of the ADC. Therefore, a fundamental way to solve the issue is to increase the resolution of the ADC. RF-CHORD adopts 16-bit ADC, which has the largest resolution in 2022 when satisfying the 200 MHz bandwidth requirement. Secondly, it adopts a carefully designed low crest factor multisine wave on the transmission side to relax the dynamic range requirement of the Rx. The intuition behind this is that since the dynamic range requirement on the ADC is more related to the peak amplitude of the self-interference signal instead of the average signal power, it can be relaxed by using a lower peak signal while remaining the average power. The crest factor is the peak amplitude divided by the RMS value of the waveform, and for a multisine signal, it has been well studied that the crest factor can be reduced by tuning the phases ϕ_i in the multisine signal. Following the methods mentioned in [21], the crest factor of the multisine waveform adopted by RF-CHORD can be reduced from 4 to 1.24 (or peak-to-average power ratio from 12 dB to 1.87 dB).

Unpredictable Noise. The unpredictable noise is caused by the response of self-interference in the circuit. As illustrated in Fig. 5b, the noise floor may be dominated by the phase noise, DAC quantization noise, *etc.* along with the self-interference. Fortunately, RF-CHORD does not require a dedicated cancellation circuit like [22] because the power of RF-CHORD's self-interference is much lower than that of a commercial RFID reader. Moreover, RF-CHORD utilizes Analog Devices ADRV9009 transceiver of 16-bit ADC [23] and HMC7044 VCXO-based clock tree [24], ensuring an optimal quantization and clock phase noise below the noise floor. Therefore,

the RF frontend of RF-CHORD's receiver is not saturated, and the noise will only go through the air instead of the feedback path of the receiver. The noise experienced by RF-CHORD is not dominated by unpredictable noise.

4.3 Full Packet Matching

RF-CHORD estimates each channel in parallel and then combines them into a wideband channel estimation. The standard channel estimation techniques for one channel can be expressed as follows:

$$h_i = \sum_t r(t)\hat{I}^*(t)$$

where $r(t)$ is received tag response and $\hat{I}(t)$ is a template. In most RFID systems, only the pilot signal part (RN16) is used for clock and phase estimation, and the main part of the tag signal (EPC ID) is left unused. RF-CHORD utilizes the full packet signal, including RN16 and EPC ID. The length of the signal will be extended from 0.31 ms to 2.31 ms when assuming the backscatter link frequency (BLE) of the tag is 250 kHz and the EPC ID length is 96 bits [25]. By doing the full packet matching, RF-CHORD can achieve $10\log_{10}\frac{2.31}{0.31} = 8.7$ dB integration gain.

We need to generate a noiseless template of the full packet for full packet channel estimation. However, unlike the predefined pilot signal, the template of the packet changes depending on the tag's EPC ID. Collecting EPC ID and timestamp from a commercial reader device in real-time is unsupported due to the interface limitation: i) the available interface from a commercial reader is usually done by using asynchronous communication, which hinders real-time processing; ii) the timing information is usually not reported by commercial readers. Therefore, RF-CHORD needs to decode the wideband signal into EPC ID independently.

4.4 Clock Offset Mitigation

Accurate decoding needs to mitigate the clock offset of the RFID tag signal. Specifically, the protocol tolerates up to $\pm$ 10% frequency offset and $\pm$ 2.5% frequency fluctuation during backscattering (refer to Tab. 6.9 of [18]). For example, say we read a tag that is 2.5% faster than nominal BLF. For a typical randomized uplink packet of 128 bits with a perfect match at the start of the frame, the received signal will be ahead of the template by one bit at the 32nd bit, and the remaining 96 bits thereafter contribute useless fluctuations to channel estimation, as figured out in Fig. 6. RF-CHORD needs to analyze the clock and estimate the offset parameters for mitigation, which can be described by:

$$\tau(t) = \text{Square}((f_{\text{BLF}} - \alpha_0 - \alpha(t))(t - t_0))$$

Where t_0 is the actual start of frame (SOF), α_0 is the initial clock frequency offset (CFO) from prescribed BLF, and $\alpha(t)$ is the fluctuation of the clock. Next, we introduce RF-CHORD's components which estimate these parameters.

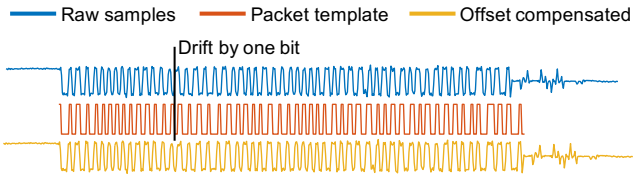


Figure 6: The waveforms of the tag signal with clock offset, the reference, and recovery signal from the offset.

Preamble Matching for t_0 and α_0. RF-CHORD first estimates the t_0 and α_0 by adopting a standard sliding window correlator with a known preamble $p(t)$. Specifically, we derive the initial estimation of $\hat{t_0}$ and $\hat{\alpha_0}$ by this correlation calculation, where the $x(t)$ is the received samples, $p_\alpha(t)$ is the reference template tuned to a clock frequency of $f_{\text{BLF}} - \alpha_0$:

$$\{\hat{t_0}, \hat{\alpha_0}\} = \underset{t_0,\alpha_0}{\text{argmax}} \left| \int_0^{T_p} p^*_{\alpha_0}(t)x(t+t_0)\text{d}t \right|$$

PLL to Track $\alpha(t)$ Variation. After eliminating α_0, the clock still has residual offset $\alpha(t)$, which comes from the tag clock fluctuation during the communication and may be significant in the long packet. Because the Miller code of RFID [18] is a self-clocked and modulated bandpass signal, RF-CHORD can extract the subcarrier of the line code to track the clock frequency offset accurately. RF-CHORD adopts a feedback-based digital Costas PLL [26] to track the clock continuously.

After compensating estimated clock $\tau(t)$, the clock offset is mitigated (the last waveform shown in Fig. 6). We can see that the signal is well synchronized with the template.

4.5 Decoding with Channel Diversity

After clock offset mitigation, we can decode the full packet, extract the correct template $\hat{I}(t)$, and assemble the decoder. Because the tag baseband signals on all channels are the same, RF-CHORD can apply nulling and beamforming algorithms to utilize the diversity across frequencies and antennas to make a joint decoder. RF-CHORD combines the signals from all channels into one *steered* single-channel signal – it first performs an adaptive maximum signal-to-noise ratio (MSNR) beamforming over the array of each frequency to null the major jammer in the spatial domain and then performs maximum-ratio combining (MRC) beamforming across the frequency domain to improve the SINR further. With this cleaned steered single channel, RF-CHORD exploits a Viterbi decoder to decode the EPC ID. It then applies the EPC ID to make accurate channel estimations on all the channels. A series of efforts introduced in this section, including suppressing external noise, canceling self-interference, matching full packet, mitigating clock offset, and decoding with diverse channels, guarantees RF-CHORD to extract wideband channel estimations at a long distance even with the ultra-low power emission signal.

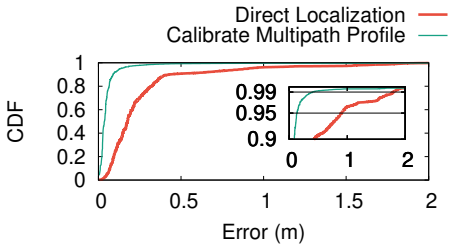


Figure 7: Eliminating the multipath effect reduces the 99th long-tail error.

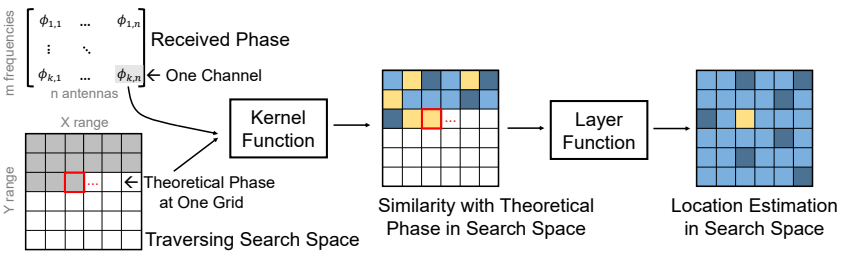


Figure 8: Kernel-Layer Framework.

5 Localization with Kernel-Layer Framework

In this section, we first conduct empirical experiments which show: i) multipath is the primary factor that confines the long-tail performance of the RFID localization system once the tag is successfully inventoried; ii) 200 MHz bandwidth is not sufficient to eliminate all the long-tail errors caused by multipath. To address these problems, we propose a kernel-layer framework for localizing RFID tags in the near field. It can suppress long tail errors from multipath by enhancing the direct path and incorporating prior knowledge from logistics.

5.1 Long-tail Errors Source Demystification

We conduct a validation experiment to confirm that multipath is the primary source of long-tail localization errors. In this experiment, we put five tags at a distance of 4 m from the reader. We use 16 carriers evenly spaced across 200 MHz bandwidth, 8 antennas, and a hologram-based localization algorithm (see details in §5.2). There is a metallic heater 1.5 m from the tag as the multipath source. Fig. 7 shows that the 99th localization error (red line) is 1.798 m, too large to ensure reliable usage in industry settings. The theoretical analysis explains this observation – the 200 MHz bandwidth is only able to differentiate paths that have a propagation distance difference larger than $c/(2B) \approx (3\times10^8 \text{ m/s})/(2\times200 \text{ MHz}) = 0.75$ m. Once the propagation distance of two paths is smaller than 0.75 m, which is common for many indoor deployments, 200 MHz is insufficient for differentiating one from the other.

Then we evaluate the performance without the multipath effect to check our results double. We keep the experiment setup, conduct RF measurement of a reference tag close to target tags and extract its phase offset from the groundtruth. Considering that the multipath profiles of nearby tags are similar, we subtract each tag's channel estimation with the offset from the reference tag. The 99th localization error of the same set of tags decreases to 0.400 m (green line in Fig. 7). It proves that multipath is the primary factor determining the long-tail performance of the RFID localization system, even with 200 MHz bandwidth.

5.2 Near-field Localization with Hologram Algorithm

Like most recent RFID localization systems, RF-CHORD locates a tag under the *near-field* condition, which differs from locating a distant target. Considering the Fraunhofer distance [27], a target is at near-field when its distance d from the antenna array meets:

$$d < \frac{2D^2}{\lambda}$$

where D is the aperture of the antenna array, and λ is the signal's wavelength. The wavelength of the 915 MHz signal is around 30 cm. When using an antenna array or SAR, the aperture can easily span to 1 m for adequate spatial resolution. $(2D^2)/(\lambda) = (2\times1 \text{ m}^2)/(0.3 \text{ m})$ = 6.7 m and $d < 6.7$ m under most circumstances. Therefore, the response from a tag does not form a plane wave when reaching different elements in the antenna array.

We propose to develop our localization algorithm on top of hologram-based localization framework, which essentially identifies the most likely location as the location estimation, independent of plane wave incidence conditions. In the basic hologram algorithm, the theoretical phase $\theta(g_{(i,j)}, A_k, f_l)$ of a tag at location $g_{(i,j)}$ received by an antenna A_k at frequency f_l can be written as:

$$\theta(g_{(i,j)},k,l) = \frac{2\pi f_l}{c}(d_{Tx-Tag}+d_{Tag-Rx}) \pmod{2\pi}$$

where d_{Tx-Tag} and d_{Tag-Rx} are the distance between the tag and the transmitter and receiver, respectively. For location $g_{(i,j)}$, its likelihood $P(g_{(i,j)})$ of being the tag's true location can be measured by the similarity between empirically received phase $\phi_{k,l}$ from lth carrier at kth antenna and the theoretically modeled phase $\theta(g_{(i,j)},k,l)$. The hologram algorithm makes the similarity comparison across multiple antennas and frequencies. $P(g_{(i,j)})$ can be written using the following equation:

$$P(g_{(i,j)}) = \left|\sum_{l=1}^{L}\sum_{k=1}^{K} e^{-j(\phi_{k,l}-\theta(g_{i,j},k,l))}\right| \quad (1)$$

Then we can estimation the location of the tag by choosing (i,j) with maximum P.

5.3 Kernel-layer Framework

Beyond the basic hologram algorithm [28], there are many hologram variants [7, 8, 29, 30]. We find that two key factors determine the performance of hologram-based localization algorithms, namely, kernel and layer:

Definition 1. Kernel is the function that measures the similarity between the received signal and the theoretical signal

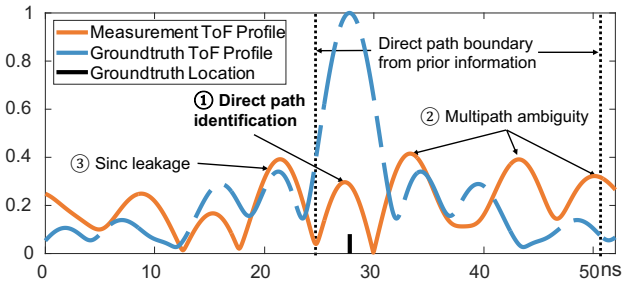


Figure 9: Direct Path Identification with ROI Information.

from one channel (*i.e.,* single carrier from a single antenna). For example, the $e^{j(\phi-\theta)}$ in Eqn. 1 is a kernel function that measures the phase similarity with an exponential function.

Definition 2. Layer is a function that determines how to combine kernels from multiple channels (*i.e.,* multiple carriers from multiple antennas) and obtain the location estimation. For example, the $\sum_{l=1}^{L}\sum_{k=1}^{K}$ in Eqn. 1 is a layer function.

We introduce a kernel-layer framework that tells us how kernel and layer affect the localization performance. Fig. 8 summarizes our kernel-layer framework, which describes the fundamentals of hologram-based algorithms. This framework can be used following these steps:

• *Model* calculates the theoretical channel information (*e.g.,* propagation phase) for each location.

• *Measurement* obtains the empirical channel information (*e.g.,* propagation phase and RSSI) by interrogating the tags.

• *Kernel* function profiles the similarity between the theoretical and empirical channel information.

• *Layer* function combines kernel function output from different antennas and frequencies.

• *Output* picks the location with the maximum likelihood as the estimated location.

Different kernels and layers can be combined into various near-field localization algorithms. See more examples in §B.

5.4 RF-CHORD's Kernel and Layer

We design our localization algorithm based the kernel-layer framework. When designing RF-CHORD's kernel and layer, we want to reduce the impact of multipath for low long-tail error, which can be achieved with the carefully designed kernel, layer, and prior information from the logistic scenario. RF-CHORD's kernel is similar to basic hologram algorithms:

$$\text{RF-CHORD's kernel: } e^{-j(\phi-\theta)}$$

RF-CHORD has 4 layer functions: ToF estimation layer, direct path identification layer, direct path enhancement layer, and summation layer. These layers work together to suppress the multipath and combat long-tail localization errors.

ToF Profile Layer. By using the wideband bandwidth captured, this layer computes the time-of-flight profile of the received signal. The computation follows Eqn. 2 where ϕ is the empirically measured phase value, f_l is the frequency, and τ is the propagation delay of each path.

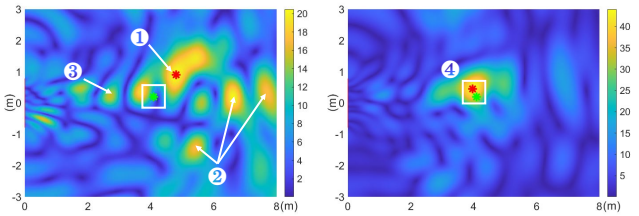
$$\text{ToF estimation layer: } S(\tau) = \sum_{l=0}^{L} e^{-j(\phi_l - 2\pi f_l \tau)} \quad (2)$$

Direct Path Identification Layer. It is still challenging to identify the direct path in the ToF profile layer in Fig. 9 because there are three interfering factors: ① If the difference is smaller than 0.75 m, we can only observe one mixed peak in the time-of-flight profile of the received signal. ② If the difference is larger than 0.75 m, there will be ambiguity from multipath at the locations farther from the groundtruth. ③ The sample on the frequency domain, which is a sinc function on the time domain, may leak its side lobe and form fake peaks at a nearer location than the groundtruth. To address these problems, RF-CHORD leverages a key observation: prior information. In practical logistic deployment, we can employ the size of the scanning area, the track of tags, *etc.* to help localization. RF-CHORD constructs a layer that leverages this prior information for direct path identification. Fig. 9 shows an example of this layer with scanning range [a,b] in meters as prior information, which is common in warehouse deployment. The corresponding algorithm is shown in Alg. 1. In this example, we first compute the bound of the theoretical propagation time in this range $\tau_a = a/(3\times10^8)$ and $\tau_b = b/(3\times10^8)$. The prior information, τ_a and τ_b, acts as a filter that eliminates any multipath with a propagation time smaller than τ_a or larger than τ_b, which helps us identify the right direct path (right peak) rather than nearer one from sinc leakage or farther one from multipath.

Algorithm 1 Direct path identification layer

Input: 1. ToF profile: $[S(\tau_1), S(\tau_2), ..., S(\tau_s)]_{1\times s}$
2. Prior info: scanning area in meters [a,b]
3. Peak threshold: p
Output: Direct path distance rough estimation $\tilde{d}_0$
1. $\tilde{d}_0$ = 0, $\tau_a = \frac{a}{3\times10^8}$, $\tau_b = \frac{b}{3\times10^8}$;
2. L = find τ_i closest to τ_a in [τ_1, τ_2, ..., τ_s], return index;
3. R = find τ_i closest to τ_b in $[\tau_1, \tau_2, ..., \tau_s]$, return index;
4. $S(\tau) \leftarrow S(\tau)$[L : R];
5. $path \leftarrow S(\tau)[0 : \text{end}-1] - S(\tau)[1 : \text{end}]$
for $i \leftarrow 1$ to $s-1$ **do**
  if $path[i] > 0$ & $path[i-1] < 0$ & $S(i) >$ p **then**
    $\tilde{d}_0 = \tau_i \times 3\times10^8$;
    break;
  end if
end for

Direct Path Enhancement Layer. RF-CHORD uses a across-frequency phase redress algorithm to further enhance the signal quality of the direct path signal. RF-CHORD first identifies potential multipath – if there are multiple peaks (identified by



(a) Basic Hologram. (b) RF-CHORD's Algorithm.

Figure 10: RF-CHORD can suppress sinc leakage, multipath ambiguity, and enhance direct path for finer resolution compared to basic hologram algorithms in Eqn. 1.

2D peak find algorithm [31]) in the basic hologram results, the location estimation is likely affected by the multipath effect. Instead of using the empirically measured phase ϕ, RF-CHORD combines the direct path signal from all frequencies and constructs an enhanced phase $\tilde{\phi}_l$. This process is done by the layer function of Eqn. 3. See §C for detailed mathematical derivation.

$$\text{Direct path enhancement layer: } \tilde{\phi}_l = \angle \sum_{i=1}^{L} e^{j\phi_i} e^{j\frac{2\pi}{c}(f_i - f_l)\tilde{d}_0} \tag{3}$$

Summation Layer. The last layer in RF-CHORD is the summation layer, which combines information from all L frequencies and K antennas and computes the likelihood of the tag position. For every location $g_{(i,j)}$, RF-CHORD computes the likelihood $P(g_{(i,j)})$ and choose the position with the highest likelihood as the estimated result.

$$\text{Summation layer: } P(g_{(i,j)}) = \left| \sum_{l=0}^{L} \sum_{k=0}^{K} e^{-j(\tilde{\phi_{l,k}} - \theta(g_{(i,j)},l,k))} \right| \tag{4}$$

Putting Everything Together. All above layers and kernel work together as our multipath suppression algorithm. Fig. 10 shows an visual example. The heatmaps are the location likelihood with the basic summation layer in Eqn. 1 (Fig. 10a) and with our direct path enhancement algorithm (Fig. 10b). The green cross is groundtruth and the red cross is location estimation. If we only use the simple summation layer, there are three factors disturbing the localization accuracy. RF-CHORD handles them with customized kernel-layer algorithm design. The peak of location estimation ① is the superimposed responses from all the paths within distance resolution nearer the direct path. RF-CHORD utilizes coherent summation layer with full 200 MHz bandwidth to increase distance resolution to 0.75 m. The paths with large distance differences from the direct path will generate ambiguity at farther arrival distances as multipath ambiguities ② or even at nearer distance as sinc leakage ③. By using prior information of work range (tags are in different check-in passage with different ranges) to clarify the direct path identification and using direct path enhancement to suppress multipath, we obtain the accurate location estimation ④.

6 Implementation

6.1 Active Sniffer

Antenna. We chose a recent variant [32] of the Foursquare patch antenna [33], which is metal-backed and of concentric dual-polarization, as our wideband Tx and Rx antennas for its advantages of small-size, low-cost, and high adaptability to surroundings. The original antenna design is for 1.7~2.7 GHz LTE and we scaled it with HFSS [34] to fit the UHF band 700~1100 MHz. We also attached each Rx antenna to a 915 MHz bandstop filter [35] to suppress the high-power ISM-band leakage from the commercial reader.

Array. We built the Rx array through a laser-cutting sheet of aluminum. The mounting holes and SMA clearances on the sheet define a 1×8 linear array with element spacing of 21 cm. We set a notable 31.5 cm gap in the middle for a 2:3 co-prime array configuration [36] to suppress the grating lobe. We hang two Txs 0.4 m lower than the receiver's horizontal array along its geometric bisection. The right one was wideband Tx and the left one was ISM-band Tx.

Baseband Processor. One of the key implementation challenges towards one-shot inventory is to convert the 31 Gbps I/Q samples from the A/D to the application processor. We developed high throughput baseband with 2 ADRV9009 [23,37] RF chips and an XCKU060 FPGA SoM [38,39] in charge of 4 receivers over 200 MHz bandwidth for PCIe streaming.

Application Processor. The host is equipped with a Core-i9 9900 CPU and an RTX 3090 GPU for real-time decoding and CSI acquisition. GPU was used to handle the template matching during the decoding with FFT convolution acceleration and parallelism. We used Process Explorer [40] to measure resource utilization and report the results in Tab. 2. The decoder is developed with C++/Eigen except that the most compute-intensive part, *i.e.,* the full packet matching algorithm, is implemented on GPU with CUFFT [41].

CPU (Utilization)	GPU (Utilization)	I/O Bandwidth	Memory
Core-i9 9900 (16.1%)	RTX 3090 (38.0%)	520.1 MBps	4.1 GB

Table 2: Hardware Resource Utilization.

6.2 RFID Tags

In order to ensure compatibility and low-cost, we used a commercial RFID IC Impinj Monza-M4A [42] and implemented a bandwidth extension technique [43] to redesign the metal inlay (antenna) on 80 × 80 mm single-sided PCB. The CAD of the RFID antenna is shown at the top left of Fig. 11 and its direction gain (similar to dipole antenna) is shown in Fig. 15a. It works on 700~1000 MHz, whose copper geometry can be transferred to flexible inlay for massive production.

7 Evaluation

7.1 Experimental Setup

Testing Environment. We evaluate RF-CHORD in an office with multiple reflectors (*e.g.,* metal furniture, low ceilings,

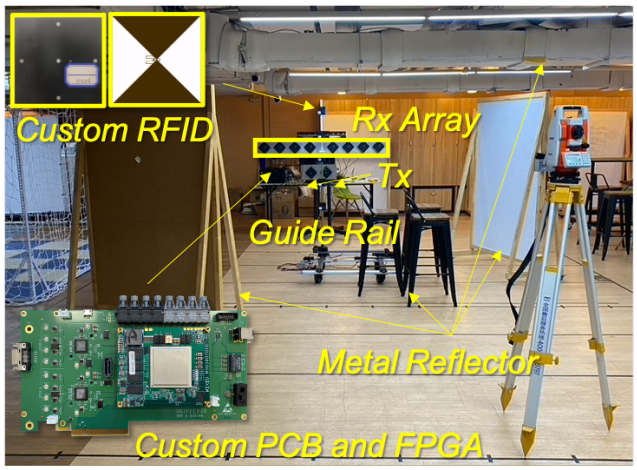


Figure 11: Experimental setup for evaluating performance. Five tags are mounted on the rail and ~20k tag responses are collected in 384 locations.

and walls). The evaluation range is the area of 6 × 3.2 m ahead of the antenna. We divide the evaluation space into 20 cm grids and use guide rails to move the tags. All the tags are facing the array. The dataset containing about 20k wideband RFID channel information measurements at 384 locations is open-sourced at [44]. The setup is shown in Fig. 11.

Location Groundtruth. Groundtruth is measured from a total station theodolite (TST) [45] with a 2 mm/2″ accuracy.

Frequency Band Configuration of Active Sniffer. We use the band of 787~987 MHz and avoid selecting carriers in ISM band 902~928 MHz. The carriers are almost evenly selected with spacing of 11.1 MHz[3]. The spectrum analyzer shows the inter-modulation distortion of carriers is very little.

ISM-band Reader. We use an Impinj R700 [46] as the ISM-band reader, which is configured on "Radio Mode 142" (Miller-4 coding and BLF of 256 kHz) and a single linear-polarized antenna aligned with the wideband Tx. We empirically pick this mode since it balances throughput and range. Other coding methods and BLF can also be adopted with few modifications to our system.

7.2 Throughput in One-shot Localization

Fig. 12 shows RF-CHORD's throughput at different distance. RF-CHORD can read and localize ~180 tags per second (97% of the tags read by an Impinj reader) at up to 6 m. RF-CHORD is 1000× faster compared to previous sniffer-based wideband systems with frequency-hopping. For instance, RFind [14] needs 6.4 seconds to localize one tag. We also evaluate RF-CHORD's throughput across emission power. Fig. 13 shows that RF-CHORD's throughput decreases when we reduce its emission peak power from -15 dBm to -35 dBm. It works fine with an emission power above -25 dBm.

[3]The frequency set of carriers is {787.1, 798.2, 809.3, 820.4, 831.5, 842.6, 853.7, 864.8, 875.9, 887.0, 898.1, 942.5, 953.6, 964.7, 975.8, 986.9 MHz}.

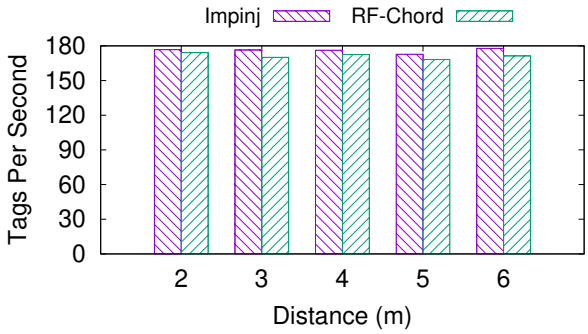


Figure 12: Throughput across distances. RF-CHORD can localize around 180 tags/s with -15 dBm emission power.

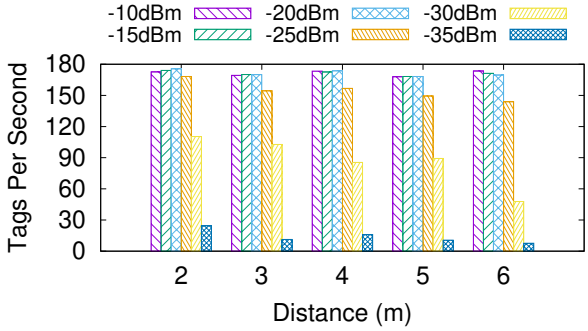


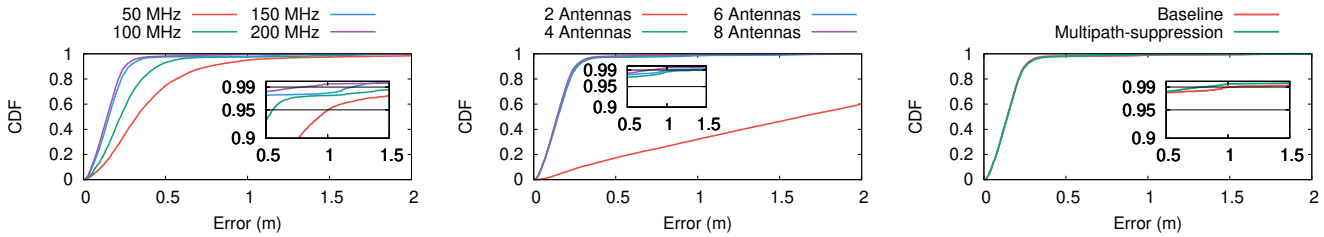
Figure 13: Throughput across distances with different emission power. The performance of RF-CHORD is stable with above -25 dBm emission power.

7.3 Localization Performance

RF-CHORD utilizes large bandwidth, multiple antennas, and the multipath-suppression algorithm to realize one-shot and high-reliability localization. We conduct microbenchmarks to evaluate how physical resources (frequency and spatial domain), algorithms, and orientation influence the localization.

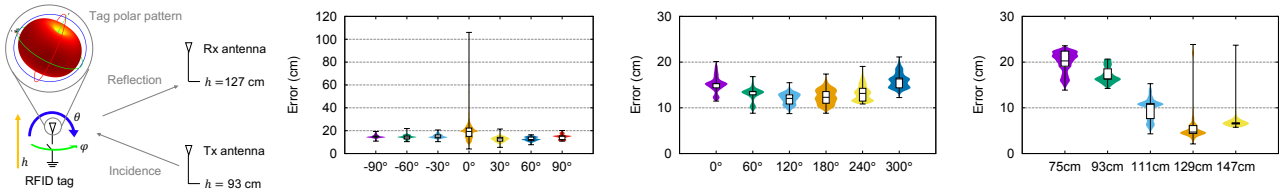
Bandwidth. We evaluate the localization performance with 8 antennas and different bandwidths. Fig. 14a shows 99th localization errors are 2.398 m, 1.646 m, 1.203 m and 0.786 m with 50 MHz, 100 MHz, 150 MHz and 200 MHz bandwidths. The median errors are 0.325 m, 0.227 m, 0.155 m, and 0.144 m, separately. The results show increasing bandwidth, thus increasing the time resolution, can not only improve the median performance but also reduce the long-tail error. Even when the median performance is close to the upper limit (150 MHz v.s. 200 MHz), the long-tail errors can still be reduced by increasing bandwidth.

Number of Antennas. We evaluate RF-CHORD's localization performance with 200 MHz bandwidth and different numbers of antennas (thus different array apertures). Fig. 14b shows RF-CHORD's 99th localization errors are 4.513 m, 1.467 m, 1.081 m and 0.786 m when 2, 4, 6 and 8 antennas are used. The performance of the 4, 6, and 8 antennas is very similar on median errors (about 0.14 m). However, their long-tail errors are significantly different. The results



(a) Bandwidth. (b) Number of Antennas. (c) Algorithm.

Figure 14: RF-CHORD's localization errors with different bandwidths, antenna numbers, and algorithms.



(a) Orientation Setup. (b) Pitch/Roll Angle (θ). (c) Yaw Angle (ϕ). (d) Height (h).

Figure 15: Microbenchmarks with different tag orientations and heights related to the antennas.

show increasing the number of antennas (from 2 to 8) can always improve long-tail performance. Increasing the number of antennas/apertures strengthens the system's immunity to interference in specific directions and improves the angular resolution for localization.

Algorithms. We take basic hologram (Eqn. 1) as the baseline algorithm and evaluate our multipath-suppression algorithm with 8 antennas and 200 MHz bandwidth. Fig. 14c shows that 99th localization errors of baseline and RF-CHORD are 1.018 m and 0.786 m respectively. The median errors of baseline and RF-CHORD are 0.143 m and 0.144 m, respectively. The algorithm effort can improve long-tail performance by handling more corner cases, but hard to improve median performance. Physical resources (*i.e.,* the bandwidth and the antenna array aperture) fundamentally limit the algorithm's performance, and the long-tail improvement from the algorithm is primarily attributed to the introduction of prior information – it provides an appropriate carrier for making use of prior information.

Orientation. In practice, the orientation of tags will influence the link angle and polarization, thus introducing SINR and phase changes. We evaluate how orientation influences the localization error. We set the target tag at a 1-m fixed distance to the antenna array to eliminate the influence of the multipath effect. Then, we change the pitch angle θ (as same as the roll angle due to the symmetry), yaw angle ϕ, and height of the tags as shown in Fig. 15a for orientation microbenchmark:

• *Pitch/Roll Angle.* In Fig. 15b, we keep $\phi = 0°$ and $h = 111$ cm (at the center between Tx and Rx). The worst performance occurs when the pole of the antenna points to the rx, which rarely happens in practical deployments (to be discussed in §8.1). It is difficult to read tags due to the low SINR, and even if successful, the long-tail error will be more than 1 m.

• *Yaw Angle.* In Fig. 15c, we keep $\theta = 0°$, $h = 111$ cm and change ϕ from $0°$ to $300°$. The errors at different yaw angles are similar because the directional gain across ϕ is symmetrical. The results show that the yaw angle does not affect long-tail localization error (bounded within 30 cm).

• *Height.* In Fig. 15d, we keep $\theta = 0°, \phi = 0°$ and move the tag from 75 cm to 147 cm. The long-tail errors do not change much across different heights, which shows that height is not the key factor affecting long-tail errors.

8 Practical Deployment

8.1 Deployment Constraints

We summarize the practical factors that influence SINR in Fig. 16 and introduce the constraints in real-world logistic scenarios. We also explain how we avoid or utilize them for high-reliability localization.

Orientation. The localization error may be significant if the pitch angle of the tag is closed to $90°$ according to §7.3. In the deployment shown as Fig. 17, the orientation of tags may not be uniform but unlikely to be completely disordered. All the tags are attached to the sides of boxes or crates and then stacked on the pallet. The chaos of stacking and the movement of the pallet may cause yaw angle (ϕ) change but not cause much pitch/roll angle (θ) change, which only introduces negligible localization errors according to Fig. 15c.

Polarization. We set the tags and sniffer antennas all vertically polarized, so horizontal tags can not be read. Similar to orientation, no tag will be misplaced in our scene because the pallet stack constraints the crate direction.

NLOS and Tag Coupling. We also stipulate that all the tags should be in the line of sight from one side dock door, which means stacking at most two-column crates on the pallet. It is because the performance of UHF RFID will decrease rapidly with nearby water [47]. This rule excludes severe NLOS occlusion/reflection and severe tag coupling. Most of the pallets

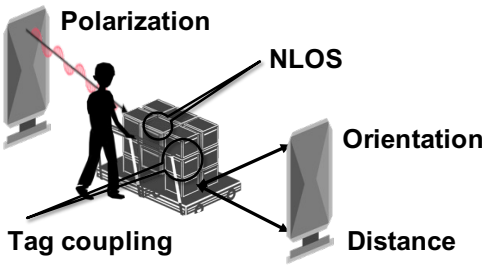


Figure 16: The Factors Affecting Signal Quality.

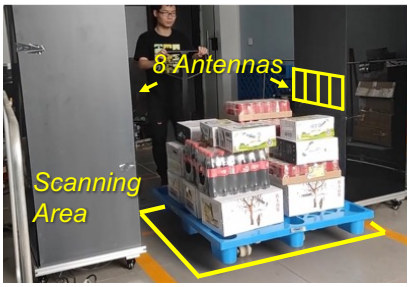


Figure 17: Warehouse Dock Door.

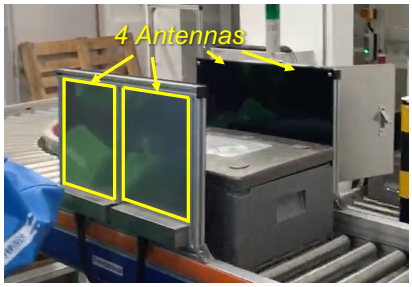


Figure 18: Food Delivery Store.

	Miss Reading Rate	Cross Reading Rate
Warehouse	0	0.0025%
Food delivery	0	0.0154%

Table 3: Reliability performance in practical deployments.

in our scenes naturally meet this requirement, and in rare cases, we need to waste some space.

8.2 Real World Deployment

We deployed the full-fledged RF-CHORD (*i.e.,* 200 MHz bandwidth and 8 antennas) in the warehouse dock door and lightweight RF-CHORD (*i.e.,* 200 MHz bandwidth and 4 antennas) in the fresh food delivery store according to cost and scene conditions for operational evaluation.

Warehouse Deployment. We deploy RF-CHORD in a warehouse to understand its performance in logistic check in and out. RF-CHORD is installed in the dock door of the warehouse as shown in Fig. 17. This warehouse's goal is to distribute a large amount of food and daily necessities supplied by the upstream warehouse to city delivery stations. The crates are various and packaged without unified standards. Ideally, RF-CHORD should report all the tags inside of the scanning area and not report any tags outside of the scanning area. Our previous deployment experiments in the same scenario show that commercial read-or-not solution Impinj xSpan [16] has ~6% miss-reading rate and ~2% cross-reading rate in the similar scene. We attached over 10,000 tags to various items, mainly plastic crates, also including water bottles, cans, milk boxes, rice, *etc.* The scanning area is 2 $\times$ 1 m between the two poles of the dock door (as ROI) and the user walks through the aisle with about 50~100 tags on a trailer in 1 to 4 seconds. According to Tab. 3, RF-CHORD is able to identify tags inside of scanning area with a 100% accuracy (perfectly no miss reading) and 0.0025% cross reading. Therefore, RF-CHORD can provide sufficient localization accuracy in the warehouse deployment, which significantly outperforms the state-of-the-art commercial solutions.

Fresh Food Deliver Store Deployment. As shown in Fig. 18, we also deploy lightweight RF-CHORD in a fresh food delivery store where fresh food is packaged into a container and transported via a moving belt. Once the RFID tag on the container is scanned, the delivery personnel will be allocated to pick it up. RF-CHORD needs to ensure all the containers on the moving belt are scanned and do not scan any tag outside of the moving belt. Tab. 3 shows that the miss-reading rate of RF-CHORD is 0%, and cross-reading rate is 0.0154%. Therefore, RF-CHORD can achieve sufficient accuracy in the fresh food delivery store deployment.

9 Discussion

Polarization Mismatch. In our scenarios, the work pipeline guarantees the polarization match. However, in more general scenarios, the polarization may be mismatched when the orientation of tags is disordered. The conventional solution is to use circular polarization antenna [48] or dual-polarization switching [16]. RF-CHORD can be adapted to them conveniently because its wideband four point antenna is inherently dual-polarized. We can plug a polarization switch into each sniffer antenna, which acts synchronously and does not influence throughput and range performance.

Blind Spots. RF-CHORD is free of cross reading, and therefore it can use high transmission power and sensitivity ISM-band reader for achieving nearly zero miss-reading rate. However, miss reading still threatens reliability in certain complex environments. It can be mitigated by switching between antennas or beam patterns [3, 49]. As our Tx is synthesizing multiple tones, it is feasible to add a Tx beamforming array for blind spot suppression.

Integration with Robots. In recent years, logistics robots (*e.g.,* automated guided vehicle (AGV) [50], automated storage and retrieval systems (ASRS) [51], and autonomous mobile robots (AMR) [52]) have been developed rapidly to reduce the movements and operations of sorters and improve efficiency. These robots still need to cooperate with a label identification system (*e.g.,* barcode or QR code). RF-Chord has the potential to replace such system and cooperate with logistics robots to achieve more efficient automation.

Cost. The ultimate goal of deploying RFID is to reduce manual labor and error while improving efficiency, which requires careful cost accounting. We emphasize that although baseband chips and RF circuits will increase the cost of readers to thousands of dollars, the main cost of RFID-based logistics

still comes from RFID tags. Considering a medium warehouse with 10k packages delivered every day, the annual cost of tags is approximately \$0.1 $\times$ 10,000 $\times$ 365 = \$365,000. Our strategy is not modifying the tag chip because most of the manufacturing cost comes from the chip and the assembly process [53]. Therefore, the wideband tag we designed maintains almost the exact cost as current commercial tags when in massive manufacturing.

10 Related Work

Narrowband Localization. There are three main localization approaches to boost accuracy even with the limited time resolution of narrow ISM bandwidth: The first approach is to improve spatial resolution by SAR. Tagoram [7] uses the motion of tags to build multiple virtual antennas, while Mobitagbot [8] exploits antenna motion. The hologram algorithms in these two systems inspired the kernel-layer framework in our paper. Other hologram algorithm variants [29, 30, 54] can also be viewed as different combinations of kernels and layers. However, the assumption of free antennas or tags mobility and lengthy startup time for tracking do not fit the logistic network. The second approach is to acquire prior information by reference tag. PinIt [6] exploits a dense grid of reference tags and determines the nearest reference tag for NLOS localization by dynamic time wrapping. However, reference tags share time slots, which influences the throughput and scalability. The third approach is to increase the number of links by tag array. Attaching more tags to the target can increase the number of links and improve localization performance. Tagyro [55], and RF-Dial [56] utilize the phase difference of the tag array to solve orientation ambiguity and improve localization performance. Trio [57] models the equivalent circuits of coupled tag and uses the tag interference for refined localization. Liu et al. [58] uses spatial-temporal phase profiling for relative RFID localization. These tag array based localization approaches are accurate but may be error-prone in a complex environment. Unlike these proposals, RF-CHORD is a sniffer-based wideband localization system that improves time resolution for fundamental performance enhancement.

Wideband Localization. Wideband RFID localization has been proposed to overcome the time resolution limitation. RFind [14] uses a low-power sniffer antenna by frequency hopping to collect the narrow sample channel state information across 220 MHz. Turbotrack [15] develops an OFDM-based one-shot wideband channel estimation approach and a Bayesian space-time super-resolution algorithm to achieve fine-grained localization. However, these systems need multiple shots in the channel estimation or the algorithm to converge for fine location estimation, thus very slow startup for localization or tracking. Modifying tags to work on other frequencies (*e.g.,* Wi-Fi [59], millimeter-wave [11], UWB [12, 13]) or cross-frequency based approaches (*e.g.,* communicate with Wi-Fi [10], communicate at 1.4~2.4 GHz [60]) are also expected as the solutions for both finer localization and higher throughput, but their tags are not ready for massive manufacturing at low cost due to the complicated RF frontend and control circuits. Inspired by these works, RF-CHORD develops a multisine waveform to realize one-shot localization without modifying the commercial tag chips, resulting in high accuracy with no throughput loss or cost increase.

RFID Reader. Commercial RFID readers [46, 61, 62] have heavily optimized RF analog frontend, decoder, and protocol stack but do not support real-time tag critical information (*i.e.,* EPC ID, timestamp) retrieval. There are a series of open-source RFID reader systems. Buettner et al. implemented EPC Gen II downlink stack [63] and the full functional reader [64], respectively. Kimionis et al. implemented a GNU radio-based reader, which supported OOK and noncoherent FSK [65]. However, their energy and edge detection algorithms are too simple to decode applicable code (*e.g.,* miller-4 coding). A recent reader designed by Kragas et al. [66] is featured by coherent detection and initial duration deviation search but only supports simple FM0 encoding. There are other research projects featured by multisine waveform [67], parallel sensing support [68], and active transmit leakage cancellation [69]. However, they only focus on specific optimization and do not provide source code. In a nutshell, no out-of-box reader design meets our requirements of high throughput and low decoding threshold, so we develop a wideband reader with a customized RF frontend and decoder while reusing the MAC layer of the commercial reader for slot arrangement and collision handling. It supports our wideband localization with high efficiency, sensitivity, and compatibility.

11 Conclusion

We illustrate the three key requirements in reliability, throughput, and range to meet the industry-grade standard of the logistic network, and present RF-CHORD, the first RFID system that considers all these factors from wideband signal and baseband processing to localization algorithm framework development. We believe our real-world empirical results demonstrate that RF-CHORD paves the way for the practical hardware-software methodological solution of RFID localization-based logistic network and makes an important step towards large-scale operational deployment.

Acknowledgments

We are grateful to the reviewers for their constructive critique, and our shepherd, Vikram Iyer in particular, for his valuable comments, all of which have helped us greatly improve this paper. We also thank Xieyang Xu and Weicheng Wang for providing an early implementation version of the work. We are grateful to Yunfei Ma for the thoughtful suggestions based on the early version of the work. This work is supported in part by National Key Research and Development Plan, China (Grant No. 2020YFB1710900), National Natural Science Foundation of China (Grant No. 62022005, 62272010, and 62061146001) and Alibaba Innovative Research. Chenren Xu and Shunmin Zhu are the corresponding authors.

References

[1] Global parcel volumes expected to double by 2026 on e-commerce boom. https://rogistics.net/global-parcel-volumes-on-course-to-double-by-2026/.

[2] Inside an amazon robotic sortation center: How automation is changing the 'middle mile'. https://www.geekwire.com/2022/inside-an-amazon-robotic-sortation-center-how-automation-is-changing-the-middle-mile/.

[3] Carlos Bocanegra, Mohammad A Khojastepour, Mustafa Y Arslan, Eugene Chai, Sampath Rangarajan, and Kaushik R Chowdhury. Rfgo: a seamless self-checkout system for apparel stores using rfid. In *ACM MobiCom*, 2020.

[4] Renjie Zhao, Purui Wang, Yunfei Ma, Pengyu Zhang, Hongqiang Harry Liu, Xianshang Lin, Xinyu Zhang, Chenren Xu, and Ming Zhang. Nfc+ breaking nfc networking limits through resonance engineering. In *ACM SIGCOMM*, 2020.

[5] Gang Li, Daniel Arnitz, Randolf Ebelt, Ulrich Muehlmann, Klaus Witrisal, and Martin Vossiek. Bandwidth dependence of cw ranging to uhf rfid tags in severe multipath environments. In *IEEE RFID*, 2011.

[6] Jue Wang and Dina Katabi. Dude, where's my card? rfid positioning that works with multipath and non-line of sight. In *ACM SIGCOMM*, 2013.

[7] Lei Yang, Yekui Chen, Xiang-Yang Li, Chaowei Xiao, Mo Li, and Yunhao Liu. Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices. In *ACM MobiCom*, 2014.

[8] Longfei Shangguan and Kyle Jamieson. The design and implementation of a mobile rfid tag sorting robot. In *ACM MobiSys*, 2016.

[9] Yunfei Ma, Xiaonan Hui, and Edwin C Kan. 3d real-time indoor localization via broadband nonlinear backscatter in passive devices with centimeter precision. In *ACM MobiCom*, 2016.

[10] Zhenlin An, Qiongzheng Lin, and Lei Yang. Cross-frequency communication: Near-field identification of uhf rfids with wifi! In *ACM MobiCom*, 2018.

[11] Ajibayo O Adeyeye, Jimmy Hester, and Manos M Tentzeris. Miniaturized millimeter wave rfid tag for spatial identification and localization in internet of things applications. In *IEEE EuMC*, 2019.

[12] Daniel Arnitz, Klaus Witrisal, and Ulrich Muehlmann. Multifrequency continuous-wave radar approach to ranging in passive uhf rfid. *IEEE transactions on microwave theory and techniques*, 57(5), 2009.

[13] Nicolo Decarli, Francesco Guidi, and Davide Dardari. Passive uwb rfid for tag localization: Architectures and design. *IEEE Sensors Journal*, 16(5), 2015.

[14] Yunfei Ma, Nicholas Selby, and Fadel Adib. Minding the billions: Ultra-wideband localization for deployed rfid tags. In *ACM MobiCom*, 2017.

[15] Zhihong Luo, Qiping Zhang, Yunfei Ma, Manish Singh, and Fadel Adib. 3d backscatter localization for fine-grained robotics. In *USENIX NSDI*, 2019.

[16] Impinj dual-polarized xspan rfid reader. https://support.impinj.com/hc/article_attachments/360002045159/xSpan_Overview_Datasheet_including_Software_Tools_Accessories_and_Specifications_20190405.pdf.

[17] Jue Wang, Deepak Vasisht, and Dina Katabi. Rf-idraw: virtual touch screen in the air using rf signals. In *ACM SIGCOMM*, 2014.

[18] Epc(tm) rfid class-1 gen-2 protocol. https://www.gs1.org/sites/default/files/docs/epc/uhfc1g2_1_2_0-standard-20080511.pdf.

[19] J. R. Pierce. Physical sources of noise. *Proceedings of the IRE*, 44(5), 1956.

[20] Quantization noise: An expanded derivation of the equation, snr = 6.02 n + 1.76 db. https://www.analog.com/media/en/training-seminars/tutorials/MT-229.pdf.

[21] Yuxiang Yang, Fu Zhang, Kun Tao, Benjamin Sanchez, He Wen, and Zhaosheng Teng. An improved crest factor minimization algorithm to synthesize multisines with arbitrary spectrum. *Physiological Measurement*, 36(5), 2015.

[22] Developing a uhf rfid reader rf front end with an analog devices solution. https://www.analog.com/en/technical-articles/developing-a-uhf-rfid-reader-rf-front-end-with-an-analog-devices-solution.html.

[23] Adrv9009. https://www.analog.com/en/products/adrv9009.html.

[24] Hmc7044. https://www.analog.com/en/products/hmc7044.html.

[25] Dogbone monza r6. https://rfid.averydennison.com/content/dam/rfid/en/products/rfid-products/data-sheets/datasheet-Dogbone-Monza-R6.pdf.

[26] John G. Proakis and Masoud Salehi. *Digital communications*. McGraw-Hill., 2008.

[27] Krishnasamy T Selvan and Ramakrishna Janaswamy. Fraunhofer and fresnel distances: Unified derivation for aperture antennas. *IEEE Antennas and Propagation Magazine*, 59(4), 2017.

[28] Robert Miesen, Fabian Kirsch, and Martin Vossiek. Holographic localization of passive uhf rfid transponders. In *IEEE RFID*, 2011.

[29] Huatao Xu, Dong Wang, Run Zhao, and Qian Zhang. Faho: deep learning enhanced holographic localization for rfid tags. In *ACM SenSys*, 2019.

[30] Huatao Xu, Dong Wang, Run Zhao, and Qian Zhang. Adarf: Adaptive rfid-based indoor localization using deep learning enhanced holography. *ACM IMWUT*, 3(3), 2019.

[31] Fast 2d peak finder. https://www.mathworks.com/matlabcentral/fileexchange/37388-fast-2d-peak-finder.

[32] Dong-Ze Zheng and Qing-Xin Chu. A wideband dual-polarized antenna with two independently controllable resonant modes and its array for base-station applications. *IEEE Antennas and Wireless Propagation Letters*, 16, 2017.

[33] Seong-Youp Suh, WL Stutzman, and WA Davis. Low-profile, dual-polarized broadband antennas. In *IEEE Antennas and Propagation Society International Symposium*, volume 2, 2003.

[34] Ansys hfss. https://www.ansys.com/products/electronics/ansys-hfss.

[35] 902-928 Cavity Band Rejection Filter WT-A3678-R10. https://www.wtmicrowave.com/en/product/WT-A3678-R10.html.

[36] Zhao Tan, Yonina C Eldar, and Arye Nehorai. Direction of arrival estimation using co-prime arrays: A super resolution viewpoint. *IEEE Transactions on Signal Processing*, 62(21), 2014.

[37] David J McLaurin, Kevin G Gard, Richard P Schubert, Manish J Manglani, Haiyang Zhu, David Alldred, Zhao Li, Steven R Bal, Jianxun Fan, Oliver E Gysel, et al. A highly reconfigurable 65nm cmos rf-to-bits transceiver for full-band multicarrier tdd/fdd 2g/3g/4g/5g macro basestations. In *IEEE ISSCC*, 2018.

[38] Xilinx ultrascale series fpga. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-fpga-product-selection-guide.pdf.

[39] Third party xcku060 som (in chinese). https://detail.tmall.com/item.htm?id=654943824333.

[40] Process-explorer. https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer.

[41] Cufft library. https://docs.nvidia.com/cuda/cufft/index.html.

[42] Monza 4 datasheet. https://support.impinj.com/hc/en-us/articles/202756908-Monza-4-Datasheet.

[43] Daniel D Deavours. Analysis and design of wideband passive uhf rfid tags using a circuit model. In *IEEE International Conference on RFID*, 2009.

[44] Towards deployable rfid localization system for logistics network. https://soar.group/projects/rfid/rfchord/.

[45] Total station instrument tutorial. https://www.aps.anl.gov/files/APS-Uploads/DET/Detector-Pool/Beamline-Components/Lecia_Optical_Level/Surveying_en.pdf.

[46] Impinj r700 rain rfid reader for enterprise-grade iot solutions. https://www.impinj.com/products/readers/impinj-r700.

[47] Supreetha Rao Aroor and Daniel D Deavours. Evaluation of the state of passive uhf rfid: An experimental approach. *IEEE Systems Journal*, 1(2), 2007.

[48] Impinj Inc. Impinj far-field rfid antenna. https://support.impinj.com/hc/article_attachments/360000841520/ANT-DS-S9028PCxx_Impinj1218.pdf.

[49] Jingxian Wang, Junbo Zhang, Rajarshi Saha, Haojian Jin, and Swarun Kumar. Pushing the range limits of commercial passive rfids. In *USENIX NSDI*, 2019.

[50] Automated guided vehicle. https://en.wikipedia.org/wiki/Automated_guided_vehicle.

[51] Maximize warehouse storage with as/rs. https://www.bastiansolutions.com/solutions/technology/asrs/.

[52] Autonomous mobile robot technology and use cases. https://www.intel.com/content/www/us/en/robotics/autonomous-mobile-robots/overview.html.

[53] Gitanjali Swamy. Manufacturing cost simulations for low cost rfid. *Available at SSRN 3690073*, 2020.

[54] Qingyun Zhang, Leixian Shen, Jiewen Shao, and Fu Xiao. Rf-track: Real-time tracking of rfid tags with stationary antennas. In *ACM TURC*, 2020.

[55] Teng Wei and Xinyu Zhang. Gyro in the air: tracking 3d orientation of batteryless internet-of-things. In *ACM MobiCom*, 2016.

[56] Yanling Bu, Lei Xie, Yinyin Gong, Chuyu Wang, Lei Yang, Jia Liu, and Sanglu Lu. Rf-dial: An rfid-based 2d human-computer interaction via tag array. In *IEEE INFOCOM*, 2018.

[57] Han Ding, Jinsong Han, Chen Qian, Fu Xiao, Ge Wang, Nan Yang, Wei Xi, and Jian Xiao. Trio: Utilizing tag interference for refined localization of passive rfid. In *IEEE INFOCOM*, 2018.

[58] Longfei Shangguan, Zheng Yang, Alex X Liu, Zimu Zhou, and Yunhao Liu. Relative localization of rfid tags using spatial-temporal phase profiling. In *USENIX NSDI*, 2015.

[59] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R Smith, and David Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *ACM SIGCOMM*, 2014.

[60] Yunfei Ma and Edwin Chihchuan Kan. Accurate indoor ranging by broadband harmonic generation in passive nltl backscatter tags. *IEEE transactions on microwave theory and techniques*, 62(5), 2014.

[61] Impinj speedway rain rfid readers for flexible solution development. https://www.impinj.com/products/readers/impinj-speedway.

[62] Alien alr-9900+. https://www.alientechnology.com/products/files-2/alr-9900/.

[63] Michael Buettner and David Wetherall. An empirical study of uhf rfid performance. In *ACM MobiCom*, 2008.

[64] Michael Buettner and David Wetherall. A software radio-based uhf rfid reader for phy/mac experimentation. In *IEEE RFID*, 2011.

[65] John Kimionis, Aggelos Bletsas, and John N Sahalos. Design and implementation of rfid systems with software defined radio. In *IEEE EUCAP*, 2012.

[66] Nikos Kargas, Fanis Mavromatis, and Aggelos Bletsas. Fully-coherent reader with commodity sdr for gen2 fm0 and computational rfid. *IEEE Wireless Communications Letters*, 4(6), 2015.

[67] Alírio J Soares Boaventura and Nuno Borges Carvalho. The design of a high-performance multisine rfid reader. *IEEE Transactions on Microwave Theory and Techniques*, 65(9), 2017.

[68] Yanwen Wang, Jiannong Cao, and Yuanqing Zheng. Toward a low-cost software-defined uhf rfid system for distributed parallel sensing. *IEEE Internet of Things Journal*, 8(17), 2021.

[69] Edward A Keehr. A low-cost software-defined uhf rfid reader with active transmit leakage cancellation. In *IEEE RFID*, 2018.

[70] Understanding the fcc part 15 regulations for low power, non-licensed transmitters. https://transition.fcc.gov/oet/info/documents/bulletins/oet63/oet63rev.pdf.

[71] 15.231 - periodic operation in the band 40.66-40.70 mhz and above 70 mhz. https://www.law.cornell.edu/cfr/text/47/15.231.

[72] Section 15.231, operating on multiple carrier frequencies. https://apps.fcc.gov/oetcf/kdb/forms/FTSSearchResultPage.cfm?id=41685&switch=P.

A FCC Compliance

RF-CHORD adopts a 200 MHz bandwidth in the UHF band, much wider than the 902~928 MHz ISM band. We need to reduce the power of the signal emitted in the licensed band to follow the FCC regulation [70]. Similar operations exist in other systems, such as RFind [14]. RFind adopts a duty-cycled single-tone signal with a peak power of -3 dBm and average power of -13.3 dBm. However, due to the throughput requirement of the localization, RF-CHORD's sniffer should always be ready to localize a tag, which means duty cycling is unacceptable. Therefore, RF-CHORD adopts a hard limit of -15 dBm per tone and can be even lower with similar performance. One may concern that the multiple carrier operation will not be the same as RFind [14] since the total bandwidth is larger than the 0.25% bandwidth limitation in FCC 15.231 (c) [71]. However, RF-CHORD can adopt the alternative method mentioned in [72], which calculates the total bandwidth by summing the individual occupied bandwidths of each carrier frequency. Since we did not apply any modulation to the carriers, the sum of respective bandwidths will be extremely small, which can comply with the FCC regulation. Other modulated waveforms (*e.g.,* OFDM) cannot follow this alternative method and may potentially violate the regulation.

B Kernel-layer Combinations for Different Localization Algorithms

Kernel-Layer near-field localization framework supports various localization algorithms because of the flexibility of measuring the similarity between receiving signal and theoretical

signal and combining information across channels. For example, traditional ToF and AoA estimation algorithms can be implemented under the near-field condition with different kernels and layers.

Kernel and Layers for ToF Estimation. ToF estimation can be done by choosing the following kernel and layer, where ϕ_l and θ_l are the empirical and theoretical phases at frequency f_l respectively, and d is the distance between tag and reader.

$$\begin{aligned} &\text{Kernel: } e^{-\boldsymbol{j}(\phi_l-\theta_l)} = e^{-\boldsymbol{j}(\phi_l-2\pi f_l d/c)} = e^{-\boldsymbol{j}\phi_l} e^{2\pi f_l \tau} \\ &\text{Layer: } \sum_{l=0}^{n} S(\tau) = \sum_{l=0}^{n} e^{-\boldsymbol{j}\phi_l} e^{2\pi f_l \tau} \end{aligned} \tag{5}$$

When using the above kernel and layer functions, $S(\tau)$ is the inverse Fourier transformation of the empirically measured phase value $\phi_1, \phi_2, ..., \phi_n$. Therefore, $S(\tau)$ is the time-of-flight expression of the empirically measured phases.

Kernel and Layers for AoA Estimation. Similar to the ToF estimation, we can also design kernel and layer functions to extract angle-of-arrive (AoA) estimation. For the AoA estimation, we can use the following kernel and layer functions, where ϕ_k and θ_k are the empirical and theoretical phases at antenna k, respectively. Δd is the distance between two neighboring antennas.

$$\begin{aligned} &\text{Kernel: } e^{-\boldsymbol{j}(\phi_k-\theta_k)} = e^{-\boldsymbol{j}(\phi_k-2\pi f k \Delta d sin(\psi)/c)} \\ &\text{Layer: } \sum_{k=1}^{m} S(\psi) = \sum_{k=0}^{m} e^{-\boldsymbol{j}\phi_k} e^{2\pi f k \Delta d sin(\psi)/c} \end{aligned} \tag{6}$$

$S(\psi)$ measures the similarity of the theoretical signal coming from angle ψ and the empirically measured phase value $\phi_1, \phi_2, ..., \phi_m$ received by m antennas. Therefore, correct AoA ψ is identified when $S(\psi)$ is maximized.

The summation layer, which sums up all the channels first by row and then by column, combines all the information for the final result. In this case, it combines near-field ToF and AoA estimations. We can develop more complex algorithms with the kernel-layer framework, such as the multipath-suppression algorithm in our paper.

C Direct Path Enhancement

We enhance the direct path and suppress the influence from multipath with a frequency domain algorithm [14]. Assume there are N paths with distances of $d_0, d_1, d_2, \ldots, d_N$, and d_0 is the direct path. The channel h_l of lth carrier can be expressed as:

$$h_l = a_0 e^{-j\frac{2\pi}{c} f_l d_0} + \sum_{i=1}^{N} a_i e^{-j\frac{2\pi}{c} f_l d_i}$$

a_i is the propagation attenuation of the ith path. To simplify the derivation without loss of generality, we assume $a_0 = a_i = 1$, $(i = 1, 2, 3, \ldots)$, and what we measure is the phase of channel response:

$$\phi_l = \angle h_l = \angle\{e^{-j\frac{2\pi}{c} f_l d_0} + \sum_{i=1}^{N} e^{-j\frac{2\pi}{c} f_l d_i}\}$$

If we have a rough estimation of d_0, called $\tilde{d}_0$, we can use this algorithm to enhance the part of $a_0 e^{-j\frac{2\pi}{c} f_l d_0}$ (direct path) and suppress the part of $\sum_{i=1}^{N} a_i e^{-j\frac{2\pi}{c} f_l d_i}$ (multipaths) for a better location estimation. In more detail, we use the prior knowledge of ROI to help determine the rough estimation of direct path $\tilde{d}_0$ with Alg. 1. Then we enhance the direct path profile and suppress profiles of other paths by Eqn. 3 because the enhanced phase $\tilde{\phi}_l$ can be written as:

$$\begin{aligned} \tilde{\phi}_l &= \angle \sum_{i=1}^{n} e^{j\phi_i} e^{j\frac{2\pi}{c}(f_i - f_l)\tilde{d}_0} \\ &= \angle\{e^{-j\frac{2\pi}{c} f_l d_0} \sum_{i=1}^{N} e^{j\frac{2\pi}{c}(f_i - f_l)(\tilde{d}_0 - d_0)} \\ &\quad + \sum_{i=1}^{N}[e^{-j\frac{2\pi}{c} f_l d_i} \sum_{i=1}^{N} e^{j\frac{2\pi}{c}(f_i - f_l)(\tilde{d}_0 - d_i)}]\} \end{aligned}$$

$\tilde{d}_0 \approx d_0$ so $(\tilde{d}_0 - d_0)\Delta f / c \ll 1$, and it leads to:

$$\sum_{i=1}^{N} e^{j\frac{2\pi}{c}(i-l)\Delta f(\tilde{d}_0 - d_0)} \approx \sum_{i=1}^{N} 1 = N$$

For multipath whose d_i is different from $\tilde{d}_0$, $\tilde{d}_0 - d_i$ is large so

$$\left| \frac{\sum_{i=1}^{N} e^{j\frac{2\pi}{c}(f_i - f_l)(\tilde{d}_0 - d_i)}}{N} \right| \approx \left| \text{sinc}\left[B\left(\tilde{d}_0 - d_i\right)/c \right] \right| \ll 1$$

The part of the direct path is much larger than the part of other paths, so the direct path is reinforced. $\tilde{d}_0$ helps to get rid of the leakage interference from multipath, and the following summation layer can make a better estimation of d_0 as the final output. Besides using the prior knowledge, other methods (*e.g.,* fingerprinting-based algorithm, Bayesian-based algorithm) can also be used to determine the rough estimation $\tilde{d}_0$, which is beyond the scope of this paper.

Exploring Practical Vulnerabilities of Machine Learning-based Wireless Systems

Zikun Liu[†], Changming Xu[†], Emerson Sie[†], Gagandeep Singh[†‡], Deepak Vasisht[†]
[†]*University of Illinois Urbana-Champaign,* [‡]*VMware Research*

Abstract

Machine Learning (ML) is an increasingly popular tool for designing wireless systems, both for communication and sensing applications. We design and evaluate the impact of practically feasible adversarial attacks against such ML-based wireless systems. In doing so, we solve challenges that are unique to the wireless domain: lack of synchronization between a benign device and the adversarial device, and the effects of the wireless channel on adversarial noise. We build, RAFA (RAdio Frequency Attack), the first hardware-implemented adversarial attack platform against ML-based wireless systems and evaluate it against two state-of-the-art communication and sensing approaches at the physical layer. Our results show that both these systems experience a significant performance drop in response to the adversarial attack.

1 Introduction

Next-generation networks, 5G and beyond, promise to be unprecedented in their scale and the diversity of applications, ranging from virtual reality to low power Internet of Things applications. Machine Learning (ML) has emerged as a key component of such future networks to deliver application-specific performance goals by optimally managing the diverse capabilities of these networks – multiple antennas, different spectrum bands, and smart surfaces. In academia, researchers have efficaciously applied ML for both communication [45, 55, 60, 79, 85] and sensing [6, 51, 59, 74, 94] applications. ML-based techniques are increasingly making their way to the industry, in both RAN (radio access network) and the network core. This trend has been accelerated by the recent shift of telcos to cloud-based execution models.

Our goal: We investigate the vulnerabilities of using ML in wireless systems. Our investigation is motivated by two reasons. First, wireless networks play a crucial role in many human-critical applications like autonomous driving, smart healthcare, factory control, etc. Any failure to meet network performance goals can have severe consequences in such settings. Second, in popular domains such as computer vision and natural language processing, past work has shown that an adversary can add small imperceptible noise to the inputs of a neural network making it predict completely different results [27, 76] (e.g., a turtle is classified as a gun). Several of these attacks have been reproduced in the real-world on state-of-the-art ML models in these domains [5, 48, 49, 71], showing that despite their impressive performance, the ML models are not robust. These practical attacks have promoted the development of new techniques for formal verification [40, 72, 73] and robust training [11, 34, 81, 84, 87, 90] in the vision and NLP domains.

Our goal is to explore the *practical* vulnerabilities of state-of-the-art ML-based wireless systems using adversarial attacks. To mount practical real-world adversarial attacks, an adversary must meet three requirements. First, it must not need access to the infrastructure in real-time, i.e., it cannot coordinate its transmissions with a benign sender, or access the signal sensed by a benign receiver. Second, it must be low complexity, i.e., it must not require large antenna arrays. Finally, it must be low power. It is relatively straightforward to jam the spectrum with blind high-power transmissions. However, jamming causes large-scale disruption to the spectrum and causes spectrum owners (e.g., telecom operators) to react. We are interested in small changes of the signal that specifically target the ML models in wireless systems, and expose their vulnerabilities.

Past work [2, 7, 9, 17, 18, 23, 43, 68] has studied adversarial attacks against ML-based wireless systems in simulation. These attacks do not meet the requirements above. Specifically, these attacks make unrealistic assumptions about the attacker capabilities. For example, they assume that an adversary can perfectly transmit adversarial signal or the attacker can directly manipulate the input matrix to the neural network. In practice, these assumptions do not hold. Adversarial signal undergoes wireless transformations described below before it arrives at a receiver. Similarly, directly altering the input matrix to the neural network requires access to the receiver.

Challenges: Consider the scenario shown in Fig. 1, where a (multi-antenna) base station communicates with a client device and uses ML-based models to deliver communication or sensing services. The adversary introduces small amounts of noise in the environment. Generating real-world adversarial attacks in such scenarios is challenging because of the underlying physics of wireless signal propagation. A typical adversarial attack takes an input to the ML model and crafts a noise vector specific to this input. This structured noise, when added to the input, causes the model to predict an incorrect output. In the wireless systems context, an attacker does not know the wireless channel between the client and the base station, and therefore does not know the signal being fed to the ML model. Secondly, the noise vector transmitted by the attacker is vastly different from what gets observed at the base station because: (a) Propagation effects: As the noise travels from the attacker to the end device, the noise vector undergoes the wireless channel experiencing reflection, attenuation, and phase shifts in the environment. (b) Clock offsets: the clock of

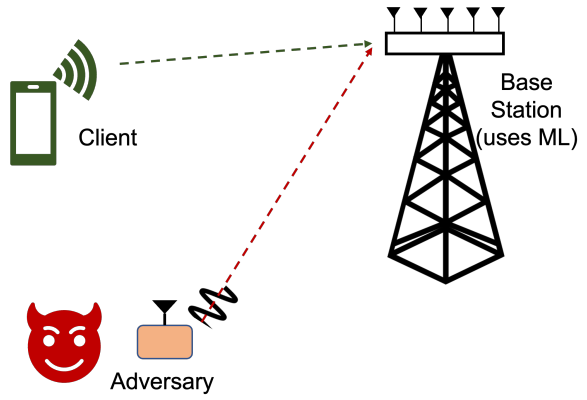


Figure 1: *A multi-antenna base station uses ML-based methods to deliver communication or sensing services to the client. The adversary transmits small amounts of noise and disrupts these services.*

the adversary is not synchronized with the end device, leading to random time and frequency offsets.

RAFA: We build the first real-world hardware-implemented adversarial attack platform, RAFA, that solves these challenges and targets ML-based wireless systems. Our system operates using a single antenna software defined radio and does not need real time access to the client or the base station in the attack setup. RAFA senses an ongoing communication on the wireless medium and introduces small amounts of noise (perturbation) to the medium that disrupts state-of-the-art ML-models. We demonstrate this attack on two state-of-the-art systems at the physical layer: one communication system (FIRE [55]) and one localization system (DLoc [6]). We show the effectiveness of RAFA in both white-box and black-box settings. The design of RAFA solves the following challenges:

Unknown Inputs: Adversarial algorithms [16, 27, 57] typically identify a perturbation that changes the output of a neural network *for a given input*. However, in practice, a wireless adversary, like RAFA, does not know the input to the ML model because: (a) the wireless channel from the client to the base station is unknown to the adversary, and (b) once the signal is received at the base station, the signal undergoes transformations (e.g., correcting for carrier frequency offsets) before it is fed to the ML model. Therefore, to model practical attacks, we require generating input-agnostic adversarial perturbations. We, first, design a universal adversarial perturbation (UAP), that focuses on changing the output on a distribution of inputs, rather than a single input. This allows us to be robust to the distribution of wireless signals, implying that we do not need to know the channel from the client to the base station. Furthermore, we develop differential versions of the pre-processing steps so that the adversary can generate perturbation vectors that remain adversarial even on pre-processed data points.

Lack of Synchronization: The adversary is not synchronized with the client or the base station. Therefore, its transmission is *not* aligned with the client in time or frequency. The lack of time synchronization creates temporal misalignment between the benign signal and the adversarial perturbation. Similarly, the lack of frequency synchronization creates a time-varying phase shift between them. Furthermore, such clock offsets are random and hard to predict beforehand. To counter such offsets, we create a robustness mechanism in our UAP design, that tests the perturbation vector for arbitrary phase offsets, and picks perturbation vectors that are robust to such offsets. By doing so, we shift the burden of dealing with the clock offsets from hardware to software, therefore simplifying our hardware design for the attack.

Channel-induced Transformation: Finally, the perturbation vector crafted by RAFA undergoes a channel transformation as it travels to the base station. The channel transformation changes both the amplitude and phase of the perturbation vector. Therefore, there are no guarantees on the value of the perturbation vector after the channel transformation. This means we cannot design a perturbation vector that is robust to these transforms. RAFA leverages reciprocity to counter this challenge. Specifically, the base station occasionally transmits beacons or responses to its legitimate clients. Our adversary overhears these transmissions and uses it to estimate the channel from the base station to itself. Due to the reciprocity principle, this channel is equal to the channel from the adversary to the base station. Once we know this channel, we use our robust UAP method to construct a perturbation vector that is effective even after the channel transform.

While these factors serve as natural protectors for wireless systems against adversarial attacks, RAFA demonstrates the ability to mount effective adversarial attacks despite these challenges. We have implemented RAFA using the USRP software defined radio against two state-of-the-art ML-based wireless systems: FIRE [55] (for MIMO communication), and DLoc [6] (for ML-based localization). For FIRE, RAFA's adversarial attack can reduce the median SNR (from original SNR of 17.8 dB) of the predicted channel by 4.1 dB on average compared to just 2.1 dB drop for Gaussian baseline. Similarly, for DLoc, RAFA increases the median localization error (from original error of 1.04 m) by 71 cm on average compared to just 2 cm increase for Gaussian baseline. Our results also present a preliminary version of potential defense strategies.

Contributions: Our main contributions are:

- We design a new robust adversarial attack against ML-based wireless systems that is input-agnostic and models real-world effects such as lack of synchronization.
- We leverage channel reciprocity to model the effect of wireless channel on adversarial perturbations.
- We demonstrate the first hardware-implemented adversarial attacks against ML-based wireless systems.

ML-based wireless systems are increasingly being pro-

posed in academia [6, 10, 35, 55, 94], and actively being explored in the industry [39, 64, 65] . Therefore, it is timely and important to explore the challenges posed by adversarial attacks in this context. To the best of our knowledge, our work is the first to demonstrate realistic hardware-implemented attacks against ML-based wireless systems. We believe RAFA will allow researchers and practitioners to test the practical robustness of ML systems before they get deployed widely in the real-world and have severe consequences for any failures when exposed to such attacks. We also envision that adversarial examples exposed by RAFA will lead to development of robust ML models. An early attempt at developing such robust models is demonstrated in Sec. 6.6.

2 Adversary Model

Objective: Our goal is to promote the development of robust ML-models by identifying the attack surface of ML-based wireless systems in the real world. We focus on the existence and performance of *practically feasible* wireless attacks. We identify *practically feasible* as attacks that can be implemented using real hardware and without requiring coordination with the base station or client. Furthermore, we are interested in vulnerabilities specific to ML-models in the wireless setting. Therefore, we do not consider jamming, which is a brute-force solution that blocks all communication in the medium. We consider the scenario in Fig. 1. A client communicates with a base station on the wireless medium. The base station can have multiple antennas. The base station relies on a machine learning based approach to deliver communication or localization services to the client. Some examples for ML-based communication systems are shown in Tab. 1.

Application	Examples
Communication	FIRE [55], OptML [10], NeuMac [35]
Localization	DLoc [6], IPS [93] , LAFA [38]

Table 1: *Examples of ML-based Wireless Systems*

Adversary Goal: The adversary wants to degrade the quality of ML-based location or communication service offered by the base station. The adversary aims to target specific ML-based services, and not jam the entire spectrum. To achieve this objective, the adversary transmits a carefully designed perturbation signal over the wireless channel. This perturbation gets superimposed at the receiver (which could be a cellular base station, access point, etc) with the benign signal transmitted from the client such as a cell phone. The receiver will later feed this seemingly intact but actually compromised signal into the ML-pipeline, negatively affecting its output prediction. Consistent with recent trends, we focus on neural networks as target ML models for this paper.

We describe the attacker properties in our threat model:

Coordination-free: We do not assume any coordination between the base station and the adversary (or between the client and adversary). This implies that the adversary is unsynchronized, i.e. has time and frequency shifts with respect to the other (benign) devices. The adversary also does not know when the transmission from the client begins or ends.

Base Station Information: The adversary does not know the location of the client or the base station. The adversary knows only public information about base station hardware, such as information which can be gleaned from FCC filings or standards documents.

Low-complexity: The adversary uses low complexity hardware. Even though the base station and the client may have multiple antennas, the adversary uses a single antenna transmitter. This reduces the cost and complexity of the attack, making it more universal, and generalizable.

Knowledge about the ML model: We assume that the adversary can sample data from the training distribution of the ML model running on the base station and knows the model family (e.g.,variational autoencoder) but not necessarily the architecture (e.g., fully-connected, convolutional), and the operations involved in the pre-processing pipeline. We believe that these assumptions are feasible for the real-world ML-based wireless systems because: (1) details about the model family are disclosed and accessible, (2) the attacker can access sample data simply by overhearing the client transmission, and measuring the corresponding wireless signals, (3) pre-processing pipelines (e.g., correcting for channel frequency offsets) are fairly standardized. We consider both white-box (access to model architecture and parameters) and black-box adversaries (no access to model architecture and parameters). Our results show that black-box adversaries are almost as effective as white-box ones.

Noise Budget: To avoid large-scale disruption to the wireless spectrum, we require that the L_∞-norm of the noise vector crafted by the adversary is bounded by a small constant $\varepsilon \in \mathbb{R}$. This prevents the noise being concentrated in individual sub-frequencies. We show the effect of the noise crafted with different values of ε on the model performance in Sec. 6.

Test Time Attack: We do not consider attacks that interfere with model training, the model is trained and fixed for our attacks. The adversary transmits a noise signal at test time.

3 System Overview

3.1 Target Systems

We consider two state-of-the-art ML-based wireless systems – one each for communication and sensing. In this paper, we focus solely on physical layer systems, while delegating investigation of attacks on higher layers to future work.

A. FIRE: Reciprocity for FDD MIMO systems – In order to achieve MIMO capabilities in 5G, base stations need to know the downlink wireless channel from their antennas to every client device. In FDD (Frequency Domain Duplexing)

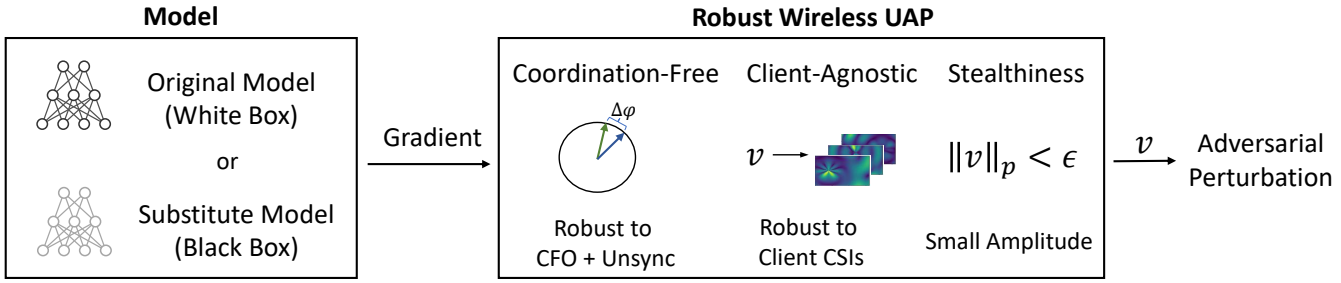


Figure 2: *RAFA Pipeline: RAFA works in both modes – white-box and black-box (with a substitute model). RAFA is robust to multiple wireless transformations such as clock offsets and channel transformations.*

systems, dominant in the United States, the client devices measure the wireless channel using extra preamble symbols transmitted by the base station and send it as feedback to the base station. However, this feedback is unsustainable and causes huge spectrum waste. Leveraging the intuition that both uplink and downlink channels are generated by the same underlying physical environment, recent work [55] proposed FIRE which uses an end-to-end ML based approach to predict the downlink channels without any feedback from the client. FIRE's ML model uses a variant of the variational autoencoders (VAE). FIRE takes uplink channels measured at the base station as input and predicts the downlink channel. The accuracy of the downlink channel prediction determines the performance of multi-antenna techniques like MIMO, MU-MIMO. Our goal is to induce errors in this model and make it predict erroneous downlink channels. Errors in downlink channel estimates reduce the communication efficiency of multi-antenna systems (e.g., MIMO).

B. DLoc: Deep Learning based Wireless Localization – DLoc [6] is a deep learning based wireless localization algorithm that overcomes traditional limitations of RF-based localization approaches such as mutlipath and occlusions. DLoc acquires wireless channels from four fixed access points to the user device that it wants to localize. The wireless channels are then sent into an autoencoder neural network as input. The network predicts the user location in 2D-Cartesian coordinates. DLoc achieves state-of-the-art localization accuracy in indoor localization. Therefore, we choose DLoc as the representative sensing application. Our goal is to increase the localization error of DLoc, and hence, increase failures in location-based applications (such as robotic navigation).

3.2 Operation Overview

Both the systems defined above rely on the wireless channel estimated at the base station. Our goal is to alter this wireless channel by transmitting a perturbation vector during the channel estimation process. During the channel estimation process, the client transmits a preamble. The preamble is by the standard and is, therefore, public knowledge. The base station receives the preamble and uses it to identify the wireless channel. Due to the broadcast characteristic of the wireless channel, we can pollute the channel estimation process by transmitting a noise signal. RAFA transmits the adversarial perturbation using our custom hardware platform. Thus, the signal received at the base station is a function of both the preamble transmitted by the client and the noise transmitted by our platform.

Fig. 2 shown an overview of RAFA's operation. RAFA's algorithm (discussed in Sec. 4) computes an adversarial perturbation which will be transmitted by the RAFA hardware, described in Sec. 5. The base station receives a sum of the signal transmitted by the client and the adversary, with channel and clock distortions. This combined signal then passes through the pre-processing steps and the ML-models defined above. We evaluate the performance of the ML-models with and without the adversarial perturbation in Sec. 6.

Notation: For the rest of this paper, we use capital-bold to denote matrices (**M,N,...**), small-bold to denote vectors (**u,v,...**), and mathematical font to denote functions or ML-models ($\mathcal{A}, \mathcal{L}, ...$).

4 RAdio Frequency Attack (RAFA)

In this section, we present our formulation of RAFA. We start with a brief background on adversarial attacks then discuss our attack formulation and modeling of wireless properties. Finally, we discuss our algorithmic implementation of RAFA.

4.1 Background on Adversarial Perturbations

For a given trained ML-model $\mathcal{M}$ mapping inputs $\mathbf{x_i}$ to outputs $\mathbf{y_i}$, an adversarial attack algorithm aims to find a perturbation vector, $\mathbf{v_i}$, whose magnitude is bounded by a small constant $\varepsilon \in \mathbb{R}$ in some norm, such that the loss function $\mathcal{L}(\mathcal{M}(\mathbf{x_i}+\mathbf{v_i}), \mathbf{y_i})$ is maximized, i.e., the output of the model for the perturbed input $\mathbf{x_i}+\mathbf{v_i}$ is far away from the target $\mathbf{y_i}$. Formally, the attack algorithm solves the following optimization problem:

$$\arg\max_{v_i} \mathcal{L}(\mathcal{M}(\mathbf{x_i}+\mathbf{v_i})) \text{ s.t. } ||\mathbf{v_i}|| < \varepsilon \quad (1)$$

This attack problem formulation is well studied and many approaches [16, 27, 57] have been proposed to approximate this optimization problem. We utilize the state-of-the-art Projected Gradient Descent (PGD) [57] method. PGD iteratively

takes steps in the direction of the gradient while restricting the total perturbation to be within ε. The constraint on the norm of the noise vector ensures that the perturbed vector is not significantly different from the original input $\mathbf{x_i}$. We note that PGD requires access to the model parameters for computing gradients. One way to handle the black-box setting where the attacker does not have this access is to train a surrogate model on the training distribution of the original model (to which the attacker has access) and transfer the attacks computed on the surrogate model to the original model.

As discussed before, our model does not know the input to the model, therefore we rely on universal adversarial perturbations (UAPs) [58]. Instead of computing a different additive noise for each input, UAPs compute a single additive noise that is effective for all inputs, $X = \{\mathbf{x_i}\}_{i=1}^{N}$ in the training distribution of the ML model. One way to encode this is by maximizing the expected value of the loss.

$$\arg\max_{\mathbf{v}} \mathbb{E}_{\mathbf{x_i} \in X} \mathcal{L}(\mathcal{M}(\mathbf{x_i} + \mathbf{v}), \mathbf{y_i}) \text{ s.t. } ||\mathbf{v}|| < \varepsilon \tag{2}$$

Typically, this optimization problem is approximated by iteratively computing input-aware gradient updates (obtained from the original or a surrogate model) to a perturbation vector over the training set [58]. We use a variant of the UAP, called robust UAP [83], which designs a universal adversarial perturbation such that it is robust to transformations (such as image rotations, and translations etc for vision models).

4.2 Our Attack Formulation

For simplicity, we start with a single input case, i.e., we wish to design a perturbation vector that is specific to wireless channel matrix, $\mathbf{H}$, observed at the base station. $\mathbf{H}$ is a $N_{ant} \times N_{subc}$ matrix, where N_{ant} is the number of antennas on the base station and N_{subc} is the number of OFDM subcarriers. Our goal is to search for a perturbation vector, $\mathbf{v}$, of length N_{subc} that can disrupt a machine learning model $\mathcal{M}_\theta$, where θ is the set of weights for the model. Specifically, we aim to optimize:

$$\arg\max_{v} \;\; \mathbb{E}_{T_\tau \in \mathcal{T}_\tau} \mathcal{L}(\mathcal{M}_\theta(\mathcal{P}(\mathbf{H} + T_\tau(\mathbf{v}))), \mathbf{y}), \;\; \text{s.t. } ||\mathbf{v}||_p < \varepsilon \tag{3}$$

where $\mathcal{L}(\cdot,\cdot)$ is a chosen loss function to measure the difference between the model's prediction and the ground truth, ε restricts the l_p norm of the perturbation vector. We also define two new abstractions in the equation above: $\mathcal{P}(\cdot)$ is the pre-processing pipeline used by the base station, before it is fed to the ML-model, $\mathcal{M}_\theta$. Similarly, $\mathcal{T}_\tau$ represents the transformations T_τ the perturbation vector goes through before it arrives at the base station. These transformations are parameterized by τ. Next, we discuss how these abstractions model the real-world effects for wireless systems.

4.3 Modeling Pre-Processing

Both our target systems perform pre-processing on the estimated channel and feed the processed channel to the neural network for prediction. We model these pre-processing steps as $\mathcal{P}$. FIRE involves two pre-processing steps: it standardizes the Carrier Frequency Offset (CFO) and hardware detection delay across different measurements of the same channel. DLoc transforms the channel into the 2D-cartesian heatmap representing the probability of a signal originating from a given location (using signal-processing approaches like Fourier Transforms).

We need to represent $\mathcal{P}$ as a differential operation, to enable optimization in Eqn. 3. In both pre-processing methods, there are non-differentiable functions such as `argmax()`, `ceil()`, `sign()` that hinder gradient propagation for our optimization problem. Therefore, we use a differentiable approximation of these functions which is supplied by common ML frameworks such as Pytorch [61].

4.4 Modeling Lack of Synchronization

Since the adversary is not synchronized with the client or the base station, the noise transmitted by it experiences two kinds of distortions that must be modelled by $\mathcal{T}_\tau(\cdot)$ defined above:

Carrier Frequency Offset: The oscillators at the adversary and the base station are not synchronized. This leads to a CFO between them. This frequency offset, denoted by Δf, will continuously add a phase shift in the received signal $\hat{s}(t)$ with respect to the true signal $s(t)$ over time: $\hat{s}(t) = s(t)e^{j2\pi\Delta f t}$.

Since the client and attacker have different transmission chains, their CFO with respect to the base station is also different. Assume the CFO between the client and the base station is Δf_1, the CFO between the attacker and the base station is Δf_2, thus the CFO discrepancy will add phase offset $e^{j2\pi(\Delta f_1 - \Delta f_2)t}$ to the transmitted adversarial signal with respect to the client signal. This is a time-varying effect, implying that the sum of the client signal and the adversary signal changes over time. Since t, Δf_1 and Δf_2 are random, we can simplify this effect as a multiplication of $e^{j\phi}, \phi \in [0, 2\pi]$ to the adversary signal.

Unsynchronized Transmissions: Ideally, the adversary should start transmitting the perturbation signal at the same time when the client starts transmitting the preamble so that the perturbation can be superimposed at the base station precisely. In the real-world setting, this is hard to achieve since the attacker cannot coordinate with the client preemptively or synchronize its clocks. There are two possible approaches to solve this problem: (a) A part of the signal preamble is used for packet detection before the channel estimation phase. One can design an attacker that detects the start of the packet, and starts transmitting a perturbation in response. This can achieve coarse-grained synchronization, but requires fast processing (e.g., FPGAs). (b) An alternative approach is to let the

adversary transmit multiple copies of the perturbation signal, and deal with the resulting large mis-alignment.

For simplicity, we go with the latter approach, which requires much lower overhead. This causes a random time delay Δt between the benign client signal and the adversary signal in the time domain. Due to the properties of OFDM, this is equal to a phase offset $e^{-j2\pi\Delta t f_i}$, where f_i is the frequency of the ith OFDM subcarrier. Note that, this is similar to timing misalignment in typical OFDM receivers [24, 77], wherein a portion of the OFDM symbol is repeated (in the cyclic prefix). Any sample misalignment adds a phase that is linearly dependent on the amount of misalignment and the frequency of the subcarrier. We model this phase shift in $\mathcal{T}_\tau(\cdot)$.

4.5 Modeling Channel Transformations on the Perturbation Vector

Like any other wireless signal, the perturbation vector transmitted by the adversary goes through the wireless channel. Let us say that the wireless channel matrix for the adversary is, $\mathbf{H_a}$, with dimensions $N_{ant} \times N_{subc}$ (same dimensions as $\mathbf{H}$ for the client). Then, if the adversary transmits the perturbation vector, $\mathbf{v}$, it is received at the base station as $\mathbf{H_a v} + \mathbf{g}$, where $\mathbf{g}$ is additive white Gaussian noise.

Each element in $\mathbf{H_a}$ has an amplitude and a phase. Therefore, the final outcome of the added perturbation can be unbounded, if we do not know $\mathbf{H_a}$. However, $\mathbf{H_a}$ can only be measured at the base station, that too in the absence of the client signal. Clearly, measurement by this method is not possible for the attacker because it does not have any coordination with the client or the base station. Therefore, it is pertinent to find an alternative method to measure $\mathbf{H_a}$.

To estimate $\mathbf{H_a}$, we leverage channel reciprocity. Channel reciprocity is a fundamental physical principle that states that wireless signals take the same path in either direction between any two devices. Therefore, the wireless channel from the adversary to the base station is equal to the wireless channel from the base station to the adversary (modulo some hardware differences). RAFA listens to the wireless medium and captures signals transmitted by the base station (either periodic beacons or communication with legitimate clients). This allows the adversary to estimate an approximation to $\mathbf{H_a}$. We discuss in Sec. 5 how this step is implemented in practice.

4.6 Generating Practical Adversarial Attacks

The above factors will jointly modify the adversarial perturbation signal v^i transmitted in the ith subcarrier by the following transformation function:

$$\mathcal{T}_{\phi,\Delta t,\mathbf{h}_a^i,\mathbf{g}_i}(\mathbf{v}^i) = \mathbf{v}^i e^{j\phi} e^{-j2\pi f_i \Delta t} \mathbf{h}_a^i + \mathbf{g}_i \quad (4)$$

Note that the CFO term ϕ is invariant to the frequency, so it's the same across all the subcarriers.

Algorithm 1: Robust Wireless UAP (`RW-UAP`)

Input : Dataset $\mathbf{x_i}, \mathbf{y_i} \in \mathcal{H}$, network model $\mathcal{M}_\theta$, l_∞ norm threshold ε, desired network loss value δ, attacker channel $\mathbf{H_a}$, number of epochs ep

Output : Robust universal perturbation $\mathbf{v_g}$ for dataset X

```
Initialize v_g ← Uniform(−ε, ε)
for n ← 0 to ep − 1 do
    for each batch B_i ⊂ H do
        Δv_i ← RW-PGD (B_i, v_g, M_θ, ε, δ, H_a)
        v_g ← (v_g + Δv_i).clamp(−ε, ε)
    end for
end for
return v_g
```

Now that we have characterized both $\mathcal{P}$ and $\mathcal{T}_\tau$, we try to find a perturbation that is robust to these wireless factors. We build on the algorithm presented in [83] which works in the vision domain to work with $\mathcal{P}$ and $\mathcal{T}_\tau$, Algorithm 1 shows the pseudocode for generating Robust Universal Adversarial Perturbations in the wireless setting. It contains twp loops to iterate multiple times on the training dataset and on every batch of data points respectively. We first initialize the perturbation vector $\mathbf{v_g}$ randomly. Because of random initialization, we can generate several different UAPs by running the algorithm multiple times. The algorithm iteratively updates the initial perturbation with the goal of being adversarial for all elements in the training set. Given a set of training data points $\mathcal{H}$ sampled from the training distribution of the ML model, the algorithm iterates over batches $\mathbf{B} \subset \mathcal{H}$. During each epoch, it iterates over every batch, $\mathbf{B}_i$, finding an adversarial direction vector $\Delta\mathbf{v_i}$ that is robust to wireless factors for that batch using the Robust Wireless PGD (`RW-PGD`) algorithm shown in Algorithm 2. $\Delta\mathbf{v_i}$ is then added to $\mathbf{v_g}$ and the result is projected back so that the updated $\mathbf{v_g}$ does not violate the constraint on its norm.

Next, we describe the `RW-PGD` algorithm shown in Algorithm 2. It contains two loops to iterate multiple times on the input data points and on different transformations respectively. The algorithm takes a batch of inputs, $\mathbf{B}$. It then first randomly samples N tuples of wireless factors including CFO, resynchronization, and gaussian noise as described in the previous section. Increasing the value of N increases both the robustness of the output perturbation and the runtime of the `RW-PGD` algorithm. We chose a value of N that balances the tradeoff between cost and robustness. At each iteration, we transform our current perturbation by each of the N transformations. We then compute the mean loss over all data points in the batch added to each of our N transformed perturbations and conduct gradient ascent on it aiming to increase the mean loss. Unlike traditional PGD [57], we compute a single vector per batch of data points. We further found that using the raw gradient

Algorithm 2: Robust Wireless PGD (RW-PGD)

Input : Batch of data points $\mathbf{x_i}, \mathbf{y_i} \in \mathbf{B}$, current perturbation $\mathbf{v_g}$, network model $\mathcal{M}_\theta$, l_∞ norm threshold ε, desired network loss value δ, attacker channel $\mathbf{H_a}$, maximum number of epochs ep, learning rate α, number of transformations N

Output : Robust perturbation $\mathbf{v}$

Initialize $\mathbf{v} \leftarrow 0$
Sample N sets of wireless factors, $\tau_j \leftarrow \{\phi_j, \Delta t_j, g_j, \mathbf{H_a}\}$, uniformly at random
for $n \leftarrow 0$ **to** $ep - 1$ **do**
  for $\mathbf{x_i}, \mathbf{y_i} \in \mathbf{B},\ j \in [N]$ **do**
    Get predictions for N transformations: $\mathbf{y^*_{i,j}} \leftarrow \mathcal{M}_\theta(\mathbf{x_i} + \mathcal{T}_{\tau_j}(v_g + v))$
  end for
  $\mathcal{L}_B \leftarrow \frac{1}{N \cdot |\mathbf{B}|} \sum_i \mathcal{L}(\mathbf{y_i}, \mathbf{y^*_{i,j}})$
  if $\mathcal{L}_B > \delta$ **then**
    break
  end if
  $\mathbf{v} \leftarrow (\mathbf{v} + \alpha \cdot \Delta \mathcal{L}_B).\text{clamp}(-\varepsilon, \varepsilon)$
end for
return v

to update was more effective than the $sign(\cdot)$ of the gradient in our case. Note that, $\mathbf{H_a}$, i.e., the wireless channel between the attacker and the base station, is fixed and obtained by the attacker without sending any preambles.

$\mathbf{H_a}$ is continuously sampled by the attacker, so we need to recompute the attack on the fly. This is inevitable as different $\mathbf{H_a}$ have essentially unbounded effect on the perturbation. Thus, we further speed up our algorithm by computing multiple UAPs at the same time. We have chosen algorithm parameters that maximize the speed for our required performance.

By using the above algorithm, we are able to find robust universal perturbations that work effectively against a variety of wireless factors. In order to truly expose the vulnerabilities of wireless system models, we show in a real-world hardware setting that our attack is effective.

5 Hardware Design

After identifying the adversarial noise to inject, we ask if this noise is feasible in practice, i.e., can we design hardware that can introduce such noise? In this section, we design a generic physical hardware platform as an attacker to inject such perturbation into wireless channels. We believe that this step is novel and unique to the wireless domain because no past work has demonstrated hardware-driven attacks on wireless systems. We describe the design principles of RAFA's hardware platform and its implementation.

5.1 Design Principle

We design our platform with following design principles:

No Synchronization Needed: We design RAFA's platform to have minimum assumptions, no requirement to access the client and the base station. Our attack belongs to the realm of pilot contamination but previous literatures [28, 36, 37] all assume that the attacker know the exact timing to synchronize with the client so that it can transmit the noise signal at the same time when client transmits the preamble. In our design, we get rid of this assumption and instead, transmit the attacker's perturbation in a unsynchronized manner. According to Sec. 6.2, our perturbation is robust to such unsynchronization on the sample level, as a result, we don't need the timing of the client to conduct effective attack.

Leveraging Reciprocity: Recall that we leverage reciprocity to estimate $\mathbf{H_a}$, the channel from the attacker to the base station. However, reciprocity requires correcting for the hardware effects caused by each respective transmitter. These effects comprise of phase and signal strength variations. The phase variations are captured in the transformations caused due to CFO and timing offsets for our perturbations. Therefore, in our attack, we just need to calibrate for the difference in transmit power between the attacker and the base station.

Specifically, the transmit power is different at the attacker and the base station (the base station transmits a much higher power). Although transmit powers don't affect the applications enabled by reciprocity such as beamforming, signal nulling, etc. [41, 47, 53], we need to know the true channel value including the power in order to control the received perturbation power at the base station, as shown in Algorithm 2. The adversary can obtain the transmit power of the base station using public documents like FCC filings. We can estimate the transmit power of the adversary hardware using specification sheets. Then, the adversary computes the difference between the two and uses it to correct for the transmit power difference.

Single Antenna: We design RAFA's hardware platform with a single antenna. This limits our hardware complexity and making it easy to implement (single transmit-receive chain). One class of adversarial attacks on traditional systems [42, 44] is only effective when the attacker has same or more number of antennas as the base station. In our case, we show that even with one antenna, it is possible to mount reasonable attacks.

However, this choice limits the capability of the attack. While the input channel matrix, $\mathbf{H}$, has dimension $N_{ant} \times N_{subc}$, the perturbation is a vector $\mathbf{v}$ with length N_{subc}. Inherently, this implies that the perturbation has less control over the final output. Mathematically, $\mathbf{v}$ operates in a 1-dimensional subspace of a N_{ant} dimensional antenna space. The larger the N_{ant}, the less powerful our adversary. We expect multi-antenna adversaries to be more effective, but we chose single antenna adversaries for their low complexity.

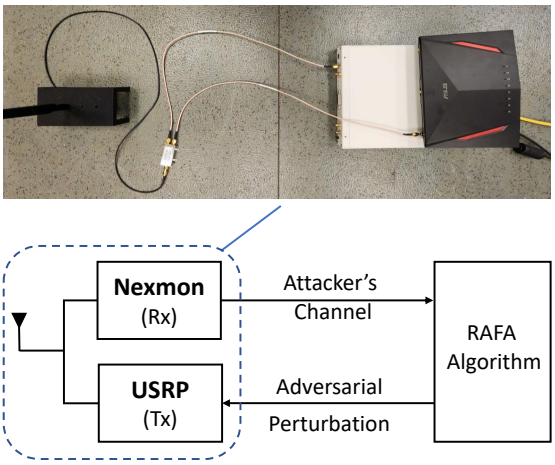


Figure 3: *(Top) The top-down view of the attacker hardware platform. (Bottom) The platform consists of a Nexmon-receiver and a USRP transmitter – both share a single antenna connected through a splitter.*

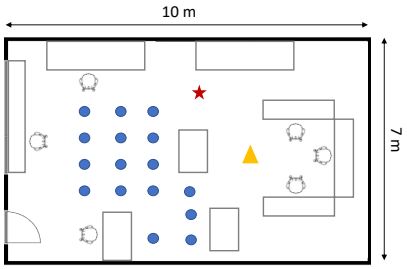


Figure 4: *Layout of the experimental environment: We conduct our experiments in a lab setup measuring 10m by 7m. (Blue: Client Locations, Yellow: Base station, Red: Attacker)*

5.2 System Implementation

We use the Nexmon [29] tool to measure wireless channels. We build a 4-antenna base station using a commercial Asus RT-AC86U router with bcm4366c0 WIFI Chip. This router reports channel state information (CSI) and signal strength (RSSI) for each received packet. We configure a client to connect to this router and send Wi-Fi packets. The router periodically sends broadcast beacons, as is standard for the Wi-Fi protocol and is set to work in the 5GHz frequency band.

For the attacker, we use a Nexmon receiver and a USRP (X310 [21], a software-defined radio) as the transmitter. The Nexmon receiver and the USRP transmitter share a single antenna through a splitter. The Nexmon receiver measures the wireless channel from the base station to itself and feeds it into the RAFA system. RAFA applies the reciprocity correction to estimate, $\mathbf{H_a}$, and computes an adversarial perturbation, $\mathbf{v}$, and transmits it using the USRP software-defined radio. The setup is shown in Fig. 3

We implement the client using a USRP software-defined radio (X310) equipped with one antenna and we configure it to transmit the 802.11ac packets generated from MATLAB wireless toolbox. This signal is received by the base station and used to measure the wireless channels as the input to the neural networks in the wireless applications mentioned in Section 3.1. Note that, the client and attacker do not have any time or frequency synchronization between them.

During our experiment, we deploy RAFA on a local machine with RTX3070 GPU, a low-end GPU. Our implementation is written in PyTorch. For attacking FIRE, it takes 13 seconds on average to generate a single perturbation. We believe that further speedup can be achieved using more advanced computing resources.

6 Results

In this section, we show the effectiveness of the RAFA attack against FIRE and DLoc. We further show that our attack can be performed in a black-box setting. Finally, we show that adversarial training can be used in order to harden our models and defend against these attacks.

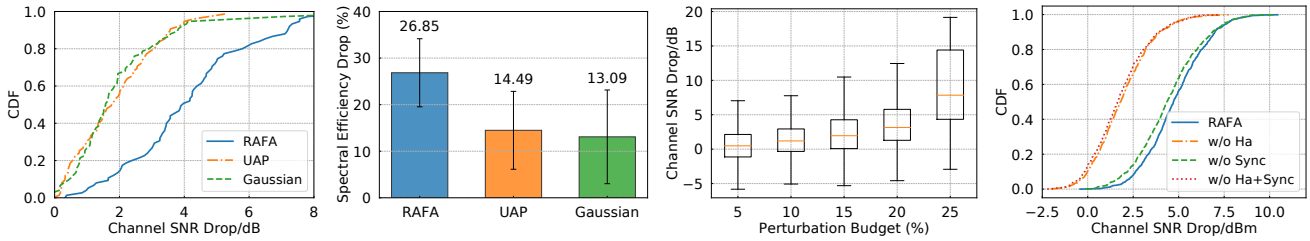
Baselines: We compare RAFA against two baselines: (a) Gaussian noise, and (b) vanilla UAP. Gaussian noise transmits randomly sampled noise into the air. Vanilla UAP designs and injects perturbations attacks that do not include robustness to wireless transformations implemented by RAFA. For fairness, we evaluate each method with the same budget ε on the magnitude of the perturbation vector measured in L_∞-norm.

6.1 Wireless Systems Re-implementation

We do a best-effort re-implementation of our target systems: FIRE [55] and DLoc [6] using details provided in the respective papers and by email exchanges with the authors.

We re-implement the Variational Autoencoder for FIRE. We adopt 7 linear layers in both the encoder and decoder networks, which is 3 layers more than the original design to optimize FIRE's performance. We train FIRE using dataset collected in our environment. We collect 10000 data points by moving the antenna randomly in a lab space shown in Fig. 4. The size of the lab is 10m by 7m, and is composed of many reflectors (like metal cupboards, white-boards, etc.) and obstacles. We split the dataset in the radio of 8:2 for training and testing , and the training takes roughly 20 minutes. Note that the training only needs to be done once before the attack is initiated. Our trained model achieves a channel SNR of 17.8dB on the test dataset and confirm the SNR of 15.8dB on validation dataset, which is consistent with the performance of FIRE reported in [55].

Training DLoc requires dataset collected using a robot (which performs joint mapping and localization). We do not have access to the robot, thus cannot recreate the original experiments. However, the datasets and code for DLoc are in the public domain. Therefore, we train DLoc model using the datasets collected by its authors with the name 'jacobs_Jul28' and keep the original neural network architecture and training setting. The dataset contains channel estimation matrices collected from 4 routers and each router has 4 antennas (so 16



(a) Attack Performance (b) Spectral Efficiency Drop (c) Perturbation Budget Analysis (d) Ablation Study

Figure 5: *Attacking FIRE: (a) RAFA causes a median drop of 4.06 dB on FIRE's channel prediction accuracy, over 2X better than baselines. (b) This corresponds to a 26.85% drop in spectral efficiency. (c) Increasing the perturbation budget increases the effectiveness of the RAFA adversary. (d) Modelling channel reciprocity is the dominant reason for RAFA's improved performance.*

antennas in total), and the training takes roughly 40 minutes. We achieve the localization accuracy of 1.04m on the test dataset which is randomly split with the ratio of 3:7 to the training dataset. This is consistent with the results reported by the authors in [6]. To create the adversarial perturbation for RAFA, we sample channels from the same distribution as the data used for training.

6.2 Adversarial Attacks against FIRE

To mimic a real-world setup, we deploy the base station at a fixed location, while the client moves across the 16 positions shown in Fig. 4. The attacker is also deployed at a fixed location shown in the figure. The attacker and the client can be in line-of-sight or non-line-of-sight.

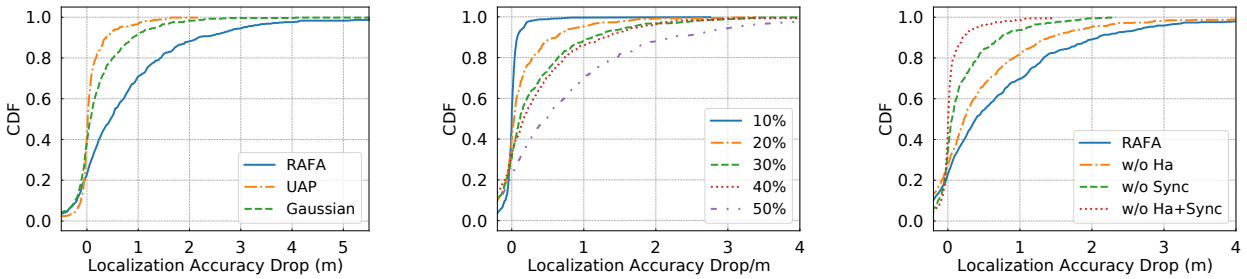
Attack Effectivenss: We first compare RAFA with other two baselines in the real world attack scenario. FIRE predicts downlink channels, given uplink channels. Our goal is to reduce the SNR of the predicted channels, so as to disrupt multi-antenna communication between the base station and the client. To perform this experiment, we perform five attack rounds for every method at each client location. In each round, the adversary transmits the RAFA adversarial perturbation for 10 seconds. Then, we transmit the UAP adversarial perturbation and Gaussian perturbation each for 10 seconds at the same perturbation budget as RAFA which is set to be maximum of 20%. The budget is the ratio of the maximum value that's allowed for any element of the perturbation vector compared to the average amplitude of the benign channel estimates and is ε in Algorithm 1. Overall, we get 8000 channel estimates at each client location for every baseline. We set the learning rate of RW-UAP to start from 10 and decay by 0.6 for every epoch. We set the number of total epochs in RW-UAP to be 3 and we only use a random 10% of the training dataset for each epochs to accelerate the training which leads to a running time of roughly 30 seconds with the setup in Sec. 5. We set the RW-PGD iterations to be 10, and 10 transformations in each iterations are used to get the robust perturbations. We use the same parameters (e.g., number of epochs, % of training dataset) for training Vanilla UAP.

We show the effect of different attack methods on FIRE prediction in Fig. 5a. Each of these methods causes the SNR of the predicted channel to drop. We plot the CDF of this drop in channel SNR across all of our attacks, e.g., a CDF value of 0.3 with a corresponding drop of X dB indicates that 30% of the inputs had a drop of X dB or less in performance. It shows that when being attacked by RAFA, the SNR of the channel predicted by FIRE drops by 4.06dB on average (7.15 dB drop on the 90-th percentile). Traditional UAP attacks and Gaussian baseline are not as effective. RAFA outperforms the baselines by 2.21× and 2.14× respectively on this metric. Our benefits over Gaussian noise stem from the directed nature of our attack, i.e., we specifically target the ML model and find its vulnerability. On the other hand, the UAP-based model finds perturbations that are directed at the ML model, but undergo transformations in-air which render it ineffective when implemented in the real world. This result highlights that: (a) FIRE is vulnerable to practical adversarial attacks, even when the adversary uses low-complexity hardware, and (b) modelling the wireless transformations on the adversarial noise (as done in RAFA) are essential for practical adversarial attacks.

We also show how these adversarial attacks affect the application quality in the real world. We plot the spectral efficiency (bits per Hz) results with the budget of 20% in Fig. 5b. Spectral efficiency is the data rate that can be transmitted over a given bandwidth and can be computed through channel SNR [62]. With small amount of budget, RAFA is able to shrink the user data rate by 26.85% which is two times as effective as the UAP attack. This will affect the user experience severely, especially in latency critical applications such as online meetings.

Finally, we note that, we focus on reducing SNR for FIRE, because FIRE is trained to optimize for SNR. SNR is also a key metric for any communication techniques. Independent mechanisms like coding and CRC checks do not prevent against attacks like RAFA. For example, if a coding scheme is chosen to optimize for SNR X dB, it won't be sufficient if the actual SNR is X-5 dB (when under attack).

Effect of Adversarial Budget: In the same setup as above, we study the effect of budget on the effectiveness of the attack. We experiment with 5 different budget parameters and plot the SNR drop for these parameters in Fig. 5c. As expected, as



(a) Attack Performance (b) Perturbation Budget Analysis (c) Ablation Study

Figure 6: *Attacking DLoc: (a) DLoc's original localization error (median) is 1.04 m. RAFA increases this by 0.71m. (b) The attack performance improves with increasing perturbation budget. (c) Synchronization robustness is the most important factor in attacks on DLoc.*

Budget (%)	10	20	30	40	50
Power Ratio-FIRE (%)	1.5	4.7	8.6	14.7	12.0
Power Ratio-DLoc (%)	1.3	8.6	20.2	34.6	51.8

Table 2: *Power Ratio vs Budget*

the budget parameter increases, RAFA's attack becomes more effective. With the budget parameter set to 25%, the SNR of the predicted channel drops by 7.4 dB on average, with lower drops at lower budget values (e.g., 2.14 dB at 15%).

Note that, the budget defines what's the maximum absolute value of any perturbation subcarrier, and not the average value. Therefore, the average power of the perturbation is expected to be lower than the budget. We compare the average power of the adversarial signal to the average power of the user signal at the base station. We show this ratio as a function of the budget parameter in Table. 2. As shown, for 20% budget, RAFA's perturbations utilize <5% power on average compared to the user signal. Even with a 50% budget parameter, this value is only 12%. This shows that RAFA's attacks are surreptitious. Note that, counter-intuitively, the power ratio at 50% is lower than 40% budget parameters. This is possible because bigger budget parameters allow some subcarriers to have larger peaks, and enable more flexibility for RAFA to choose "peakier" more effective perturbation vectors.

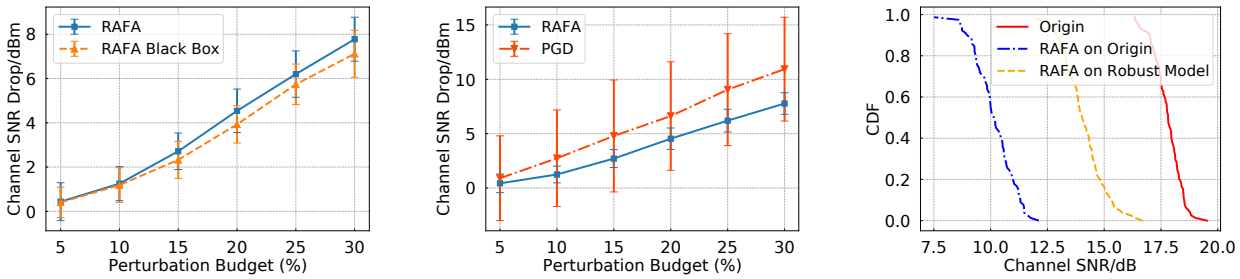
Ablation Study: Next, we analyze the contribution of different components of RAFA. We conduct an ablation study on attacking FIRE using the same dataset that we used for real-world attack experiment. We compare the original RAFA effectiveness with the following cases: removing the attacker channel ($\mathbf{H_a}$), removing the synchronization robustness term (Sync), and removing both of them. The results are shown in Fig. 5d. For FIRE application, removing the knowledge of the attacker's channel has the most significant impact on the effectiveness of the attack, the channel SNR drop decreases by 59.5% compared to the optimal performance achieved by RAFA. This is because our `RW-UAP` algorithm derives an adversarial perturbation specifically for a given attacker's channel. When removing the synchronization robustness term, the SNR drop decreases by 7.5%. We believe that this effect is milder because FIRE's design includes some pre-processing to normalize CFO, hardware detection delays, etc.

6.3 Adversarial Attacks against DLoc

Next, we study the effect of RAFA on DLoc. DLoc conducts user localization using the channels estimated from 4 routers with a total of 16 antennas as the input to a neural network, consisting of 12 Resnet blocks [32]. Compared to FIRE, which uses a single router, four antennas, and a simpler network structure, the attack scenario is harder.

Experiment Setup: As noted before, getting ground truth location estimates for DLoc in new environments requires a robot for data collection. We perform our attack in a trace-driven simulation using the data collected by the authors as we do not have access to their robot. We randomly sample, $\mathbf{H_a}$, the attacker's channel to each access point from the set of channels in the original DLoc dataset. Then, RAFA's perturbation undergoes the attacker channel in addition to random time and frequency offsets. We repeat this experiment with different values of $\mathbf{H_a}$ to remove any bias. Our attacker is still a single antenna attack. We limit to a set of 128 datapoints, out of 8008, randomly sampled from the original training set and use only 4 transformations during `RW-PGD`. We train the perturbation for 3 epochs. The learning rate is set to 0.006 and decays by 0.99 per iteration. We evaluate on randomly sampled ∼500 datapoints from the test set and average the performance of the attack over 8 randomly sampled transformations to generate different wireless transmissions. The same parameters are used for training Vanilla UAP.

Attack Effectiveness: The adversary aim is to reduce DLoc's localization accuracy. We plot the accuracy degradation caused by RAFA, traditional UAP, and Gaussian noise in Fig. 6a. The attacker has a single antenna simultaneously attacking 16 infrastructure antennas, so we set a budget of 50%. The original DLoc median localization accuracy is 1.04 meters, RAFA is able to increase this error by 0.71 meters. Furthermore, RAFA increases the error by more than 1 meter in 30 % of cases. This is significant as some safety-critical applications such as autonomous driving [66,67] are sensitive to even 0.1 meters of localization accuracy drop. Our two baselines, UAP and Gaussian are only able to drop the accuracy by 0.02 m, and 0.08 m respectively. Similar to our discussion before, this shows the benefit of our directed, robust attack.



(a) Black box attack (b) Comparison to PGD attack (c) Robust Training

Figure 7: *Other Attack Models and Defense on FIRE: (a) A black-box attack, where RAFA does not know about the model running on the base station is feasible and has similar attack performance to the white-box attack. (b) An unrealistic attack, where adversary has access to model inputs, performs better than RAFA, but is not practically feasible. (c) Our adversarial training approach can improve model robustness on FIRE.*

Next, we study how choosing different budgets will affect RAFA's potency on DLoc. We increase the budget from 10% to 50% and plot the DLoc accuracy on Fig. 6b. We also calculated the power ratio of the perturbation generated on DLoc in the second row of the Table. 2. As expected, as the perturbation budget increases, the power used by RAFA increases. Similarly, the attack efficacy increases. We highlight that the power used by RAFA for attacking DLoc is higher than that for FIRE. This is because we have a single-antenna attacker attacking a sixteen-antenna system spread out in space (one-dimensional control in a 16-dimensional space). RAFA adapts to this large space by increasing the power of its transmitted noise, achieving an error increase of 0.71 m at 50% budget.

Ablation Study on attacking DLoc: We conduct an ablation study on DLoc to understand the contribution of RAFA's different components, using the same 50% budget from above. Similar to ablation study settings on FIRE, we compare the original RAFA effectiveness with the cases when removing Ha, removing Sync and removing both, and the results are shown in Fig. 6c. Different from FIRE's ablation study results, it shows that removing Sync term will bring down the effectiveness the most, the median localization accuracy drop decreases by 73.4% compared to the original RAFA. This is because DLoc localizes the user by receiving the signal at four routers instead of one in FIRE case. Thus misalignment becomes much more significant when the attack signal reaches multiple routers and modeling this effect is necessary. Compared to the ablation study on FIRE, we confirm that different components in RAFA modeling have different impact on different applications. So, to expose vulnerabilities on generic wireless algorithms, we should leverage the end-to-end model with all the components of RAFA.

6.4 Comparison to Input-Aware Attacks

RAFA is an adversarial attack that aims to work in the real-world. Therefore, it does not assume access to the input. What is the impact of this assumption on RAFA's performance? What if we give RAFA access to the input. We evaluate this hypothetical case next. We compare RAFA's adversarial performance with input-aware attack, e.g, PGD. Since this attack is not realistic, we evaluate this in a simulator. We plot the result in Fig. 7b. Note that we implement the PGD by using Algorithm. 2, for each data point in the test dataset, we get a PGD perturbation that is robust to wireless properties, but this perturbation is not universal across data point.

The results show that an input-aware attack is more effective. At 30% budget on FIRE, the input-aware attack achieves an SNR drop of 10.97 dB, compared to the 7.77 dB drop of RAFA. This result highlights that the properties of the wireless medium (e.g., the adversary not knowing the channel from client to base station) provide some natural protection against adversarial attacks. However, even with this protection, real-world adversarial attacks are possible and effective.

6.5 Black Box RAFA

In this section, we evaluate RAFA under a preliminary black box setting to show its feasibility, where the attacker knows the model family but not the specific architecture or weights. In order to conduct the black box attack, the attacker can train a substitute model on a dataset with a similar distribution to attack. The attacker can later use the obtained perturbation from the substitute model to attack the true model.

In order to conduct the black box attack, we use a substitute model with a different architecture (4 less layers, different number of neurons, and batchnorm). Using the same pipeline for RAFA we attack on a different dataset collected in the same lab. We then use that perturbation to attack the original model in a trace-driven simulation. Fig. 7a compares the performance of RAFA in white box and black box scenarios under different perturbation budgets. As shown, the performance of the black-box adversary closely matches that of the white-box adversary, with only a minor drop. At 25% perturbation budget, the black-box adversary causes a 5.7dB of channel SNR drop for FIRE which is only 0.5dB less than the white box setting. This result shows that while access to the model helps, we can still get good performance without it.

6.6 Defense: Adversarial Training

RAFA's attacks demonstrate effectiveness across different systems and settings. To initiate a discussion on potential defense strategies, we show that FIRE can enhance its robustness to adversarial attacks with robust training. Adversarial training involves adding adversarial examples while training [57]. In our case, since we are defending against a UAP based attack, we use our `RW-PGD` algorithm in order to compute batch-wise perturbations. We choose the `RW-PGD` algorithm since it is similar but stronger than `RW-UAP` as it assumes that the attacker has access to the inputs processed by the ML model.

During training, we compute 5 random attacks on each batch with a budget of 10%. We then apply each attack to the entire batch before learning. While this method adds overhead during training, it significantly reduces the ability of RAFA to find successful attacks. With our initial study and parameters we find that training time goes from ∼5 mins to ∼125 mins; however, we expect that further tuning could significantly reduce this training time. Fig. 7c shows the effect of RAFA at the budget of 30% on FIRE. When applying RAFA on the original model, the average channel SNR provided by FIRE drops from 17.79dB to 10.16dB. Promisingly, the robust model maintains an SNR of 14.09dB, which is 38.6% higher than original model improving the model robustness by 57%. This result highlights the potential for building robust training approaches. We delegate a detailed study of such methods to future work.

7 Related Work

Adversarial Attacks in Other Domains: Adversarial attacks have been widely studied for measuring model robustness in computer vision for tasks including object detection [80, 82], image classification [19, 20, 33], and semantic segmentation [4, 12]. Beyond vision, adversarial attacks exist for natural language processing [14, 88, 91], reinforcement learning [25, 52, 63], and graph classification [92, 96]. While most of these works only expose theoretical vulnerabilities as the generated attacks are not physically realizable, recent studies show that real-world adversarial attacks are possible. The works of [5, 22, 46, 49, 56, 71, 82] generate real-world adversarial examples for the models in the vision domain. Compared to the wireless setting, there is less signal distortion and hardware imperfections in the vision domain. The authors in [50] attack voice assistant systems such as Alexa, but their attack generates loud guitar music which is easy to detect and defend against. [83] proposes the basic structure of robust adversarial attacks in vision domain, we extend this into wireless domain by modeling the effect of wireless transformations and further test them out in real-world rather than purely simulation. To the best of our knowledge, we are the first to consider real-world attacks in wireless systems.

ML-based Wireless Systems: Machine learning has been extensively used in different tasks in wireless systems including both sensing and communication. In sensing, ML has been leveraged for human motion sensing [3, 95], sleep monitoring [30, 51, 89], emotion detection [94], indoor positioning [1, 6, 15, 75], etc. In communication, ML is also widely used in MIMO systems [31, 45, 60], modulation and signal classification [54, 78], resource allocation and management, and MAC protocol design [13, 35, 86]. In this paper, we limit our analysis to physical layer ML-models.

Adversarial Attacks against Wireless Systems: Recent work has shown theoretical attacks [2, 7, 9, 17, 18, 23, 43, 68–70] on ML-based wireless systems. However, none of these are feasible in the real world as they consider unrealistic threat models such as no distortion exists during transmission or the availability of coordination between the base station and the attacker. The closest work to ours is [8], where the authors use generative models [26] to obtain universal perturbations. They demonstrate the attack on simulated data and are not feasible in the real-world because the attack model does not account for: (a) the effect of the wireless channel on adversarial noise, (b) the lack of time-synchronization between a client's and adversary's transmissions. Our work is the first to demonstrate real-world hardware-implemented adversarial attacks by explicitly incorporating robustness to real-world channel transformations and un-synchronized transmissions.

8 Concluding Discussion

We present RAFA, the first real-world adversarial attack design on machine learning-based wireless systems. Our results show that adversarial attacks are feasible in the real-world, in spite of channel distortions, hardware noise, and black-box assumptions. We conclude with some directions that future work may consider for expanding on our paper:

- **More Capable Adversary:** We consider a single antenna adversary and show its feasibility in conducting real world attack. A multi-antenna adversary has more degrees of freedom and can cause more damage. It also opens up new questions on synchronization between different antennas, the tradeoffs between antenna count, efficiency, etc.
- **Higher Layer Attacks:** We focus on physical layer ML systems. We envision future work will consider attacks at higher layers (e.g., MAC), which can explore new modalities such as frame injection attacks.
- **Robust Training and Other Defenses:** How do we train models that are not prone to adversarial attacks? We show it is feasible to defend, but can this be made faster and more robust? Adversarial training provides empirical robustness, can we provide formal guarantees on when a model does or does not work? Finally, can we design cross-layer defense mechanisms that are robust to attacks in the physical layer?

Acknowledgements – We are grateful to the Qualcomm Innovation Fellowship program and NSF RINGS Award 2148583 for supporting this work. We also thank the reviewers and our shepherd, Fadel Adib, for constructive feedback.

References

[1] F. Adib, Z. Kabelac, D. Katabi, and R. C. Miller. 3d tracking via body radio reflections. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 317–329, 2014.

[2] A. Albaseer, B. S. Ciftler, and M. M. Abdallah. Performance evaluation of physical attacks against E2E autoencoder over rayleigh fading channel. In *proc. IEEE International Conference on Informatics, IoT, and Enabling Technologies, ICIoT*, pages 177–182. IEEE, 2020.

[3] M. A. Alsheikh, S. Lin, D. Niyato, and H.-P. Tan. Machine learning in wireless sensor networks: Algorithms, strategies, and applications. *IEEE Communications Surveys & Tutorials*, 16(4):1996–2018, 2014.

[4] A. Arnab, O. Miksik, and P. H. Torr. On the robustness of semantic segmentation models to adversarial attacks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 888–897, 2018.

[5] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok. Synthesizing robust adversarial examples. In *International conference on machine learning*, pages 284–293. PMLR, 2018.

[6] R. Ayyalasomayajula, A. Arun, C. Wu, S. Sharma, A. R. Sethi, D. Vasisht, and D. Bharadia. Deep learning based wireless localization for indoor navigation. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[7] A. Bahramali, M. Nasr, A. Houmansadr, D. Goeckel, and D. Towsley. Robust adversarial attacks against dnn-based wireless communication systems. In *ACM Conference on Computer and Communications Security (CCS)*, pages 126–140. ACM, 2021.

[8] A. Bahramali, M. Nasr, A. Houmansadr, D. Goeckel, and D. Towsley. Robust adversarial attacks against dnn-based wireless communication systems. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 126–140, 2021.

[9] S. Bair, M. DelVecchio, B. Flowers, A. J. Michaels, and W. C. Headley. On the limitations of targeted adversarial evasion attacks against deep learning enabled modulation recognition. In *Proc. ACM Workshop on Wireless Security and Machine Learning, WiseML@WiSec*, pages 25–30. ACM, 2019.

[10] A. Bakshi, Y. Mao, K. Srinivasan, and S. Parthasarathy. Fast and efficient cross band channel prediction using machine learning. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[11] M. Balunovic and M. T. Vechev. Adversarial training and provable defenses: Bridging the gap. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[12] V. Besnier, A. Bursuc, D. Picard, and A. Briot. Triggering failures: Out-of-distribution detection by learning from local adversarial attacks in semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15701–15710, 2021.

[13] N. Z. binti Zubir, A. F. Ramli, and H. Basarudin. Optimization of wireless sensor networks mac protocols using machine learning; a survey. In *2017 International Conference on Engineering Technology and Technopreneurship (ICE2T)*, pages 1–5. IEEE, 2017.

[14] N. Boucher, I. Shumailov, R. Anderson, and N. Papernot. Bad characters: Imperceptible nlp attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1987–2004. IEEE, 2022.

[15] S. Bozkurt, G. Elibol, S. Gunal, and U. Yayan. A comparative study on machine learning algorithms for indoor positioning. In *2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA)*, pages 1–8. IEEE, 2015.

[16] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy*, pages 39–57. IEEE Computer Society, 2017.

[17] M. DelVecchio, V. Arndorfer, and W. C. Headley. Investigating a spectral deception loss metric for training machine learning-based evasion attacks. *CoRR*, abs/2005.13124, 2020.

[18] M. DelVecchio, B. Flowers, and W. C. Headley. Effects of forward error correction on communications aware evasion attacks. *CoRR*, abs/2005.13123, 2020.

[19] D. I. Dimitrov, G. Singh, T. Gehr, and M. T. Vechev. Provably robust adversarial examples. In *Proc. International Conference on Learning Representations, ICLR*. OpenReview.net, 2022.

[20] Y. Dong, Q.-A. Fu, X. Yang, T. Pang, H. Su, Z. Xiao, and J. Zhu. Benchmarking adversarial robustness on image classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 321–331, 2020.

[21] ettus. USRP X310. https://www.ettus.com/all-products/x310-kit/.

[22] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1625–1634, 2018.

[23] B. Flowers, R. M. Buehrer, and W. C. Headley. Communications aware adversarial residual networks for over the air evasion attacks. In *Proc. IEEE Military Communications Conference, MILCOM*, pages 133–140. IEEE, 2019.

[24] M.-G. Garcia and J. M. Páez-Borrallo. Tracking of time misalignments for ofdm systems in multipath fading channels. *IEEE Transactions on Consumer Electronics*, 48(4):982–989, 2002.

[25] A. Gleave, M. Dennis, C. Wild, N. Kant, S. Levine, and S. Russell. Adversarial policies: Attacking deep reinforcement learning. *arXiv preprint arXiv:1905.10615*, 2019.

[26] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Commun. ACM*, 63(11):139–144, oct 2020.

[27] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[28] B. Gopalakrishnan and N. Jindal. An analysis of pilot contamination on multi-user mimo cellular systems with many antennas. In *2011 IEEE 12th international workshop on signal processing advances in wireless communications*, pages 381–385. IEEE, 2011.

[29] F. Gringoli, M. Schulz, J. Link, and M. Hollick. Free your csi: A channel state information extraction platform for modern wi-fi chipsets. In *Proceedings of the 13th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization*, WiNTECH '19, page 21–28, 2019.

[30] Y. Gu, Y. Wang, Z. Liu, J. Liu, and J. Li. Sleepguardian: An rf-based healthcare system guarding your sleep from afar. *IEEE Network*, 34(2):164–171, 2020.

[31] D. He, C. Liu, T. Q. Quek, and H. Wang. Transmit antenna selection in mimo wiretap channels: A machine learning approach. *IEEE Wireless Communications Letters*, 7(4):634–637, 2018.

[32] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[33] H. Hirano, A. Minagi, and K. Takemoto. Universal adversarial attacks on deep neural networks for medical image classification. *BMC medical imaging*, 21(1):1–13, 2021.

[34] R. Jia, A. Raghunathan, K. Göksel, and P. Liang. Certified robustness to adversarial word substitutions. In K. Inui, J. Jiang, V. Ng, and X. Wan, editors, *Proc. Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*, pages 4127–4140. Association for Computational Linguistics, 2019.

[35] S. Jog, Z. Liu, A. Franques, V. Fernando, S. Abadal, J. Torrellas, and H. Hassanieh. One protocol to rule them all: Wireless {Network-on-Chip} using deep reinforcement learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 973–989, 2021.

[36] J. Jose, A. Ashikhmin, T. L. Marzetta, and S. Vishwanath. Pilot contamination problem in multi-cell tdd systems. In *2009 IEEE International Symposium on Information Theory*, pages 2184–2188. IEEE, 2009.

[37] J. Jose, A. Ashikhmin, T. L. Marzetta, and S. Vishwanath. Pilot contamination and precoding in multi-cell tdd systems. *IEEE Transactions on Wireless Communications*, 10(8):2640–2651, 2011.

[38] S. Kagi and B. S. Mathapati. Localization in wireless sensor network using machine learning optimal trained deep neural network by parametric analysis. *Measurement: Sensors*, page 100427, 2022.

[39] I. Karmanov, F. G. Zanjani, I. Kadampot, S. Merlin, and D. Dijkman. Wicluster: Passive indoor 2d/3d positioning using wifi without precise labels. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2021.

[40] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[41] R. A. Kennedy, D. B. Ward, and T. D. Abhayapala. Nearfield beamforming using radial reciprocity. *IEEE Transactions on Signal Processing*, 47(1):33–40, 1999.

[42] B. Kim, Y. Sagduyu, K. Davaslioglu, T. Erpek, and S. Ulukus. Adversarial machine learning for nextg covert communications using multiple antennas. *Entropy*, 24(8):1047, 2022.

[43] B. Kim, Y. E. Sagduyu, K. Davaslioglu, T. Erpek, and S. Ulukus. Channel-aware adversarial attacks against deep learning-based wireless signal classifiers. *IEEE Trans. Wirel. Commun.*, 21(6):3868–3880, 2022.

[44] B. Kim, Y. E. Sagduyu, T. Erpek, K. Davaslioglu, and S. Ulukus. Adversarial attacks with multiple antennas against deep learning-based modulation classifiers. In *2020 IEEE Globecom Workshops (GC Wkshps*, pages 1–6. IEEE, 2020.

[45] A. Klautau, P. Batista, N. González-Prelcic, Y. Wang, and R. W. Heath. 5g mimo data for machine learning: Application to beam-selection using deep learning. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–9. IEEE, 2018.

[46] A. Kurakin, I. Goodfellow, S. Bengio, et al. Adversarial examples in the physical world, 2016.

[47] L. Lan, G. Liao, J. Xu, Y. Zhang, and B. Liao. Transceive beamforming with accurate nulling in fda-mimo radar for imaging. *IEEE Transactions on Geoscience and Remote Sensing*, 58(6):4145–4159, 2020.

[48] J. Li, S. Qu, X. Li, J. Szurley, J. Z. Kolter, and F. Metze. Adversarial music: Real world audio adversary against wake-word detection system. In *Proc. Neural Information Processing Systems (NeurIPS)*, pages 11908–11918, 2019.

[49] J. Li, F. R. Schmidt, and J. Z. Kolter. Adversarial camera stickers: A physical camera-based attack on deep learning systems. In *Proc. International Conference on Machine Learning, ICML*, volume 97, pages 3896–3904, 2019.

[50] J. B. Li, S. Qu, X. Li, J. Szurley, J. Z. Kolter, and F. Metze. Adversarial music: Real world audio adversary against wake-word detection system. *arXiv preprint arXiv:1911.00126*, 2019.

[51] C.-T. Lin, M. Prasad, C.-H. Chung, D. Puthal, H. El-Sayed, S. Sankar, Y.-K. Wang, J. Singh, and A. K. Sangaiah. Iot-based wireless polysomnography intelligent system for sleep monitoring. *IEEE Access*, 6:405–414, 2017.

[52] Y.-C. Lin, Z.-W. Hong, Y.-H. Liao, M.-L. Shih, M.-Y. Liu, and M. Sun. Tactics of adversarial attack on deep reinforcement learning agents. *arXiv preprint arXiv:1703.06748*, 2017.

[53] D. Liu, W. Ma, S. Shao, Y. Shen, and Y. Tang. Performance analysis of tdd reciprocity calibration for massive mu-mimo systems with zf beamforming. *IEEE Communications Letters*, 20(1):113–116, 2015.

[54] X. Liu, C. Zhao, P. Wang, Y. Zhang, and T. Yang. Blind modulation classification algorithm based on machine learning for spatially correlated mimo system. *IET Communications*, 11(7):1000–1007, 2017.

[55] Z. Liu, G. Singh, C. Xu, and D. Vasisht. Fire: enabling reciprocity for fdd mimo systems. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 628–641, 2021.

[56] B. Luo, Y. Liu, L. Wei, and Q. Xu. Towards imperceptible and robust adversarial example attacks against neural networks. In *Thirty-second aaai conference on artificial intelligence*, 2018.

[57] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *Proc. International Conference on Learning Representations (ICLR)*, 2018.

[58] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1765–1773, 2017.

[59] I. A. Najm, A. K. Hamoud, J. Lloret, and I. Bosch. Machine learning prediction approach to enhance congestion control in 5g iot environment. *Electronics*, 8(6):607, 2019.

[60] T. J. O'Shea, T. Erpek, and T. C. Clancy. Deep learning based mimo communications. *arXiv preprint arXiv:1707.07980*, 2017.

[61] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. *Advances in neural information processing systems*, 2017.

[62] P. S. Pati, S. S. Sahoo, D. Krishnaswamy, and R. Datta. A novel machine learning approach for link adaptation in 5g wireless networks. In *2020 2nd PhD Colloquium on Ethically Driven Innovation and Technology for Society (PhD EDITS)*, pages 1–2, 2020.

[63] A. Pattanaik, Z. Tang, S. Liu, G. Bommannan, and G. Chowdhary. Robust deep reinforcement learning with adversarial attacks. *arXiv preprint arXiv:1712.03632*, 2017.

[64] Qualcomm. 5G RF. https://www.qualcomm.com/news/releases/2021/02/qualcomm-announces-next-generation-5g-rf-front-end-solutions-featuring-use.

[65] Qualcomm. X70. https://www.qualcomm.com/news/releases/2022/02/new-snapdragon-x70-modem-rf-harnesses-worlds-first-5g-ai-processor-industry.

[66] K. Rehrl and S. Gröchenig. Evaluating localization accuracy of automated driving systems. *Sensors*, 21(17):5855, 2021.

[67] T. G. Reid, S. E. Houts, R. Cammarata, G. Mills, S. Agarwal, A. Vora, and G. Pandey. Localization requirements for autonomous vehicles. *arXiv preprint arXiv:1906.01061*, 2019.

[68] M. Sadeghi and E. G. Larsson. Adversarial attacks on deep-learning based radio signal classification. *CoRR*, abs/1808.07713, 2018.

[69] M. Sadeghi and E. G. Larsson. Adversarial attacks on deep-learning based radio signal classification. *IEEE Wireless Communications Letters*, 8(1):213–216, 2018.

[70] M. Sadeghi and E. G. Larsson. Physical adversarial attacks against end-to-end autoencoder communication systems. *IEEE Communications Letters*, 23(5):847–850, 2019.

[71] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1528–1540. ACM, 2016.

[72] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev. Fast and effective robustness certification. *NeurIPS*, 1(4):6, 2018.

[73] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[74] B. Sliwa, R. Falkenberg, T. Liebig, J. Pillmann, and C. Wietfeld. Machine learning based context-predictive car-to-cloud communication using multi-layer connectivity maps for upcoming 5g networks. In *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, pages 1–7. IEEE, 2018.

[75] P. Sthapit, H.-S. Gang, and J.-Y. Pyun. Bluetooth based indoor positioning using machine learning algorithms. In *2018 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 206–212. IEEE, 2018.

[76] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[77] D. Tse and P. Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[78] F. Wang, S. Huang, H. Wang, and C. Yang. Automatic modulation classification exploiting hybrid machine learning network. *mathematical Problems in engineering*, 2018, 2018.

[79] M. Wasilewska, H. Bogucka, and A. Kliks. Spectrum sensing and prediction for 5g radio. In *Big Data Technologies and Applications*, pages 176–194. Springer, 2020.

[80] X. Wei, S. Liang, N. Chen, and X. Cao. Transferable adversarial attacks for image and video object detection. *arXiv preprint arXiv:1811.12641*, 2018.

[81] E. Wong and J. Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In J. G. Dy and A. Krause, editors, *Proc. International Conference on Machine Learning, ICML*, volume 80, pages 5283–5292, 2018.

[82] C. Xie, J. Wang, Z. Zhang, Y. Zhou, L. Xie, and A. Yuille. Adversarial examples for semantic segmentation and object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 1369–1378, 2017.

[83] C. Xu and G. Singh. Robust universal adversarial perturbations. *CoRR*, abs/2206.10858, 2022.

[84] K. Xu, Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. In *Proc. Neural Information Processing Systems (NeurIPS)*, pages 1129–1141, 2020.

[85] T. Xu, T. Zhou, J. Tian, J. Sang, and H. Hu. Intelligent spectrum sensing: When reinforcement learning meets automatic repeat sensing in 5g communications. *IEEE Wireless Communications*, 27(1):46–53, 2020.

[86] B. Yang, X. Cao, Z. Han, and L. Qian. A machine learning enabled mac framework for heterogeneous internet-of-things networks. *IEEE Transactions on Wireless Communications*, 18(7):3697–3712, 2019.

[87] R. Yang, J. Laurel, S. Misailovic, and G. Singh. Training certifiably robust neural networks against semantic perturbations. In *Proc. International Conference on Learning Representations, ICLR*. OpenReview.net, 2023.

[88] J. Y. Yoo and Y. Qi. Towards improving adversarial training of nlp models. *arXiv preprint arXiv:2109.00544*, 2021.

[89] B. Yu, Y. Wang, K. Niu, Y. Zeng, T. Gu, L. Wang, C. Guan, and D. Zhang. Wifi-sleep: sleep stage monitoring using commodity wi-fi devices. *IEEE internet of things journal*, 8(18):13900–13913, 2021.

[90] H. Zhang, H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. Boning, and C.-J. Hsieh. Towards stable and efficient training of verifiably robust neural networks. In *Proc. International Conference on Learning Representations (ICLR)*, 2020.

[91] W. E. Zhang, Q. Z. Sheng, A. Alhazmi, and C. Li. Adversarial attacks on deep-learning models in natural language processing: A survey. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 11(3):1–41, 2020.

[92] X. Zhang and M. Zitnik. Gnnguard: Defending graph neural networks against adversarial attacks. *Advances in neural information processing systems*, 33:9263–9275, 2020.

[93] Z. Zhang, M. Lee, and S. Choi. Deep-learning-based wi-fi indoor positioning system using continuous csi of trajectories. *Sensors*, 21(17):5776, 2021.

[94] M. Zhao, F. Adib, and D. Katabi. Emotion recognition using wireless signals. In *Proceedings of the 22nd annual international conference on mobile computing and networking*, pages 95–108, 2016.

[95] M. Zhao, T. Li, M. Abu Alsheikh, Y. Tian, H. Zhao, A. Torralba, and D. Katabi. Through-wall human pose estimation using radio signals. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[96] D. Zügner, O. Borchert, A. Akbarnejad, and S. Günnemann. Adversarial attacks on graph neural networks: Perturbations and their patterns. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(5):1–31, 2020.